The UNIX Time-Sharing System

Dennis M. Ritchie and Ken Thompson
Bell Laboratories

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages. This paper discusses the nature and implementation of the file system and of the user command interface.

Key Words and Phrases: time-sharing, operating system, file system, command language, PDP-11

CR Categories: 4.30, 4.32

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights. New York, October 15–17, 1973. Authors' address: Bell Laboratories, Murray Hill, NJ 07974.

The electronic version was recreated by Eric A. Brewer, University of California at Berkeley, brewer@cs.berkeley.edu. Please notify me of any deviations from the original; I have left errors in the original unchanged.

1. Introduction

There have been three versions of UNIX. The earliest version (circa 1969–70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. This paper describes only the PDP-11/40 and /45 [l] system since it is more modern and many of the differences between it and older UNIX systems result from redesign of features found to be deficient or lacking.

Since PDP-11 UNIX became operational in February 1971, about 40 installations have been put into service; they are generally smaller than the system described here. Most of them are engaged in applications such as the preparation and formatting of patent applications and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as $40,000, and less than two man years were spent on the main system software. Yet UNIX contains a number of features seldom offered even in much larger systems. It is hoped, however, the users of UNIX will find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the system proper, the major programs available under UNIX are: assembler, text editor based on QED [2], linking loader, symbolic debugger, compiler for a language resembling BCPL [3] with types and structures (C), interpreter for a dialect of BASIC, text formatting program, Fortran compiler, Snobol interpreter, top-down compiler-compiler (TMG) [4], bottom-up compiler-compiler (YACC), form letter generator, macro processor (M6) [5], and permuted index program.

There is also a host of maintenance, utility, recreation, and novelty programs. All of these programs were written locally. It is worth noting that the system is totally self-supporting. All UNIX software is maintained under UNIX; likewise, UNIX documents are generated and formatted by the UNIX editor and text formatting program.

2. Hardware and Software Environment

The PDP-11/45 on which our UNIX installation is implemented is a 16-bit word (8-bit byte) computer with 144K bytes of core memory; UNIX occupies 42K bytes. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the

software mentioned above can require as little as 50K bytes of core altogether.

The PDP-11 has a 1M byte fixed-head disk, used for file system storage and swapping, four moving-head disk drives which each provide 2.5M bytes on removable disk cartridges, and a single moving-head disk drive which uses removable 40M byte disk packs. There are also a high-speed paper tape reader-punch, nine-track magnetic tape, and D-tape (a variety of magnetic tape facility in which individual records may be addressed and rewritten). Besides the console typewriter, there are 14 variable-speed communications interfaces attached to 100-series datasets and a 201 dataset interface used primarily for spooling printout to a communal line printer. There are also several one-of-a-kind devices including a Picturephone® interface, a voice response unit, a voice synthesizer, a phototypesetter, a digital switching network, and a satellite PDP-11/20 which generates vectors, curves, and characters on a Tektronix 611 storage-tube display.

The greater part of UNIX software is written in the above-mentioned C language [6]. Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system is about one third greater than the old. Since the new system is not only much easier to understand and to modify but also includes many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we considered this increase in size quite acceptable.

3. The File System

The most important job of UNIX is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary Files

A file contains whatever information the user places on it, for example symbolic or binary (object) programs. No particular structuring is expected by the system. Files of text consist simply of a string of characters, with lines demarcated by the new-line character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure: the assembler generates and the loader expects an object file in a particular format. However, the structure of files is controlled by the programs which use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the *root* directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the root. Another system directory contains all the programs provided for general use; that is, all the *commands*. As will be seen however, it is by no means necessary that a program reside in this directory for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name*, which is a sequence of directory names separated by slashes "/" and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name /*alpha*/*beta*/*gamma* causes the system to search the root for directory *alpha*, then to search *alpha* for *beta*, finally to find *gamma* in *beta*. *Gamma* may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root itself.

A path name not starting with "/" causes the system to begin the search in the user's current directory. Thus, the name *alpha*/*beta* specifies the file named *beta* in subdirectory *alpha* of the current directory. The simplest kind of name, for example *alpha*, refers to a file which itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same nondirectory file may appear in several directories under possibly different names. This feature is called *linking*; a directory entry for a file is sometimes called a link. UNIX differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name in each directory refers to the directory itself. Thus a program may read the current directory under the name "." without knowing its complete path name. The name ".." by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries "." and "..", each directory must appear as an entry in exactly one other, which is its parent. The reason for this is to simplify the writing of programs which visit subtrees of the directory

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special Files

Special files constitute the most unusual feature of the UNIX file system. Each I/O device supported by UNIX is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory */dev*, although a link may be made to one of these files just like an ordinary file. Thus, for example, to punch paper tape, one may write on the file */dev/ppt*. Special files exist for each communication line, each disk, each tape drive, and for physical core memory. Of course, the active disks and the core special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

3.4 Removable File Systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a *mount* system request which has two arguments: the name of an existing ordinary file, and the name of a direct-access special file whose associated storage volume (e.g. disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of *mount* is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, *mount* replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the *mount*, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on the fixed-head disk, and the large disk drive, which contains user's files, is mounted by the system initialization program, the four smaller disk drives are available to users for mounting their own disk packs. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping which would otherwise be required to assure removal of the links when the removable volume is finally dismounted. In particular, in the root directories of all file systems, removable or not, the name ".." refers to the directory itself instead of to its parent.

3.5 Protection

Although the access control scheme in UNIX is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of seven protection bits. Six of these specify independently read, write, and execute permission for the owner of the file and for all other users.

If the seventh bit is on, the system will temporarily change the user identification of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program which calls for it. The set-user-ID feature provides for privileged programs which may use files inaccessible to other users. For example, a program may keep an accounting file which should neither be read nor changed except by the program itself. If the set-user-identification bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully written commands which call privileged system entries. For example, there is a system entry invocable only by the "super-user" (below) which creates an empty directory. As indicated above, directories are expected to have entries for "." and "..". The command which creates a directory is owned by the super user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for "." and "..".

Since anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available with- out administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed in [7].

The system recognizes one particular user ID (that of the "super-user") as exempt from the usual constraints on file access; thus (for example) programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O Calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between "random" and sequential I/O, nor is any logical record size imposed by the system. The size of an ordinary file is determined by the

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

highest byte written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O in UNIX, Some of the basic calls are summarized below in an anonymous language which will indicate the required parameters without getting into the complexities of machine language programming. Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

To read or write a file assumed to exist already, it must be opened by the following call:

filep = open (name, flag)

Name indicates the name of the file. An arbitrary path name may be given. The *flag* argument indicates whether the file is to be read, written, or "updated", that is read and written simultaneously.

The returned value *filep* is called a *file descriptor*. It is a small integer used to identify the file in subsequent calls to read, write, or otherwise manipulate it.

To create a new file or completely rewrite an old one, there is a *create* system call which creates the given file if it does not exist, or truncates it to zero length if it does exist. *Create* also opens the new file for writing and, like *open*, returns a file descriptor.

There are no user-visible locks in the file system, nor is there any restriction on the number of users who may have a file open for reading or writing; although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice, difficulties do not arise. We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes. They are insufficient because locks in the ordinary sense, whereby one user is prevented from writing on a file which another user is reading, cannot prevent confusion when, for example, both users are editing a file with an editor which makes a copy of the file being edited.

It should be said that the system has sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in such inconvenient activities as writing on the same file, creating files in the same directory or deleting each other's open files.

Except as indicated below, reading and writing are sequential. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the first following byte. For each open file there is a pointer, maintained by the system, which indicates the next byte to be read or written. If n bytes are read or written, the pointer advances by n bytes.

Once a file is open, the following calls may be used:

n = read(filep, buffer, count)
n = write(filep, buffer, count)

Up to *count* bytes are transmitted between the file specified by *filep* and the byte array specified by *buffer*. The returned value *n* is the number of bytes actually transmitted. In the *write* case, *n* is the same as *count* except under exceptional conditions like I/O errors or end of physical medium on special files; in a read, however, *n* may without error be less than *count*. If the read pointer is so near the end of the file that reading count characters would cause reading beyond the end, only sufficient bytes are transmitted to reach the end of the file; also, typewriter-like devices never return more than one line of input. When a *read* call returns with *n* equal to zero, it indicates the end of the file. For disk files this occurs when the read pointer becomes equal to the current size of the file. It is possible to generate an end-of-file from a typewriter by use of an escape sequence which depends on the device used.

Bytes written on a file affect only those implied by the position of the write pointer and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file is grown as needed.

To do random (direct access) I/O, it is only necessary to move the read or write pointer to the appropriate location in the file.

location = seek(filep, base, offset)

The pointer associated with *filep* is moved to a position *offset* bytes from the beginning of the file, from the current position of the pointer, or from the end of the file, depending on *base*. *Offset* may be negative. For some devices (e.g. paper tape and typewriters) seek calls are ignored. The actual offset from the beginning of the file to which the pointer was moved is returned in *location*.

3.6.1 Other I/O Calls. There are several additional system entries having to do with I/O and with the file system which will not be discussed. For example: close a file, get the status of a file, change the protection mode or the owner of a file, create a directory, make a link to an existing file, delete a file.

4. Implementation of the File System

As mentioned in §3.2 above, a directory entry contains only a name for the associated file and a pointer to the file itself. This pointer is an integer called the *i-number* (for index number) of the file. When the file is accessed, its *i-number* is used as an index into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry thereby found (the file's *i-node*) contains the description of the file as follows.

1. Its owner.
2. Its protection bits.
3. The physical disk or tape addresses for the file contents.
4. Its size.

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

5. Time of last modification
6. The number of links to the file, that is, the number of times it appears in a directory.
7. A bit indicating whether the file is a directory.
8. A bit indicating whether the file is a special file.
9. A bit indicating whether the file is "large" or "small."

The purpose of an *open* or *create* system call is to turn the path name given by the user into an i-number by searching the explicitly or implicitly named directories. Once a file is open, its device, i-number, and read/write pointer are stored in a system table indexed by the file descriptor returned by the *open* or *create*. Thus the file descriptor supplied during a subsequent call to read or write the file may be easily related to the information necessary to access the file.

When a new file is created, an i-node is allocated for it and a directory entry is made which contains the name of the file and the i-node number. Making a link to an existing file involves creating a directory entry with the new name, copying the i-number from the original file entry, and incrementing the link-count field of the i-node. Removing (deleting) a file is done by decrementing the link-count of the i-node specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the i-node is deallocated.

The space on all fixed or removable disks which contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit which depends on the device. There is space in the i-node of each file for eight device addresses. A *small* (nonspecial) file fits into eight or fewer blocks; in this case the addresses of the blocks themselves are stored. For *large* (nonspecial) files, each of the eight device addresses may point to an indirect block of 256 addresses of blocks constituting the file itself. These files may be as large as 8·256·512, or l,048,576 (2^{20}) bytes.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last seven device address words are immaterial, and the list is interpreted as a pair of bytes which constitute an internal *device* name. These bytes specify respectively a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar typewriter interfaces.

In this environment, the implementation of the *mount* system call (§3.4) is quite straightforward. *Mount* maintains a system table whose argument is the i-number and device name of the ordinary file specified during the *mount*, and whose corresponding value is the device name of the indicated special file. This table is searched for each (i-number, device)-pair which turns up while a path name is being scanned during an *open* or *create*; if a match is found, the i-number is replaced by 1 (which is the i-number of the root directory on all file systems), and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is immediately after return from a *read* call the data are available, and conversely after a *write* the user's workspace may be reused. In fact the system maintains a rather complicated buffering mechanism which reduces greatly the number of I/O operations required to access a file. Suppose a *write* call is made specifying transmission of a single byte.

UNIX will search its buffers to see whether the affected disk block currently resides in core memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer, and an entry is made in a list of blocks to be written. The return from the *write* call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

A program which reads or writes files in units of 512 bytes has an advantage over a program which reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. A program which is used rarely or which does no great volume of I/O may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name which is related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, since it need only scan the linearly-organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, since all directory entries for a file have equal status. Charging the owner of a file is unfair, in general, since one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. The current version of UNIX avoids the issue by not charging any fees at all.

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

4.1 Efficiency of the File System

To provide an indication of the overall efficiency of UNIX and of the file system in particular, timings were made of the assembly of a 7621-line program. The assembly was run alone on the machine; the total clock time was 35.9 sec, for a rate of 212 lines per sec. The time was divided as follows: 63.5 percent assembler execution time, 16.5 percent system overhead, 20.0 percent disk wait time. We will not attempt any interpretation of these figures nor any comparison with other systems, but merely note that we are generally satisfied with the overall performance of the system.

5. Processes and Images

An *image* is a computer execution environment. It includes a core image, general register values, status of open files, current directory, and the like. An image is the current state of a pseudo computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in core; during the execution of other processes it remains in core unless the appearance of an active, higher-priority process forces it to be swapped out to the fixed-head disk.

The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

5.1 Processes

Except while UNIX is bootstrapping itself into operation, a new process can come into existence only by use of the fork system call:

processid = fork (label)

When *fork* is executed by a process, it splits into two independently executing processes. The two processes have independent copies of the original core image, and share any open files. The new processes differ only in that one is considered the parent process: in the parent, control returns directly from the *fork*, while in the child, control is passed to location *label*. The *processid* returned by the *fork* call is the identification of the other process.

Because the return points in the parent and child process are not the same, each image existing after a *fork* may determine whether it is the parent or child process.

5.2 Pipes

Processes may communicate with related processes using the same system *read* and *write* calls that are used for file system I/O. The call

filep = pipe( )

returns a file descriptor *filep* and creates an interprocess channel called a *pipe*. This channel, like other open flies, is passed from parent to child process in the image by the *fork* call. A *read* using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although interprocess communication via pipes is a quite valuable tool (see §6.2), it is not a completely general mechanism since the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of Programs

Another major system primitive is invoked by

execute(file, arg_1, arg_2, ..., arg_n)

which requests the system to read in and execute the program named by *file*, passing it string arguments arg_1, arg_2, ..., arg_n. Ordinarily, arg_1 should be the same string as *file*, so that the program may determine the name by which it was invoked. All the code and data in the process using *execute* is replaced from the file, but open files, current directory, and interprocess relationships are unaltered. Only if the call fails, for example because *file* could not be found or because its execute-permission bit was not set, does a return take place from the *execute* primitive; it resembles a "jump" machine instruction rather than a subroutine call.

5.4 Process Synchronization

Another process control system call

processid = wait( )

causes its caller to suspend execution until one of its children has completed execution. Then *wait* returns the *processid* of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available. *Wait* may also present status from a grandchild or more distant ancestor; see §5.5.

5.5 Termination

Lastly,

exit (status)

terminates a process, destroys its image, closes its open files, and generally obliterates it. When the parent is notified through the *wait* primitive, the indicated *status* is available to the parent; if the parent has already terminated, the status is available to the grandparent, and so on. Processes

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

may also terminate as a result of various illegal actions or user-generated signals (§7 below).

6. The Shell

For most users, communication with UNIX is carried on with the aid of a program called the Shell. The Shell is a command line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

command arg_1 $\text{arg}_2 \cdots \text{arg}_n$

The Shell splits up the command name and the arguments into separate strings. Then a file with name *command* is sought; *command* may be a path name including the "/" character to specify any file in the system. If *command* is found, it is brought into core and executed. The arguments collected by the Shell are accessible to the command. When the command is finished, the Shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file *command* cannot be found, the Shell prefixes the string /*bin*/ to command and attempts again to find the file. Directory /*bin* contains all the commands intended to be generally used.

6.1 Standard I/O

The discussion of I/O in §3 above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the Shell, however, start off with two open files which have file descriptors 0 and 1. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user's typewriter. Thus programs which wish to write informative or diagnostic information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs which wish to read messages typed by the user usually read this file.

The Shell is able to change the standard assignments of these file descriptors from the user's typewriter printer and keyboard. If one of the arguments 'to a command is prefixed by "⟩", file descriptor 1 will, for the duration of the command, refer to the file named after the "⟩". For example,

ls

ordinarily lists, on the typewriter, the names of the files in the current directory. The command

ls ⟩there

creates a file called *there* and places the listing there. Thus the argument "⟩there" means, "place output on *there*." On the other hand,

ed

ordinarily enters the editor, which takes requests from the user via his typewriter. The command

ed ⟨script

interprets *script* as a file of editor commands; thus "⟨script" means, "take input from *script*."

Although the file name following "⟨" or "⟩" appears to be an argument to the command, in fact it is interpreted completely by the Shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the Shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line

ls | pr –2 | opr

ls lists the names of the files in the current directory; its output is passed to *pr*, which paginates its input with dated headings. The argument "–2" means double column. Likewise the output from *pr* is input to *opr*. This command spools its input onto a file for off-line printing.

This process could have been carried out more clumsily by

ls ⟩temp1
pr –2 ⟨temp1 ⟩temp2
opr ⟨temp2

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the *ls* command to accept user requests to paginate its output, to print in multicolumn format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as *ls* to provide such a wide variety of output options.

A program such as *pr* which copies its standard input to its standard output (with processing) is called a *filter*. Some filters which we have found useful perform character transliteration, sorting of the input, and encryption and decryption.

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

6.3 Command Separators: Multitasking

Another feature provided by the Shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons.

```
ls; ed
```

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by "&", the Shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example,

```
as source >output &
```

causes source to be assembled, with diagnostic output going to output; no matter how long the assembly takes, the Shell returns immediately. When the Shell does not wait for the completion of a command, the identification of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The "&" may be used several times in a line:

```
as source >output & ls >files &
```

does both the assembly and the listing in the background. In the examples above using "&", an output file other than the typewriter was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The Shell also allows parentheses in the above operations. For example,

```
(date; ls) >x &
```

prints the current date and time followed by a list of the current directory onto the file *x*. The Shell also returns immediately for another request.

6.4 The Shell as a Command: Command files

The Shell is itself a command, and may be called recursively. Suppose file *tryout* contains the lines

```
as source
mv a.out testprog
testprog
```

The *mv* command causes the file *a.out* to be renamed *testprog*. *a.out* is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the console, *source* would be assembled, the resulting program named *testprog*, and *testprog* executed. When the lines are in *tryout*, the command

```
sh <tryout
```

would cause the Shell *sh* to execute the commands sequentially.

The Shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It is also possible to execute commands conditionally on character string comparisons or on existence of given files and to perform transfers of control within filed command sequences.

6.5 Implementation of the Shell

The outline of the operation of the Shell can now be understood. Most of tile time, the Shell is waiting for the user to type a command. When the new-line character ending the line is typed, the Shell's *read* call returns. The Shell analyzes the command line, putting the arguments in a form appropriate for *execute*. Then *fork* is called. The child process, whose code of course is still that of the Shell, attempts to perform an *execute* with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the *fork*, which is the parent process, *waits* for the child process to die. When this happens, the Shell knows the command is finished, so it types its prompt and reads the typewriter to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line contains "&", the Shell merely refrains from waiting for the process which it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the *fork* primitive, it inherits not only the core image of its parent but also all the files currently open in its parent, including those with file descriptors 0 and 1. The Shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children—the command programs—inherit them automatically. When an argument with "<" or ">" is given however, the offspring process, just before it performs *execute*, makes the standard I/O file descriptor 0 or 1 respectively refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is *open*ed (or *create*d); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after "<" or ">" and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the Shell need not know the actual names of the files which are its own standard input and output since it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the Shell never terminates. (The main loop includes that branch of the return from *fork* belonging to the parent process; that is, the branch which does a *wait*, then reads another command line.) The one thing which causes the Shell to terminate is discovering an end-of-file condition on its input file. Thus,

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

when the Shell is executed as a command with a given input file, as in

sh ⟨comfile

the commands in *comfile* will be executed until the end of *comfile* is reached; then the instance of the Shell invoked by *sh* will terminate. Since this Shell process is the child of another instance of the Shell, the *wait* executed in the latter will return, and another command may be processed.

6.6 Initialization

The instances of the Shell to which users type commands are themselves children of another process. The last step in the initialization of UNIX is the creation of a single process and the invocation (via *execute*) of a program called *init*. The role of *init* is to create one process for each typewriter channel which may be dialed up by a user. The various subinstances of *init* open the appropriate typewriters for input and output. Since when *init* was invoked there were no files open, in each process the typewriter keyboard will receive file descriptor 0 and the printer file descriptor 1. Each process types out a message requesting that the user log in and waits, reading the typewriter, for a reply. At the outset, no one is logged in, so each process simply hangs. Finally someone types his name or other identification. The appropriate instance of *init* wakes up, receives the log-in line, and reads a password file. If the user name is found, and if he is able to supply the correct password, *init* changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an *execute* of the Shell. At this point the Shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of *init* (the parent of all the subinstances of itself which will later become Shells) does a *wait*. If one of the child processes terminates, either because a Shell found an end of file or because a user typed an incorrect name or password, this path of *init* simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another login message. Thus a user may log out simply by typing the end-of-file sequence in place of a command to the Shell.

6.7 Other Programs as Shell

The Shell as described above is designed to allow users full access to the facilities of the system since it will invoke the execution of any program with appropriate protection mode. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged.

Recall that after a user has successfully logged in by supplying his name and password, *init* ordinarily invokes the Shell to interpret command lines. The user's entry in tile password file may contain the name of a program to be invoked after login instead of the Shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system specify that the editor *ed* is to be used instead of the Shell. Thus when editing system users log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking UNIX programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g. chess, blackjack, 3D tic-tac-toe) available on UNIX illustrate a much more severely restricted environment. For each of these an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the Shell. People who log in as a player of one of the games find themselves limited to the game and unable to investigate the presumably more interesting offerings of UNIX as a whole.

7. Traps

The PDP-11 hardware detects a number of program faults, such as references to nonexistent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. When an illegal action is caught, unless other arrangements have been made, the system terminates the process and writes the user's image on file *core* in the current directory. A debugger can be used to determine the state of the program at the time of the fault.

Programs which are looping, which produce unwanted output, or about which the user has second thoughts may be halted by the use of the *interrupt* signal, which is generated by typing the "delete" character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a core image file.

There is also a *quit* signal which is used to force a core image to be produced. Thus programs which loop unexpectedly may be halted and the core image examined without prearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by the process. For example, the Shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating point hardware, unimplemented instructions are caught, and floating point instructions are interpreted.

8. Perspective

Perhaps paradoxically, the success of UNIX is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

facilities, discovered a little-used system PDP-7 and set out to create a more hospitable environment. This essentially personal effort was sufficiently successful to gain the interest of the remaining author and others, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. Then in turn the 11/20 was outgrown, UNIX had proved useful enough to persuade management to invest in the PDP-11/45. Our goals throughout the effort, when articulated at all, have always concerned themselves with building a comfortable relationship with the machine and with exploring ideas and inventions in operating systems. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations which influenced the design of UNIX are visible in retrospect.

First, since we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly designed interactive system is much more productive and satisfying to use than a "batch" system. Moreover such a system is rather easily adaptable to noninteractive use, while the converse is not true.

Second there have always been fairly severe size constraints on the system and its software. Given the partiality antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy but a certain elegance of design. This may be a thinly disguised version of the "salvation through suffering" philosophy, but in our case it worked.

Third, nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Since all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large "access method" routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, only tens of instructions long, which buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no "control blocks" with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable from a space-efficiency standpoint to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load routines for dealing with each device with all programs, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process control scheme and command interface have proved both convenient and efficient. Since the Shell operates as an ordinary, swappable user program, it consumes no wired-down space in the system proper, and it may be made as powerful as desired at little cost, In particular, given the framework in which the Shell executes as a process which spawns other processes to perform commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

8.1 Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The *fork* operation, essentially as we implemented it, was present in the Berkeley time-sharing system [8]. On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls [9] and both the name of the Shell and its general functions, The notion that the Shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX [10].

9. Statistics

The following statistics from UNIX are presented to show the scale of the system and to show how a system of this scale is used. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important "applications" programs.

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

9.1 Overall

72 user population
14 maximum simultaneous users
300 directories
4400 files
34000 512-byte secondary storage blocks used

9.2 Per day (24-hour day, 7-day week basis)

There is a "background" process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant $e - 2$, and is now generating composite pseudoprimes (base 2).

1800 commands
4.3 CPU hours (aside from background)
70 connect hours
30 different users
75 logins

9.3 Command CPU Usage (cut off at 1%)

15.7%	C compiler	1.7%	Fortran compiler
15.2%	users' programs	1.6%	remove file
11.7%	editor	1.6%	tape archive
5.8%	Shell (used as a command, including command times)	1.6%	file system consistency check
		1.4%	library maintainer
5.3%	chess	1.3%	concatenate/print files
3.3%	list directory	1.3%	paginate and print file
3.1%	document formatter	1.1%	print disk usage
1.6%	backup dumper	1.0%	copy file
1.8%	assembler		

9.4 Command Accesses (cut off at 1%)

15.3%	editor	1.6%	debugger
9.6%	list directory	1.6%	Shell (used as a command)
6.3%	remove file	1.5%	print disk availability
6.3%	C compiler	1.4%	list processes executing
6.0%	concatenate/print file	1.4%	assembler
6.0%	users' programs	1.4%	print arguments
3.3%	list people logged on system	1.2%	copy file
		1.1%	paginate and print file
3.2%	rename/move file	1.1%	print current date/time
3.1%	file status	1.1%	file system consistency check
1.8%	library maintainer		
1.8%	document formatter	1.0%	tape archive
1.6%	execute another command conditionally		

9.5 Reliability

Our statistics on reliability are much more subjective than the others. The following results are true to the best of our combined recollections. The time span is over one year with a very early vintage 11/45.

There has been one loss of a file system (one disk out of five) caused by software inability to cope with a hard ware problem causing repeated power fail traps. Files on that disk were backed up three days.

A "crash" is an unscheduled system reboot or halt. There is about one crash every other day; about two-thirds of them are caused by hardware-related difficulties such as power dips and inexplicable processor interrupts to random locations. The remainder are software failures. The longest uninterrupted up time was about two weeks. Service calls average one every three weeks, but are heavily clustered. Total up time has been about 98 percent of our 24-hour, 365-day schedule.

Acknowledgments. We are grateful to R.H. Canaday, L.L. Cherry, and L.E. McMahon for their contributions to UNIX. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M.D. McIlroy, and J.F. Ossanna.

References

1. Digital Equipment Corporation. PDP-11/40 Processor Handbook, 1972, and PDP-11/45 Processor Handbook. 1971.
2. Deutsch, L.P., and Lampson, B.W. An online editor. *Comm. ACM 10*, 12 (Dec, 1967) 793–799, 803.
3. Richards, M. BCPL: A tool for compiler writing and system programming. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., pp. 557–566.
4. McClure, R.M. TMG—A syntax directed compiler. Proc. ACM 20th Nat. Conf., ACM, 1965, New York, pp. 262–274.
5. Hall. A.D. The M6 macroprocessor. Computing Science Tech. Rep. #2, Bell Telephone Laboratories, 1969.
6. Ritchie, D.M. C reference manual. Unpublished memorandum, Bell Telephone Laboratories, 1973.
7. Aleph-null. Computer Recreations. *Software Practice and Experience 1*, 2 (Apr.–June 1971), 201–204.
8. Deutsch, L.P., and Lampson, B.W. SDS 930 time-sharing system preliminary reference manual. Doc. 30.10.10, Project GENIE, U of California at Berkeley, Apr. 1965.
9. Feiertag. R.J., and Organick, E.I. The Multics input-output system. Proc. Third Symp. on Oper. Syst. Princ., Oct. 18–20, 1971, ACM, New York, pp. 35–41.
10. Bobrow, D.C., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time sharing system for the PDP-10. *Comm. ACM 15*, 3 (Mar. 1972) 135–143.

Electronic version recreated by Eric A. Brewer
University of California at Berkeley

A History and Evaluation of System R

Donald D. Chamberlin
Morton M. Astrahan
Michael W. Blasgen
James N. Gray
W. Frank King
Bruce G. Lindsay
Raymond Lorie
James W. Mehl
Thomas G. Price
Franco Putzolu
Patricia Griffiths Selinger
Mario Schkolnick
Donald R. Slutz
Irving L. Traiger
Bradford W. Wade
Robert A. Yost

IBM Research Laboratory
San Jose, California

SUMMARY: System R, an experimental database system, was constructed to demonstrate that the usability advantages of the relational data model can be realized in a system with the complete function and high performance required for everyday production use. This paper describes the three principal phases of the System R project and discusses some of the lessons learned from System R about the design of relational systems and database systems in general.

1. Introduction

Throughout the history of information storage in computers, one of the most readily observable trends has been the focus on data independence. C.J. Date [27] defined data independence as "immunity of applications to change in storage structure and access strategy." Modern database systems offer data independence by providing a high-level user interface through which users deal with the information content of their data, rather than the various bits, pointers, arrays, lists, etc. which are used to represent that information. The system assumes responsibility for choosing an appropriate internal representation for the information; indeed, the representation of a given fact may change over time without users being aware of the change.

The relational data model was proposed by E.F. Codd [22] in 1970 as the next logical step in the trend toward data independence. Codd observed that conventional database systems store information in two ways: (1) by the contents of records stored in the database, and (2) by the ways in which these records are connected together. Different systems use various names for the connections among records, such as links, sets, chains, parents, etc. For example, in Figure 1(a), the fact that supplier Acme supplies bolts is represented by connections between the relevant part and supplier records. In such a system, a user frames a question, such as "What is the lowest price for bolts?", by writing a program which "navigates" through the maze of connections until it arrives at the answer to the question. The user of a "navigational" system has the burden (or opportunity) to specify exactly how the query is to be processed; the user's algorithm is then embodied in a program which is dependent on the data structure that existed at the time the program was written.

Relational database systems, as proposed by Codd, have two important properties: (1) all information is

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
Key words and phrases: database management systems, relational model, compilation, locking, recovery, access path selection, authorization
CR Categories: 3.50, 3.70, 3.72, 4.33, 4.6
Authors' address: D. D. Chamberlin et al., IBM Research Laboratory, 5600 Cottle Road, San Jose, California 95193.
© 1981 ACM 0001-0782/81/1000-0632 75¢.

represented by data values, never by any sort of "connections" which are visible to the user; (2) the system supports a very high-level language in which users can frame requests for data without specifying algorithms for processing the requests. The relational representation of the data in Figure 1(a) is shown in Figure 1(b). Information about parts is kept in a PARTS relation in which each record has a "key" (unique identifier) called PARTNO. Information about suppliers is kept in a SUPPLIERS relation keyed by SUPPNO. The information which was formerly represented by connections between records is now contained in a third relation, PRICES, in which parts and suppliers are represented by their respective keys. The question "What is the lowest price for bolts?" can be framed in a high-level language like SQL [16] as follows:

```
SELECT MIN(PRICE)
FROM   PRICES
WHERE  PARTNO IN
       (SELECT PARTNO
        FROM   PARTS
        WHERE  NAME = 'BOLT');
```

A relational system can maintain whatever pointers, indices, or other access aids it finds appropriate for processing user requests, but the user's request is not framed in terms of these access aids and is therefore not dependent on them. Therefore, the system may change its data representation and access aids periodically to adapt to changing requirements without disturbing users' existing applications.

Since Codd's original paper, the advantages of the relational data model in terms of user productivity and data independence have become widely recognized. However, as in the early days of high-level programming languages, questions are sometimes raised about whether or not an automatic system can choose as efficient an algorithm for processing a complex query as a trained programmer would. System R is an experimental system constructed at the San Jose IBM Research Laboratory to demonstrate that a relational database system can incorporate the high performance and complete function required for everyday production use.

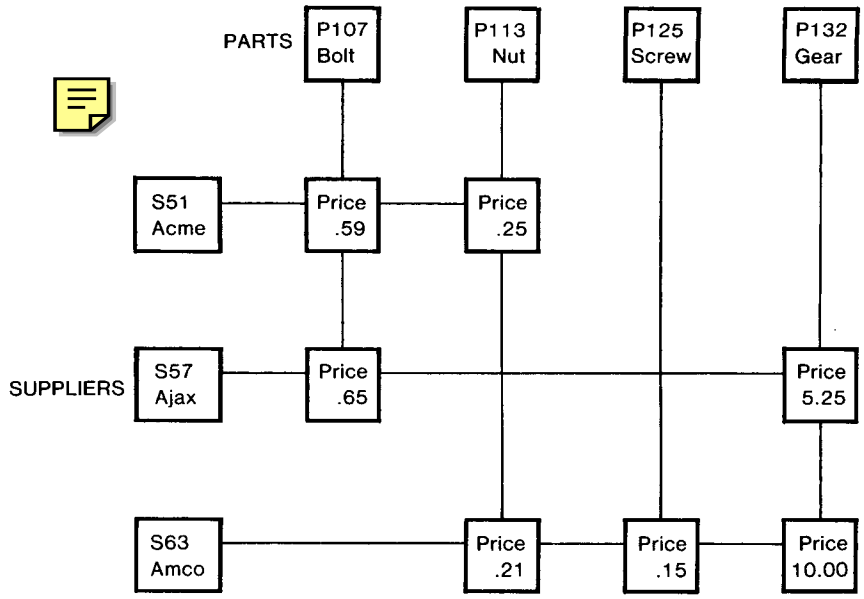


Fig. 1(a). A "Navigational" Database.

The key goals established for System R were:

(1) To provide a high-level, nonnavigational user interface for maximum user productivity and data independence.

(2) To support different types of database use including programmed transactions, ad hoc queries, and report generation.

(3) To support a rapidly changing database environment, in which tables, indexes, views, transactions, and other objects could easily be added to and removed from the database without stopping the system.

(4) To support a population of many concurrent users, with mechanisms to protect the integrity of the database in a concurrent-update environment.

(5) To provide a means of recovering the contents of the database to a consistent state after a failure of hardware or software.

(6) To provide a flexible mechanism whereby different views of stored data can be defined and various users can be authorized to query and update these views.

(7) To support all of the above functions with a level of performance comparable to existing lower-function database systems.

Throughout the System R project, there has been a strong commitment to carry the system through to an operationally complete prototype

PARTS

PARTNO	NAME
P107	Bolt
P113	Nut
P125	Screw
P132	Gear

SUPPLIERS

SUPPNO	NAME
S51	Acme
S57	Ajax
S63	Amco

PRICES

PARTNO	SUPPNO	PRICE
P107	S51	.59
P107	S57	.65
P113	S51	.25
P113	S63	.21
P125	S63	.15
P132	S57	5.25
P132	S63	10.00

Fig. 1(b). A Relational Database.

which could be installed and evaluated in actual user sites.

The history of System R can be divided into three phases. "Phase Zero" of the project, which occurred during 1974 and most of 1975, involved the development of the SQL user interface [14] and a quick implementation of a subset of SQL for one user at a time. The Phase Zero prototype, described in [2], provided valuable insight in several areas, but its code was eventually abandoned. "Phase One" of the project, which took place throughout most of 1976 and 1977, involved the design and construction of the full-function, multiuser version of System R. An initial system architecture was presented in [4] and subsequent updates to the design were described in [10]. "Phase Two" was the evaluation of System R in actual use. This occurred during 1978 and 1979 and involved experiments at the San Jose Research Laboratory and several other user sites. The results of some of these experiments and user experiences are described in [19–21]. At each user site, System R was installed for experimental purposes only, and not as a supported commercial product.[1]

This paper will describe the decisions which were made and the lessons learned during each of the three phases of the System R project.

2. Phase Zero: An Initial Prototype

Phase Zero of the System R project involved the quick implementation of a subset of system functions. From the beginning, it was our intention to learn what we could from this initial prototype, and then scrap the Phase Zero code before construction of the more complete version of System R. We decided to use the relational access method called XRM, which had been developed by R. Lorie at IBM's Cambridge Scientific Center [40]. (XRM was influenced, to some extent, by the "Gamma Zero" interface defined by E.F. Codd and others at San Jose [11].) Since XRM is a single-user access method without locking or recovery capabilities, issues relating to concurrency and recovery were excluded from consideration in Phase Zero.

An interpreter program was written in PL/I to execute statements in the high-level SQL (formerly SEQUEL) language [14, 16] on top of XRM. The implemented subset of the SQL language included queries and updates of the database, as well as the dynamic creation of new database relations. The Phase Zero implementation supported the "subquery" construct of SQL, but not its "join" construct. In effect, this meant that a query could search through several relations in computing its result, but the final result would be taken from a single relation.

The Phase Zero implementation was primarily intended for use as a standalone query interface by end users at interactive terminals. At the time, little emphasis was placed on issues of interfacing to host-language programs (although Phase Zero could be called from a PL/I program). However, considerable thought was given to the human factors aspects of the SQL language, and an experimental study was conducted on the learnability and usability of SQL [44].

One of the basic design decisions in the Phase Zero prototype was that the system catalog, i.e., the description of the content and structure of the database, should be stored as a set of regular relations in the database itself. This approach permits the system to keep the catalog up to date automatically as changes are made to the database, and also makes the catalog information available to the system optimizer for use in access path selection.

The structure of the Phase Zero interpreter was strongly influenced by the facilities of XRM. XRM stores relations in the form of "tuples," each of which has a unique 32-bit "tuple identifier" (TID). Since a TID contains a page number, it is possible, given a TID, to fetch the associated tuple in one page reference. However, rather than actual data values, the tuple contains pointers to the "domains" where the actual data is stored, as shown in Figure 2. Optionally, each domain may have an "inversion," which associates domain values (e.g., "Programmer") with the TIDs of tuples in which the values appear. Using the inversions, XRM makes it easy to find a list of TIDs of tuples which contain a given value. For example, in Figure 2, if inversions exist on both the JOB and LOCATION domains, XRM provides commands to create a list of TIDs of employees who are programmers, and another list of TIDs of employees who work in Evanston. If the SQL query calls for programmers who work in Evanston, these TID lists can be intersected to obtain the list of TIDs of tuples which satisfy the query, before any tuples are actually fetched.

The most challenging task in constructing the Phase Zero prototype was the design of optimizer algorithms for efficient execution of SQL statements on top of XRM. The design of the Phase Zero optimizer is given in [2]. The objective of the optimizer was to minimize the number of tuples fetched from the database in processing a query. Therefore, the optimizer made extensive use of inversions and often manipulated TID lists before beginning to fetch tuples. Since the TID lists were potentially large, they were stored as temporary objects in the database during query processing.

The results of the Phase Zero implementation were mixed. One strongly felt conclusion was that it is a very good idea, in a project the size of System R, to plan to throw away the initial implementation. On the positive side, Phase Zero demonstrated the usability of the SQL language, the feasibility of creating new tables and inversions "on the fly"

[1] The System R research prototype later evolved into SQL/Data System, a relational database management product offered by IBM in the DOS/VSE operating system environment.

and relying on an automatic optimizer for access path selection, and the convenience of storing the system catalog in the database itself. At the same time, Phase Zero taught us a number of valuable lessons which greatly influenced the design of our later implementation. Some of these lessons are summarized below.

(1) The optimizer should take into account not just the cost of fetching tuples, but the costs of creating and manipulating TID lists, then fetching tuples, then fetching the data pointed to by the tuples. When these "hidden costs" are taken into account, it will be seen that the manipulation of TID lists is quite expensive, especially if the TID lists are managed in the database rather than in main storage.

(2) Rather than "number of tuples fetched," a better measure of cost would have been "number of I/Os." This improved cost measure would have revealed the great importance of clustering together related tuples on physical pages so that several related tuples could be fetched by a single I/O. Also, an I/O measure would have revealed a serious drawback of XRM: Storing the domains separately from the tuples causes many extra I/Os to be done in retrieving data values. Because of this, our later implementation stored data values in the actual tuples rather than in separate domains. (In defense of XRM, it should be noted that the separation of data values from tuples has some advantages if data values are relatively large and if many tuples are processed internally compared to the number of tuples which are materialized for output.)

(3) Because the Phase Zero implementation was observed to be CPU-bound during the processing of a typical query, it was decided the optimizer cost measure should be a weighted sum of CPU time and I/O count, with weights adjustable according to the system configuration.

(4) Observation of some of the applications of Phase Zero convinced us of the importance of the "join" formulation of SQL. In our subsequent implementation, both "joins" and "subqueries" were supported.

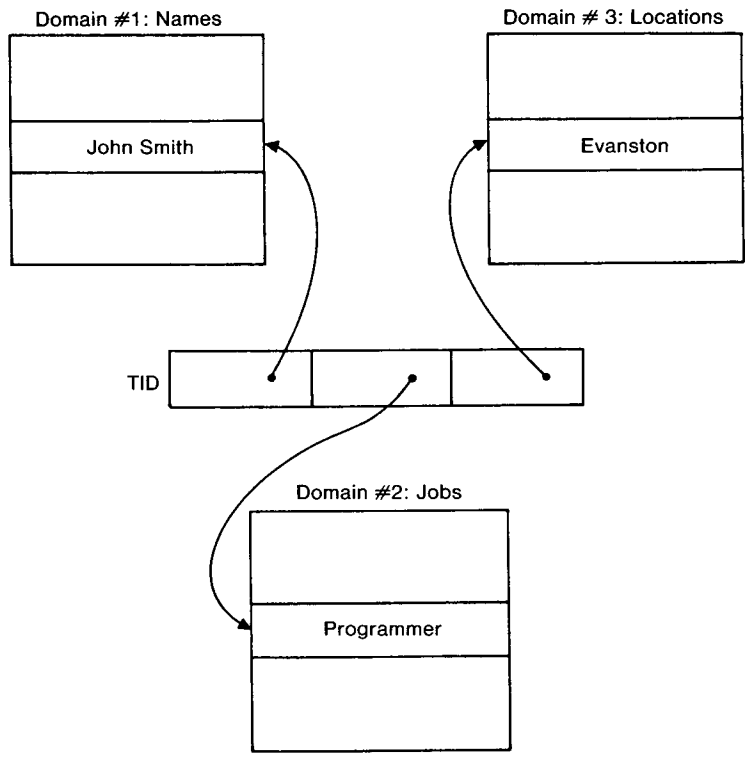


Fig. 2. XRM Storage Structure.

(5) The Phase Zero optimizer was quite complex and was oriented toward complex queries. In our later implementation, greater emphasis was placed on relatively simple interactions, and care was taken to minimize the "path length" for simple SQL statements.

3. Phase One: Construction of a Multiuser Prototype

After the completion and evaluation of the Phase Zero prototype, work began on the construction of the full-function, multiuser version of System R. Like Phase Zero, System R consisted of an access method (called RSS, the Research Storage System) and an optimizing SQL processor (called RDS, the Relational Data System) which runs on top of the RSS. Separation of the RSS and RDS provided a beneficial degree of modularity; e.g., all locking and logging functions were isolated in the RSS, while all authorization and access path selection functions were isolated in the RDS. Construction of the RSS was underway in 1975 and construction of the RDS began in 1976. Unlike XRM, the RSS was originally designed to support multiple concurrent users.

The multiuser prototype of System R contained several important subsystems which were not present in the earlier Phase Zero prototype. In order to prevent conflicts which might arise when two concurrent users attempt to update the same data value, a locking subsystem was provided. The locking subsystem ensures that each data value is accessed by only one user at a time, that all the updates made by a given transaction become effective simultaneously, and that deadlocks between users are detected and resolved. The security of the system was enhanced by view and authorization subsystems. The view subsystem permits users to define alternative views of the database (e.g., a view of the employee file in which salaries are deleted or aggregated by department).

authorization subsystem ensures that each user has access only to those views for which he has been specifically authorized by their creators. Finally, a recovery subsystem was provided which allows the database to be restored to a consistent state in the event of a hardware or software failure.

In order to provide a useful host-language capability, it was decided that System R should support both PL/I and Cobol application programs as well as a standalone query interface, and that the system should run under either the VM/CMS or MVS/TSO operating system environment. A key goal of the SQL language was to present the same capabilities, and a consistent syntax, to users of the PL/I and Cobol host languages and to ad hoc query users. The imbedding of SQL into PL/I is described in [16]. Installation of a multiuser database system under VM/CMS required certain modifications to the operating system in support of communicating virtual machines and writable shared virtual memory. These modifications are described in [32].

The standalone query interface of System R (called UFI, the User-Friendly Interface) is supported by a dialog manager program, written in PL/I, which runs on top of System R like any other application program. Therefore, the UFI support program is a cleanly separated component and can be modified independently of the rest of the system. In fact, several users improved on our UFI by writing interactive dialog managers of their own.

The Compilation Approach

Perhaps the most important decision in the design of the RDS was inspired by R. Lorie's observation, in early 1976, that it is possible to compile very high-level SQL statements into compact, efficient routines in System/370 machine language [42]. Lorie was able to demonstrate that SQL statements of arbitrary complexity could be decomposed into a relatively small collection of machine-language "fragments," and that an optimizing compiler could assemble these code fragments from a library to form a specially tailored routine for processing a given SQL statement. This technique had a very dramatic effect on our ability to support application programs for transaction processing. In System R, a PL/I or Cobol program is run through a preprocessor in which its SQL statements are examined, optimized, and compiled into small, efficient machine-language routines which are packaged into an "access module" for the application program. Then, when the program goes into execution, the access module is invoked to perform all interactions with the database by means of calls to the RSS. The process of creating and invoking an access module is illustrated in Figures 3 and 4. All the overhead of parsing, validity checking, and access path selection is removed from the path of the executing program and placed in a separate preprocessor step which need not be repeated. Perhaps even more important is the fact that the running program interacts only with its small, special-purpose access module rather than with a much larger and less efficient general-purpose SQL interpreter. Thus, the power and ease of use of the high-level SQL language are combined with the execution-time efficiency of the much lower level RSS interface.

Since all access path selection decisions are made during the preprocessor step in System R, there is the possibility that subsequent changes in the database may invalidate the decisions which are embodied in an access module. For example, an index selected by the optimizer may later be dropped from the database. Therefore, System R records with each access module a list of its "dependencies" on database objects such as tables and indexes. The dependency list is stored in the form of a regular relation in the system catalog. When the structure of the database changes (e.g., an index is dropped), all affected access modules are marked "invalid." The next time an invalid access module is invoked, it is regenerated from its original SQL statements, with newly optimized access paths. This process is completely transparent to the System R user.

SQL statements submitted to the interactive UFI dialog manager are processed by the same optimizing compiler as preprocessed SQL statements. The UFI program passes the ad hoc SQL statement to System R with a special "EXECUTE" call. In response to the EXECUTE call, System R parses and optimizes the SQL statement and translates it into a machine-language routine. The routine is indistinguishable from an access module and is executed immediately. This process is described in more detail in [20].

RSS Access Paths

Rather than storing data values in separate "domains" in the manner of XRM, the RSS chose to store data values in the individual rcords of the database. This resulted in records becoming variable in length and longer, on the average, than the equivalent XRM records. Also, commonly used values are represented many times rather than only once as in XRM. It was felt, however, that these disadvantages were more than offset by the following advantage: All the data values of a record could be fetched by a single I/O.

In place of XRM "inversions," the RSS provides "indexes," which are associative access aids implemented in the form of B-Trees [26]. Each table in the database may have anywhere from zero indexes up to an index on each column (it is also possible to create an index on a combination of columns). Indexes make it possible to scan the table in order by the indexed values, or to directly access the records which match a particular value. Indexes are maintained automatically by the RSS in the event of updates to the database.

The RSS also implements "links," which are pointers stored

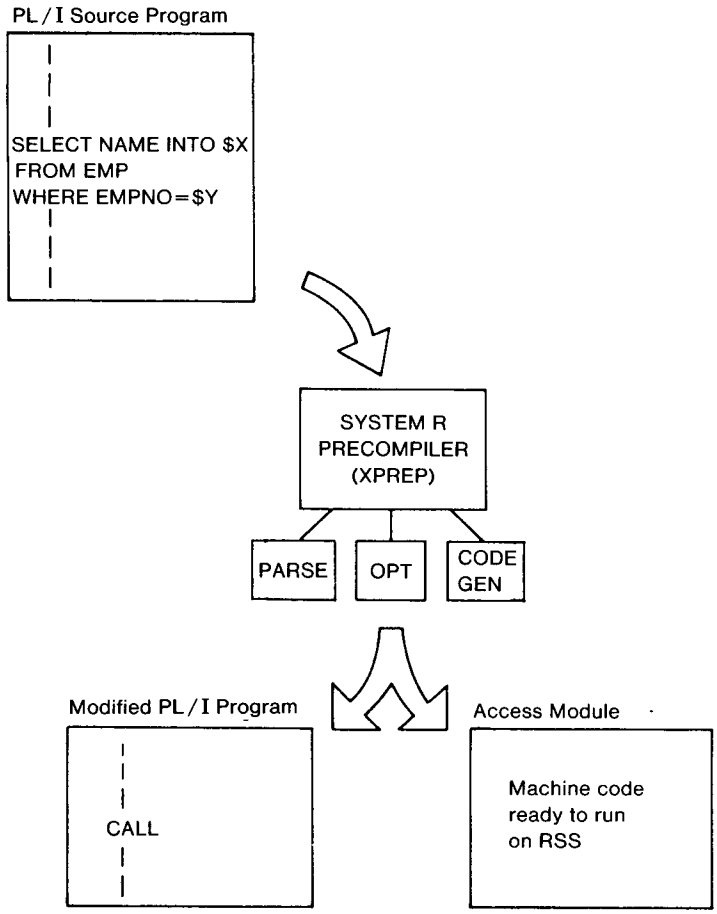


Fig. 3. Precompilation Step.

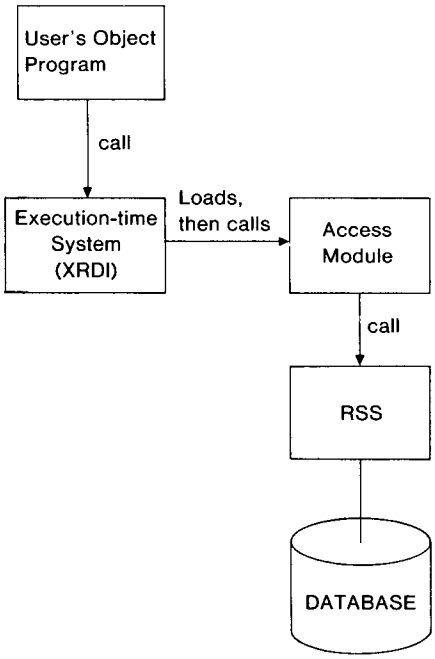


Fig. 4. Execution Step.

with [illegible]ord which connect it to other [illegible]ed records. The connection of records on links is not performed automatically by the RSS, but must be done by a higher level system.

The access paths made available by the RSS include (1) index scans, which access a table associatively and scan it in value order using an index; (2) relation scans, which scan over a table as it is laid out in physical storage; (3) link scans, which traverse from one record to another using links. On any of these types of scan, "search arguments" may be specifie[illegible]ch limit the records returned [illegible]ose satisfying a certain predicate. Also, the RSS provides a built-in sorting mechanism which can take records from any of the scan methods and sort them into some value order, storing the result in a temporary list in the database. In System R, the RDS makes extensive use of index and relation scans and sorting. The RDS also utilizes links for internal purposes but not as an access path to user data.

The Optimizer

Building on our Phase Zero experience, we designed the System R optimizer to minimize the weighted sum of the predicted number of I/Os and RSS calls in processing an SQL statement (the relative weights of these two terms are adjustable according to system configuration). Rather than manipulating TID lists, the optimizer chooses to scan each table in the SQL query by means of only one index (or, if no suitable index exists, by means of a relation scan). For example, if the query calls for programmers who work in Evanston, the optimizer might choose to use the job index to find programmers and then examine their locations; it might use the location index to find Evanston employees and examine their jobs; or it might simply scan the relation and examine the job and location of all employees. The choice would be based on the optimizer's estimate of both the clustering and selectivity properties of each index, based on statistics stored in the system catalog. An index is considered highly selective if it has a large ratio of distinct key values to total entries. An index is considered to have the clustering property if the key order of the index corresponds closely to the ordering of records in physical storage. The clustering property is important because when a record is fetched via a clustering index, it is likely that other records with the same key will be found on the same page, thus minimizing the number of page fetches. Because of the importance of clustering, mechanisms were provided for loading data in value order and preserving the value ordering when new records are inserted into the database.

The techniques of the System R optimizer for performing joins of two or more tables have their origin in a study conducted by M. Blasgen and

K. Eswaran [7]. Using APL models, Blasgen and Eswaran studied ten methods of joining together tables, based on the use of indexes, sorting, physical pointers, and TID lists. The number of disk accesses required to perform a join was predicted on the basis of various assumptions for the ten join methods. Two join methods were identified such that one or the other was optimal or nearly optimal under most circumstances. The two methods are as follows:

Join Method 1: Scan over the qualifying rows of table A. For each row, fetch the matching rows of table B (usually, but not always, an index on table B is used).

Join Method 2: (Often used when no suitable index exists.) Sort the qualifying rows of tables A and B in order by their respective join fields. Then scan over the sorted lists and merge them by matching values.

When selecting an access path for a join of several tables, the System R optimizer considers the problem to be a sequence of binary joins. It then performs a tree search in which each level of the tree consists of one of the binary joins. The choices to be made at each level of the tree include which join method to use and which index, if any, to select for scanning. Comparisons are applied at each level of the tree to prune away paths which achieve the same results as other, less costly paths. When all paths have been examined, the optimizer selects the one of minimum predicted cost. The System R optimizer algorithms are described more fully in [47].

Views and Authorization

The major objectives of the view and authorization subsystems of System R were power and flexibility. We wanted to allow any SQL query to be used as the definition of a view. This was accomplished by storing each view definition in the form of an SQL parse tree. When an SQL operation is to be executed against a view, the parse tree which defines the operation is merged with the parse tree which defines the view, producing a composite parse tree which is then sent to the optimizer for access path selection. This approach is similar to the "query modification" technique proposed by Stonebraker [48]. The algorithms developed for merging parse trees were sufficiently general so that nearly any SQL statement could be executed against any view definition, with the restriction that a view can be updated only if it is derived from a single table in the database. The reason for this restriction is that some updates to views which are derived from more than one table are not meaningful (an example of such an update is given in [24]).

The authorization subsystem of System R is based on privileges which are controlled by the SQL statements GRANT and REVOKE. Each user of System R may optionally be given a privilege called RESOURCE which enables him/her to create new tables in the database. When a user creates a table, he/she receives all privileges to access, update, and destroy that table. The creator of a table can then grant these privileges to other individual users, and subsequently can revoke these grants if desired. Each granted privilege may optionally carry with it the "GRANT option," which enables a recipient to grant the privilege to yet other users. A REVOKE destroys the whole chain of granted privileges derived from the original grant. The authorization subsystem is described in detail in [37] and discussed further in [31].

The Recovery Subsystem

The key objective of the recovery subsystem is provision of a means whereby the database may be recovered to a consistent state in the event of a failure. A consistent state is defined as one in which the database does not reflect any updates made by transactions which did not complete successfully. There are three basic types of failure: the disk media may fail, the system may fail, an individual transaction may fail. Though both the scope of the failure and the time to effect recovery may be different, all three types of recovery require that an alternate copy of data be available when the primary copy is not.

When a media failure occurs, database information on disk is lost. When this happens, an image dump of the database plus a log of "before" and "after" changes provide the alternate copy which makes recovery possible. System R's use of "dual logs" even permits recovery from media failures on the log itself. To recover from a media failure, the database is restored using the latest image dump and the recovery process reapplies all database changes as specified on the log for completed transactions.

When a system failure occurs, the information in main memory is lost. Thus, enough information must always be on disk to make recovery possible. For recovery from system failures, System R uses the change log mentioned above plus something called "shadow pages." As each page in the database is updated, the page is written out in a new place on disk, and the original page is retained. A directory of the "old" and "new" locations of each page is maintained. Periodically during normal operation, a "checkpoint" occurs in which all updates are forced out to disk, the "old" pages are discarded, and the "new" pages become "old." In the event of a system crash, the "new" pages on disk may be in an inconsistent state because some updated pages may still be in the system buffers and not yet reflected on disk. To bring the database back to a consistent state, the system reverts to the "old" pages, and then uses the log to redo all committed transactions and to undo all updates made by incomplete transactions. This aspect of the System R recovery subsystem is described in more detail in [36].

When a transaction failure occurs, all database changes which have been made by the failing transaction must be undone. To accom-

plish this, System R simply processes the change log backwards removing all changes made by the transaction. Unlike media and system recovery which both require that System R be reinitialized, transaction recovery takes place on-line.

The Locking Subsystem

A great deal of thought was given to the design of a locking subsystem which would prevent interference among concurrent users of System R. The original design involved the concept of "predicate locks," in which the lockable unit was a database property such as "employees whose location is Evanston." Note that, in this scheme, a lock might be held on the predicate LOC = 'EVANSTON', even if no employees currently satisfy that predicate. By comparing the predicates being processed by different users, the locking subsystem could prevent interference. The "predicate lock" design was ultimately abandoned because: (1) determining whether two predicates are mutually satisfiable is difficult and time-consuming; (2) two predicates may appear to conflict when, in fact, the semantics of the data prevent any conflict, as in "PRODUCT = AIRCRAFT" and "MANUFACTURER = ACME STATIONERY CO."; and (3) we desired to contain the locking subsystem entirely within the RSS, and therefore to make it independent of any understanding of the predicates being processed by various users. The original predicate locking scheme is described in [29].

The locking scheme eventually chosen for System R is described in [34]. This scheme involves a hierarchy of locks, with several different sizes of lockable units, ranging from individual records to several tables. The locking subsystem is transparent to end users, but acquires locks on physical objects in the database as they are processed by each user. When a user accumulates many small locks, they may be "traded" for a larger lockable unit (e.g., locks on many records in a table might be traded for a lock on the table). When locks are acquired on small objects, "intention" locks are simultaneously acquired on the larger objects which contain them. For example, user A and user B may both be updating employee records. Each user holds an "intention" lock on the employee table, and "exclusive" locks on the particular records being updated. If user A attempts to trade her individual record locks for an "exclusive" lock at the table level, she must wait until user B ends his transaction and releases his "intention" lock on the table.

4. Phase Two: Evaluation

The evaluation phase of the System R project lasted approximately 2½ years and consisted of two parts: (1) experiments performed on the system at the San Jose Research Laboratory, and (2) actual use of the system at a number of internal IBM sites and at three selected customer sites. At all user sites, System R was installed on an experimental basis for study purposes only, and not as a supported commercial product. The first installations of System R took place in June 1977.

General User Comments

In general, user response to System R has been enthusiastic. The system was mostly used in applications for which ease of installation, a high-level user language, and an ability to rapidly reconfigure the database were important requirements. Several user sites reported that they were able to install the system, design and load a database, and put into use some application programs within a matter of days. User sites also reported that it was possible to tune the system performance after data was loaded by creating and dropping indexes without impacting end users or application programs. Even changes in the database tables could be made transparent to users if the tables were read-only, and also in some cases for updated tables.

Users found the performance characteristics and resource consumption of System R to be generally satisfactory for their experimental applications, although no specific performance comparisons were drawn. In general, the experimental databases used with System R were smaller than one 3330 disk pack (200 Megabytes) and were typically accessed by fewer than ten concurrent users. As might be expected, interactive response slowed down during the execution of very complex SQL statements involving joins of several tables. This performance degradation must be traded off against the advantages of normalization [23, 30], in which large database tables are broken into smaller parts to avoid redundancy, and then joined back together by the view mechanism or user applications.

The SQL Language

The SQL user interface of System R was generally felt to be successful in achieving its goals of simplicity, power, and data independence. The language was simple enough in its basic structure so that users without prior experience were able to learn a usable subset on their first sitting. At the same time, when taken as a whole, the language provided the query power of the first-order predicate calculus combined with operators for grouping, arithmetic, and built-in functions such as SUM and AVERAGE.

Users consistently praised the uniformity of the SQL syntax across the environments of application programs, ad hoc query, and data definition (i.e., definition of views). Users who were formerly required to learn inconsistent languages for these purposes found it easier to deal with the single syntax (e.g., when debugging an application program by querying the database to observe its effects). The single syntax also enhanced communication among different functional organizations (e.g., between database administrators and application programmers).

While developing applications using SQL, our experimental users made a number of suggestions for extensions and improvements to the language, most of which were implemented during the course of the proj-

ect. Some of these suggestions are summarized below:

(1) Users requested an easy-to-use syntax when testing for the existence or nonexistence of a data item, such as an employee record whose department number matches a given department record. This facility was implemented in the form of a special "EXISTS" predicate.

(2) Users requested a means of searching for character strings whose contents are only partially known, such as "all license plates beginning with NVK." This facility was implemented in the form of a special "LIKE" predicate which searches for "patterns" that are allowed to contain "don't care" characters.

(3) A requirement arose for an application program to compute an SQL statement dynamically, submit the statement to the System R optimizer for access path selection, and then execute the statement repeatedly for different data values without reinvoking the optimizer. This facility was implemented in the form of PREPARE and EXECUTE statements which were made available in the host-language version of SQL.

(4) In some user applications the need arose for an operator which Codd has called an "outer join" [25]. Suppose that two tables (e.g., SUPPLIERS and PROJECTS) are related by a common data field (e.g., PARTNO). In a conventional join of these tables, supplier records which have no matching project record (and vice versa) would not appear. In an "outer join" of these tables, supplier records with no matching project record would appear together with a "synthetic" project record containing only null values (and similarly for projects with no matching supplier). An "outer-join" facility for SQL is currently under study.

A more complete discussion of user experience with SQL and the resulting language improvements is presented in [19].

The Compilation Approach

The approach of compiling SQL statements into machine code was one of the most successful parts of the System R project. We were able to generate a machine-language routine to execute any SQL statement of arbitrary complexity by selecting code fragments from a library of approximately 100 fragments. The result was a beneficial effect on transaction programs, ad hoc query, and system simplicity.

In an environment of short, repetitive transactions, the benefits of compilation are obvious. All the overhead of parsing, validity checking, and access path selection are removed from the path of the running transaction, and the application program interacts with a small, specially tailored access module rather than with a larger and less efficient general-purpose interpreter program. Experiments [38] showed that for a typical short transaction, about 80 percent of the instructions were executed by the RSS, with the remaining 20 percent executed by the access module and application pro-

Example 1:

```
SELECT  SUPPNO, PRICE
FROM    QUOTES
WHERE   PARTNO = '010002'
AND MINQ<=1000 AND MAXQ>=1000;
```

Operation	CPU time (msec on 168)	Number of I/Os
Parsing	13.3	0
Access Path Selection	40.0	9
Code Generation	10.1	0
Fetch answer set (per record)	1.5	0.7

Example 2:

```
SELECT  ORDERNO,ORDERS.PARTNO,DESCRIP,DATE,QTY
FROM    ORDERS,PARTS
WHERE   ORDERS.PARTNO = PARTS.PARTNO
AND     DATE BETWEEN '750000' AND '751231'
AND     SUPPNO = '797';
```

Operation	CPU time (msec on 168)	Number of I/Os
Parsing	20.7	0
Access Path Selection	73.2	9
Code Generation	19.3	0
Fetch answer set (per record)	8.7	10.7

Fig. 5. Measurements of Cost of Compilation.

gram. Thus, the user pays only a small cost for the power, flexibility, and data independence of the SQL language, compared with writing the same transaction directly on the lower level RSS interface.

In an ad hoc query environment the advantages of compilation are less obvious since the compilation must take place on-line and the query is executed only once. In this environment, the cost of generating a machine-language routine for a given query must be balanced against the increased efficiency of this routine as compared with a more conventional query interpreter. Figure 5 shows some measurements of the cost of compiling two typical SQL statements (details of the experiments are given in [20]). From this data we may draw the following conclusions:

(1) The code generation step adds a small amount of CPU time and no I/Os to the overhead of parsing and access path selection. Parsing and access path selection must be done in any query system, including interpretive ones. The additional instructions spent on code generation are not likely to be perceptible to an end user.

(2) If code generation results in a routine which runs more efficiently than an interpreter, the cost of the code generation step is paid back after fetching only a few records. (In Example 1, if the CPU time per record of the compiled module is half that of an interpretive system, the cost of generating the access module is repaid after seven records have been fetched.)

A final advantage of compilation is its simplifying effect on the system architecture. With both ad hoc queries and precanned transactions being treated in the same way, most of the code in the system can be made to serve a dual purpose. This ties in very well with our objective of supporting a uniform syntax between query users and transaction programs.

Available Access Paths

As described earlier, the principal access path used in System R for retrieving data associatively by its value is the B-tree index. A typical index is illustrated in Figure 6. If we assume a fan-out of approximately 200 at each level of the tree, we can index up to 40,000 records by a two-level index, and up to 8,000,000 records by a three-level index. If we wish to begin an associative scan through a large table, three I/Os will typically be required (assuming the root page is referenced frequently enough to remain in the system buffers, we need an I/O for the intermediate-level index page, the "leaf" index page, and the data page). If several records are to be fetched using the index scan, the three start-up I/Os are relatively insignificant. However, if only one record is to be fetched, other access techniques might have provided a quicker path to the stored data.

Two common access techniques which were not utilized for user data in System R are hashing and direct links (physical pointers from one record to another). Hashing was not used because it does not have the convenient ordering property of a B-tree index (e.g., a B-tree index on SALARY enables a list of employees ordered by SALARY to be retrieved very easily). Direct links, although they were implemented at the RSS level, were not used as an access path for user data by the RDS for a two-fold reason. *Essential links* (links whose semantics are not known to the system but which are connected directly by users) were rejected because they were inconsistent with the nonnavigational user interface of a relational system, since they could not be used as access paths by an automatic optimizer. *Nonessential links* (links which connect records to other records with matching data values) were not implemented because of the difficulties in automatically maintaining their connections. When a record is updated, its connections on many links may need to be updated as well, and this may involve many "subsidiary queries" to find the other records which are involved in these connections. Problems also arise relating to records which have no matching partner record on the link, and records whose link-controlling data value is null.

In general, our experience showed that indexes could be used very efficiently in queries and transactions which access many records,

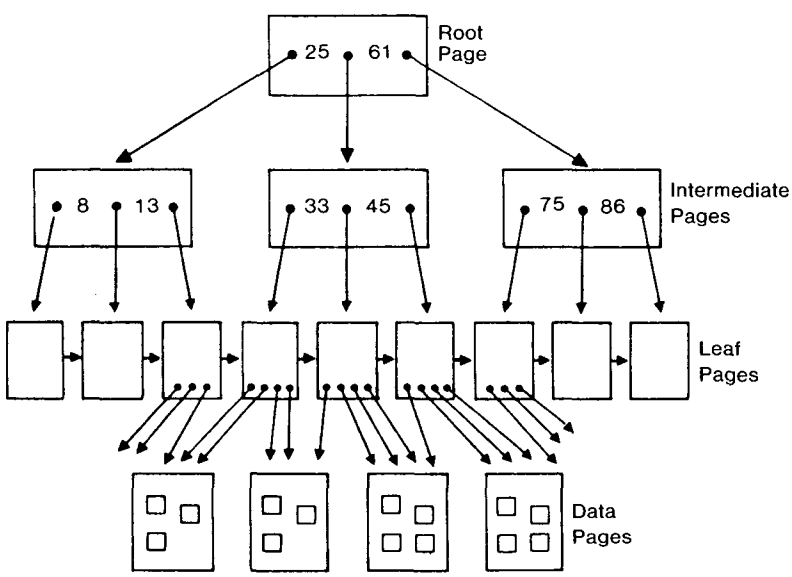


Fig. 6. A B-Tree Index.

but that hashing and links would have enhanced the performance of "canned transactions" which access only a few records. As an illustration of this problem, consider an inventory application which has two tables: a PRODUCTS table, and a much larger PARTS table which contains data on the individual parts used for each product. Suppose a given transaction needs to find the price of the heating element in a particular toaster. To execute this transaction, System R might require two I/Os to traverse a two-level index to find the toaster record, and three more I/Os to traverse another three-level index to find the heating element record. If access paths based on hashing and direct links were available, it might be possible to find the toaster record in one I/O via hashing, and the heating element record in one more I/O via a link. (Additional I/Os would be required in the event of hash collisions or if the toaster parts records occupied more than one page.) Thus, for this very simple transaction hashing and links might reduce the number of I/Os from five to three, or even two. For transactions which retrieve a large set of records, the additional I/Os caused by indexes compared to hashing and links are less important.

The Optimizer

A series of experiments was conducted at the San Jose IBM Research Laboratory to evaluate the success of the System R optimizer in choosing among the available access paths for typical SQL statements. The results of these experiments are reported in [6]. For the purpose of the experiments, the optimizer was modified in order to observe its behavior. Ordinarily, the optimizer searches through a tree of path choices, computing estimated costs and pruning the tree until it arrives at a single preferred access path. The optimizer was modified in such a way that it could be made to generate the complete tree of access paths, without pruning, and to estimate the cost of each path (cost is defined as a weighted sum of page fetches and RSS calls). Mechanisms were also added to the system whereby it could be forced to execute an SQL statement by a particular access path and to measure the actual number of page fetches and RSS calls incurred. In this way, a comparison can be made between the optimizer's predicted cost and the actual measured cost for various alternative paths.

In [6], an experiment is described in which ten SQL statements, including some single-table queries and some joins, are run against a test database. The database is artificially generated to conform to the two basic assumptions of the System R optimizer: (1) the values in each column are uniformly distributed from some minimum to some maximum value; and (2) the distribution of values of the various columns are independent of each other. For each of the ten SQL statements, the ordering of the predicted costs of the various access paths was the same as the ordering of the actual measured costs (in a few cases the optimizer predicted two paths to have the same cost when their actual costs were unequal but adjacent in the ordering).

Although the optimizer was able to correctly order the access paths in the experiment we have just described, the magnitudes of the predicted costs differed from the measured costs in several cases. These discrepancies were due to a variety of causes, such as the optimizer's inability to predict how much data would remain in the system buffers during sorting.

The above experiment does not address the issue of whether or not a very good access path for a given SQL statement might be overlooked because it is not part of the optimizer's repertoire. One such example is known. Suppose that the database contains a table T in which each row has a unique value for the field SEQNO, and suppose that an index exists on SEQNO. Consider the following SQL query:

```
SELECT * FROM T WHERE SEQNO IN
(15, 17, 19, 21);
```

This query has an answer set of (at most) four rows, and an obvious method of processing it is to use the SEQNO index repeatedly: first to find the row with SEQNO = 15, then SEQNO = 17, etc. However, this access path would not be chosen by System R, because the optimizer is not presently structured to consider multiple uses of an index within a single query block. As we gain more experience with access path selection, the optimizer may grow to encompass this and other access paths which have so far been omitted from consideration.

Views and Authorization

Users generally found the System R mechanisms for defining views and controlling authorization to be powerful, flexible, and convenient. The following features were considered to be particularly beneficial:

(1) The full query power of SQL is made available for defining new views of data (i.e., any query may be defined as a view). This makes it possible to define a rich variety of views, containing joins, subqueries, aggregation, etc., without having to learn a separate "data definition language." However, the view mechanism is not completely transparent to the end user, because of the restrictions described earlier (e.g., views involving joins of more than one table are not updateable).

(2) The authorization subsystem allows each installation of System R to choose a "fully centralized policy" in which all tables are created and privileges controlled by a central administrator; or a "fully decentralized policy" in which each user may create tables and control access to them; or some intermediate policy.

During the two-year evaluation of System R, the following suggestions were made by users for improvement of the view and authorization subsystems:

(1) The authorization subsystem could be augmented by the concept of a "group" of users. Each group would have a "group administrator" who controls enrollment of new members in the group. Privileges could then be granted to the group as a whole rather than to each member of the group individually.

(2) A new command could be added to the SQL language to change the ownership of a table from one user to another. This suggestion is more difficult to implement than it seems at first glance, because the owner's name is part of the fully qualified name of a table (i.e., two tables owned by Smith and Jones could be named SMITH.PARTS and JONES.PARTS). References to the table SMITH.PARTS might exist in many places, such as view definitions and compiled programs. Finding and changing all these references would be difficult (perhaps impossible, as in the case of users' source programs which are not stored under System R control).

(3) Occasionally it is necessary to reload an existing table in the database (e.g., to change its physical clustering properties). In System R this is accomplished by dropping the old table definition, creating a new table with the same definition, and reloading the data into the new table. Unfortunately, views and authorizations defined on the table are lost from the system when the old definition is dropped, and therefore they both must be redefined on the new table. It has been suggested that views and authorizations defined on a dropped table might optionally be held "in abeyance" pending reactivation of the table.

The Recovery Subsystem

The combined "shadow page" and log mechanism used in System R proved to be quite successful in safeguarding the database against media, system, and transaction failures. The part of the recovery subsystem which was observed to have the greatest impact on system performance was the keeping of a shadow page for each updated page. This performance impact is due primarily to the following factors:

(1) Since each updated page is written out to a new location on disk, data tends to move about. This limits the ability of the system to cluster related pages in secondary storage to minimize disk arm movement for sequential applications.

(2) Since each page can potentially have both an "old" and "new" version, a directory must be maintained to locate both versions of each page. For large databases, the directory may be large enough to require a paging mechanism of its own.

(3) The periodic checkpoints which exchange the "old" and "new" page pointers generate I/O activity and consume a certain amount of CPU time.

A possible alternative technique for recovering from system failures would dispense with the concept of shadow pages, and simply keep a log of all database updates. This design would require that all updates be written out to the log before the updated page migrates to disk from the system buffers. Mechanisms could be developed to minimize I/Os by retaining updated pages in the buffers until several pages are written out at once, sharing an I/O to the log.

The Locking Subsystem

The locking subsystem of System R provides each user with a choice of three levels of isolation from other users. In order to explain the three levels, we define "uncommitted data" as those records which have been updated by a transaction that is still in progress (and therefore still subject to being backed out). Under no circumstances can a transaction, at any isolation level, perform updates on the uncommitted data of another transaction, since this might lead to lost updates in the event of transaction backout.

The three levels of isolation in System R are defined as follows:

Level 1: A transaction running at Level 1 may read (but not update) uncommitted data. Therefore, successive reads of the same record by a Level-1 transaction may not give consistent values. A Level-1 transaction does not attempt to acquire any locks on records while reading.

Level 2: A transaction running at Level 2 is protected against reading uncommitted data. However, successive reads at Level 2 may still yield inconsistent values if a second transaction updates a given record and then terminates between the first and second reads by the Level-2 transaction. A Level-2 transaction locks each record before reading it to make sure it is committed at the time of the read, but then releases the lock immediately after reading.

Level 3: A transaction running at Level 3 is guaranteed that successive reads of the same record will yield the same value. This guarantee is enforced by acquiring a lock on each record read by a Level-3 transaction and holding the lock until the end of the transaction. (The lock acquired by a Level-3 reader is a "share" lock which permits other users to read but not update the locked record.)

It was our intention that Isolation Level 1 provide a means for very quick scans through the database when approximate values were acceptable, since Level-1 readers acquire no locks and should never need to wait for other users. In practice, however, it was found that Level-1 readers did have to wait under certain circumstances while the physical consistency of the data was suspended (e.g., while indexes or pointers were being adjusted). Therefore, the potential of Level 1 for increasing system concurrency was not fully realized.

It was our expectation that a tradeoff would exist between Isolation Levels 2 and 3 in which Level 2 would be "cheaper" and Level 3 "safer." In practice, however, it was observed that Level 3 actually involved less CPU overhead than Level 2, since it was simpler to acquire locks and keep them than to acquire locks and immediately release them. It is true that Isolation Level 2 permits a greater degree of

access to the database by concurrent readers and updaters than does Level 3. However, this increase in concurrency was not observed to have an important effect in most practical applications.

As a result of the observations described above, most System R users ran their queries and application programs at Level 3, which was the system default.

The Convoy Phenomenon

Experiments with the locking subsystem of System R identified a problem which came to be known as the "convoy phenomenon" [9]. There are certain high-traffic locks in System R which every process requests frequently and holds for a short time. Examples of these are the locks which control access to the buffer pool and the system log. In a "convoy" condition, interaction between a high-traffic lock and the operating system dispatcher tends to serialize all processes in the system, allowing each process to acquire the lock only once each time it is dispatched.

In the VM/370 operating system, each process in the multiprogramming set receives a series of small "quanta" of CPU time. Each quantum terminates after a preset amount of CPU time, or when the process goes into page, I/O, or lock wait. At the end of the series of quanta, the process drops out of the multiprogramming set and must undergo a longer "time slice wait" before it once again becomes dispatchable. Most quanta end when a process waits for a page, an I/O operation, or a low-traffic lock. The System R design ensures that no process will ever hold a high-traffic lock during any of these types of wait. There is a slight probability, however, that a process might go into a long "time slice wait" while it is holding a high-traffic lock. In this event, all other dispatchable processes will soon request the same lock and become enqueued behind the sleeping process. This phenomenon is called a "convoy."

In the original System R design, convoys are stable because of the protocol for releasing locks. When a process *P* releases a lock, the locking subsystem grants the lock to the first waiting process in the queue (thereby making it unavailable to be reacquired by *P*). After a short time, *P* once again requests the lock, and is forced to go to the end of the convoy. If the mean time between requests for the high-traffic lock is 1,000 instructions, each process may execute only 1,000 instructions before it drops to the end of the convoy. Since more than 1,000 instructions are typically used to dispatch a process, the system goes into a "thrashing" condition in which most of the cycles are spent on dispatching overhead.

The solution to the convoy problem involved a change to the lock release protocol of System R. After the change, when a process *P* releases a lock, all processes which are enqueued for the lock are made dispatchable, but the lock is not granted to any particular process. Therefore, the lock may be regranted to process *P* if it makes a subsequent request. Process *P* may acquire and release the lock many times before its time slice is exhausted. It is highly probable that process *P* will not be holding the lock when it goes into a long wait. Therefore, if a convoy should ever form, it will most likely evaporate as soon as all the members of the convoy have been dispatched.

Additional Observations

Other observations were made during the evaluation of System R and are listed below:

(1) When running in a "canned transaction" environment, it would be helpful for the system to include a data communications front end to handle terminal interactions, priority scheduling, and logging and restart at the message level. This facility was not included in the System R design. Also, space would be saved and the working set reduced if several users executing the same "canned transaction" could share a common access module. This would require the System R code generator to produce reentrant code. Approximately half the space occupied by the multiple copies of the access module could be saved by this method, since the other half consists of working storage which must be duplicated for each user.

(2) When the recovery subsystem attempts to take an automatic checkpoint, it inhibits the processing of new RSS commands until all users have completed their current RSS command; then the checkpoint is taken and all users are allowed to proceed. However, certain RSS commands potentially involve long operations, such as sorting a file. If these "long" RSS operations were made interruptible, it would avoid any delay in performing checkpoints.

(3) The System R design of automatically maintaining a system catalog as part of the on-line database was very well liked by users, since it permitted them to access the information in the catalog with exactly the same query language they use for accessing other data.

5. Conclusions

We feel that our experience with System R has clearly demonstrated the feasibility of applying a relational database system to a real production environment in which many concurrent users are performing a mixture of ad hoc queries and repetitive transactions. We believe that the high-level user interface made possible by the relational data model can have a dramatic positive effect on user productivity in developing new applications, and on the data independence of queries and programs. System R has also demonstrated the ability to support a highly dynamic database environment in which application requirements are rapidly changing.

In particular, System R has illustrated the feasibility of compiling a very high-level data sublanguage, SQL, into machine-level code. The

result of this compilation technique is that most of the overhead cost for implementing the high-level language is pushed into a "precompilation" step, and performance for canned transactions is comparable to that of a much lower level system. The compilation approach has also proved to be applicable to the ad hoc query environment, with the result that a unified mechanism can be used to support both queries and transactions.

The evaluation of System R has led to a number of suggested improvements. Some of these improvements have already been implemented and others are still under study. Two major foci of our continuing research program at the San Jose laboratory are adaptation of System R to a distributed database environment, and extension of our optimizer algorithms to encompass a broader set of access paths.

Sometimes questions are asked about how the performance of a relational database system might compare to that of a "navigational" system in which a programmer carefully hand-codes an application to take advantage of explicit access paths. Our experiments with the System R optimizer and compiler suggest that the relational system will probably approach but not quite equal the performance of the navigational system for a particular, highly tuned application, but that the relational system is more likely to be able to adapt to a broad spectrum of unanticipated applications with adequate performance. We believe that the benefits of relational systems in the areas of user productivity, data independence, and adaptability to changing circumstances will take on increasing importance in the years ahead.

Acknowledgments

From the beginning, System R was a group effort. Credit for any success of the project properly belongs to the team as a whole rather than to specific individuals.

The inspiration for constructing a relational system came primarily from E. F. Codd, whose landmark paper [22] introduced the relational model of data. The manager of the project through most of its existence was W. F. King.

In addition to the authors of this paper, the following people were associated with System R and made important contributions to its development:

M. Adiba	M. Mresse
R.F. Boyce	J.F. Nilsson
A. Chan	R.L. Obermarck
D.M. Choy	D. Stott Parker
K. Eswaran	D. Portal
R. Fagin	N. Ramsperger
P. Fehder	P. Reisner
T. Haerder	P.R. Roever
R.H. Katz	R. Selinger
W. Kim	H.R. Strong
H. Korth	P. Tiberio
P. McJones	V. Watson
D. McLeod	R. Williams

References

1. Adiba, M.E., and Lindsay, B.G. Database snapshots. IBM Res. Rep. RJ2772, San Jose, Calif., March 1980.

2. Astrahan, M.M., and Chamberlin, D.D. Implementation of a structured English query language. *Comm. ACM 18*, 10 (Oct. 1975), 580–588.

3. Astrahan, M.M., and Lorie, R.A. SEQUEL-XRM: A Relational System. Proc. ACM Pacific Regional Conf., San Francisco, Calif., April 1975, p. 34.

4. Astrahan, M.M., et al. System R: A relational approach to database management. *ACM Trans. Database Syst.1*, 2 (June 1976) 97–137.

5. Astrahan, M.M., et al. System R: A relational data base management system. *IEEE Comptr. 12*, 5 (May 1979), 43–48.

6. Astrahan, M.M., Kim, W., and Schkolnick, M. Evaluation of the System R access path selection mechanism. Proc. IFIP Congress, Melbourne, Australia, Sept. 1980, pp. 487–491.

7. Blasgen, M.W., Eswaran, K.P. Storage and access in relational databases. *IBM Syst. J. 16*, 4 (1977), 363–377.

8. Blasgen, M.W., Casey, R.G., and Eswaran, K.P. An encoding method for multifield sorting and indexing. *Comm. ACM 20*, 11 (Nov. 1977), 874–878.

9. Blasgen, M., Gray, J., Mitoma, M., and Price, T. The convoy phenomenon. *Operating Syst. Rev. 13*, 2 (April 1979), 20–25.

10. Blasgen, M.W., et al. System R: An architectural overview. *IBM Syst. J. 20*, 1 (Feb. 1981), 41–62.

11. Bjorner, D., Codd, E.F., Deckert, K.L., and Traiger, I.L. The Gamma Zero N-ary relational data base interface. IBM Res. Rep. RJ1200, San Jose, Calif., April 1973.

12. Boyce, R.F., and Chamberlin, D.D. Using a structured English query language as a data definition facility. IBM Res. Rep. RJ1318, San Jose, Calif., Dec. 1973.

13. Boyce, R.F., Chamberlin, D.D., King, W.F., and Hammer, M.M. Specifying queries as relational expressions: The SQUARE data sublanguage. *Comm. ACM 18*, 11 (Nov. 1975), 621–628.

14. Chamberlin, D.D., and Boyce, R.F. SEQUEL: A structured English query language. Proc. ACM-SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor, Mich., May 1974, pp. 249–264.

15. Chamberlin, D.D., Gray, J.N., and Traiger, I.L. Views, authorization, and locking in a relational database system. Proc. 1975 Nat. Comptr. Conf., Anaheim, Calif., pp. 425–430.

16. Chamberlin, D.D., et al. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. and Develop. 20*, 6 (Nov. 1976), 560–575 (also see errata in Jan. 1977 issue).

17. Chamberlin, D.D. Relational database management systems. *Comptng. Surv. 8*, 1 (March 1976), 43–66.

18. Chamberlin, D.D., et al. Data base system authorization. In *Foundations of Secure Computation*, R. Demillo, D. Dobkin, A. Jones, and R. Lipton, Eds., Academic Press, New York, 1978, pp. 39–56.

19. Chamberlin, D.D. A summary of user experience with the SQL data sublanguage. Proc. Internat. Conf. Data Bases, Aberdeen, Scotland, July 1980, pp. 181–203 (also IBM Res. Rep. RJ2767, San Jose, Calif., April 1980).

20. Chamberlin, D.D., et al. Support for repetitive transactions and ad-hoc queries in System R. *ACM Trans. Database Syst. 6*, 1 (March 1981), 70–94.

21. Chamberlin, D.D., Gilbert, A.M., and Yost, R.A. A history of System R and SQL/data system (presented at the Internat. Conf. Very Large Data Bases, Cannes, France, Sept. 1981).

22. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.

23. Codd, E.F. Further normalization of the data base relational model. In *Courant Computer Science Symposia, Vol. 6: Data Base Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 33–64.

24. Codd, E.F. Recent investigations in relational data base systems. Proc. IFIP Congress, Stockholm, Sweden, Aug. 1974.

25. Codd, E.F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (Dec. 1979), 397–434.

26. Comer, D. The ubiquitous B-Tree. *Comptng. Surv. 11*, 2 (June 1979), 121–137.

27. Date, C.J. *An Introduction to Database Systems.* 2nd Ed., Addison-Wesley, New York, 1977.

28. Eswaran, K.P., and Chamberlin, D.D. Functional specifications of a subsystem for database integrity. Proc. Conf. Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 48–68.

29. Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. On the notions of consistency and predicate locks in a database system. *Comm. ACM 19*, 11 (Nov. 1976), 624–633.

30. Fagin, R. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 262–278.

31. Fagin, R. On an authorization mechanism. *ACM Trans. Database Syst. 3*, 3 (Sept. 1978), 310–319.

32. Gray, J.N., and Watson, V. A shared segment and inter-process communication facility for VM/370. IBM Res. Rep. RJ1579, San Jose, Calif., Feb. 1975.

33. Gray, J.N., Lorie, R.A., and Putzolu, G.F. Granularity of locks in a large shared database. Proc. Conf. Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 428–451.

34. Gray, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I.L. Granularity of locks and degrees of consistency in a shared data base. Proc. IFIP Working Conf. Modelling of Database Management Systems, Freudenstadt, Germany, Jan. 1976, pp. 695–723 (also IBM Res. Rep. RJ1654, San Jose, Calif.).

35. Gray, J.N. Notes on database operating systems. In *Operating Systems: An Advanced Course*, Goos and Hartmanis, Eds., Springer-Verlag, New York, 1978, pp. 393–481 (also IBM Res. Rep. RJ2188, San Jose, Calif.).

36. Gray, J.N., et al. The recovery manager of a data management system. IBM Res. Rep. RJ2623, San Jose, Calif., June 1979.

37. Griffiths, P.P., and Wade, B.W. An authorization mechanism for a relational database system. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 242–255.

38. Katz, R.H., and Selinger, R.D. Internal comm., IBM Res. Lab., San Jose, Calif., Sept. 1978.

39. Kwan, S.C., and Strong, H.R. Index path length evaluation for the research storage system of System R. IBM Res. Rep. RJ2736, San Jose, Calif., Jan. 1980.

40. Lorie, R.A. XRM—An extended (N-ary) relational memory. IBM Tech. Rep. G320-2096, Cambridge Scientific Ctr., Cambridge, Mass., Jan. 1974.

41. Lorie, R.A. Physical integrity in a large segmented database. *ACM Trans. Database Syst. 2*, 1 (March 1977), 91–104.

42. Lorie, R.A., and Wade, B.W. The compilation of a high level data language. IBM Res. Rep. RJ2598, San Jose, Calif., Aug. 1979.

43. Lorie, R.A., and Nilsson, J.F. An access specification language for a relational data base system. *IBM J. Res. and Develop. 23*, 3 (May 1979), 286–298.

44. Reisner, P., Boyce, R.F., and Chamberlin, D.D. Human factors evaluation of two data base query languages: SQUARE and SEQUEL. Proc. AFIPS Nat. Comptr. Conf., Anaheim, Calif., May 1975, pp. 447–452.

45. Reisner, P. Use of psychological experimentation as an aid to development of a query language. *IEEE Trans. Software Eng. SE-3*, 3 (May 1977), 218–229.

46. Schkolnick, M., and Tiberio, P. Considerations in developing a design tool for a relational DBMS. Proc. IEEE COMPSAC 79, Nov. 1979, pp. 228–235.

47. Selinger, P.G., et al. Access path selection in a relational database management system. Proc. ACM SIGMOD Conf., Boston, Mass., June 1979, pp. 23–34.

48. Stonebraker, M. Implementation of integrity constraints and views by query modification. Tech. Memo ERL-M514, College of Eng., Univ. of Calif. at Berkeley, March 1975.

49. Strong, H.R., Traiger, I.L., and Markowsky, G. Slide Search. IBM Res. Rep. RJ2274, San Jose, Calif., June 1978.

50. Traiger, I.L., Gray J.N., Galtieri, C.A., and Lindsay, B.G. Transactions and consistency in distributed database systems. IBM Res. Rep. RJ2555, San Jose, Calif., June 1979.

A Fast File System for UNIX*

Marshall Kirk McKusick, William N. Joy†,
Samuel J. Leffler‡, Robert S. Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

A reimplementation of the UNIX file system is described. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies that allow better locality of reference and can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access to large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long needed enhancements to the programmers' interface are discussed. These include a mechanism to place advisory locks on files, extensions of the name space across file systems, the ability to use long file names, and provisions for administrative control of resource usage.

Revised February 18, 1984

CR Categories and Subject Descriptors: D.4.3 **[Operating Systems]**: File Systems Management – *file organization, directory structures, access methods*; D.4.2 **[Operating Systems]**: Storage Management – *allocation/deallocation strategies, secondary storage devices*; D.4.8 **[Operating Systems]**: Performance – *measurements, operational analysis*; H.3.2 **[Information Systems]**: Information Storage – *file organization*

Additional Keywords and Phrases: UNIX, file system organization, file system performance, file system design, application program interface.

General Terms: file system, measurement, performance.

---

* UNIX is a trademark of Bell Laboratories.

† William N. Joy is currently employed by: Sun Microsystems, Inc, 2550 Garcia Avenue, Mountain View, CA 94043

‡ Samuel J. Leffler is currently employed by: Lucasfilm Ltd., PO Box 2009, San Rafael, CA 94912

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under ARPA Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

TABLE OF CONTENTS

1. Introduction

2. Old file system

3. New file system organization
3.1. Optimizing storage utilization
3.2. File system parameterization
3.3. Layout policies

4. Performance

5. File system functional enhancements
5.1. Long file names
5.2. File locking
5.3. Symbolic links
5.4. Rename
5.5. Quotas

Acknowledgements

References

1. Introduction

This paper describes the changes from the original 512 byte UNIX file system to the new one released with the 4.2 Berkeley Software Distribution. It presents the motivations for the changes, the methods used to effect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the facilities that are available to programmers.

The original UNIX system that runs on the PDP-11† has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. Virtually no constraints other than available disk space are placed on file growth [Ritchie74], [Thompson78].*

When used on the VAX-11 together with other UNIX enhancements, the original 512 byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications such as VLSI design and image processing do a small amount of processing on a large quantities of data and need to have a high throughput from the file system. High throughput rates are also needed by programs that map files from the file system into large virtual address spaces. Paging data in and out of the file system is likely to occur frequently [Ferrin82b]. This requires a file system providing higher bandwidth than the original 512 byte UNIX one that provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput. Consequently, users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. Previous work to improve the UNIX file system performance has been done by [Ferrin82a]. The UNIX operating system drew many of its ideas from Multics, a large, high performance

† DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

* In practice, a file's size is constrained to be less than about one gigabyte.

operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a LISP environment [Symbolics81]. A good introduction to the physical latencies of disks is described in [Pechura83].

2. Old File System

In the file system developed at Bell Laboratories (the "traditional" file system), each disk drive is divided into one or more partitions. Each of these disk partitions may contain one file system. A file system never spans multiple partitions.† A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to the *free list*, a linked list of all the free blocks in the file system.

Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. An inode contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in an inode itself*. An inode may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further singly indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A 150 megabyte traditional UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from the file's inode to its data. Files in a single directory are not typically allocated consecutive slots in the 4 megabytes of inodes, causing many non-consecutive blocks of inodes to be accessed when executing operations on the inodes of several files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by staging modifications to critical file system information so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors: each disk transfer accessed twice as much data, and most files could be described without need to access indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system.*

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually the free list became entirely random, causing files to have their blocks allocated randomly over the disk. This forced a seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second when they were first created, this rate deteriorated to 30 kilobytes per second after a few weeks of moderate use because of this randomization of data block placement. There was no way of restoring the performance of an old file system except to dump, rebuild, and restore the file system. Another possibility, as suggested by [Maruyama76], would be to have a process that periodically

† By "partition" here we refer to the subdivision of physical space on a disk drive. In the traditional file system, as in the new file system, file systems are really located in logical disk partitions that may overlap. This overlapping is made available, for example, to allow programs to copy entire disk drives containing multiple file systems.

* The actual number may vary from system to system, but is usually in the range 5-13.

reorganized the data on the disk to restore locality.

3. New file system organization

In the new file system organization (as in the old file system organization), each disk drive contains one or more file systems. A file system is described by its super-block, located at the beginning of the file system's disk partition. Because the super-block contains critical data, it is replicated to protect against catastrophic loss. This is done when the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

To insure that it is possible to create files as large as 2^{32} bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of a file system is recorded in the file system's super-block so it is possible for file systems with different block sizes to be simultaneously accessible on the same system. The block size must be decided at the time that the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization divides a disk partition into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. The bit map of available blocks in the cylinder group replaces the traditional file system's free list. For each cylinder group a static number of inodes is allocated at file system creation time. The default policy is to allocate one inode for each 2048 bytes of space in the cylinder group, expecting this to be far more than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all redundant copies of the super-block. Thus the cylinder group bookkeeping information begins at a varying offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group than the preceding cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-block. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks.†

3.1. Optimizing storage utilization

Data is laid out so that larger blocks can be transferred in a single disk transaction, greatly increasing file system throughput. As an example, consider a file in the new file system composed of 4096 byte data blocks. In the old file system this file would be composed of 1024 byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096 byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before requiring a seek.

The main problem with larger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Table 1 shows the effect of file system block size on the amount of wasted space in the file system. The files measured to obtain these figures reside on one of our

† While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with blocks sizes of 16 kilobytes or greater. This is because of a requirement that the first 8 kilobytes of the disk be reserved for a bootstrap program and a separate requirement that the cylinder group information begin on a file system block boundary. To start the cylinder group on a file system block boundary, file systems with block sizes larger than 8 kilobytes would have to leave an empty space between the end of the boot block and the beginning of the cylinder group. Without knowing the size of the file system blocks, the system would not know what roundup function to use to find the beginning of the first cylinder group.

time sharing systems that has roughly 1.2 gigabytes of on-line storage. The measurements are based on the active user file systems containing about 920 megabytes of formatted space.

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9	Data + inodes, 512 byte block UNIX file system
866.5 Mb	11.8	Data + inodes, 1024 byte block UNIX file system
948.5 Mb	22.4	Data + inodes, 2048 byte block UNIX file system
1128.3 Mb	45.6	Data + inodes, 4096 byte block UNIX file system

Table 1 – Amount of wasted space as a function of block size.

The space wasted is calculated to be the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks.

To be able to use large blocks without undue waste, small files must be stored in a more efficient way. The new file system accomplishes this goal by allowing the division of a single file system block into one or more *fragments*. The file system fragment size is specified at the time that the file system is created; each file system block can optionally be broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space available in a cylinder group at the fragment level; to determine if a block is available, aligned fragments are examined. Figure 1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1 – Example layout of blocks and fragments in a 4096/1024 file system.

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0–5, 10, and 11 are in use, while fragments 6–9, and 12–15 are free. Fragments of adjoining blocks cannot be used as a full block, even if they are large enough. In this example, fragments 6–9 cannot be allocated as a full block; only fragments 12–15 can be coalesced into a full block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remaining fragments of the block are made available for allocation to other files. As an example consider an 11000 byte file stored on a 4096/1024 byte file system. This file would uses two full size blocks and one three fragment portion of another block. If no block with three aligned fragments is available at the time the file is created, a full size block is split yielding the necessary fragments and a single unused fragment. This remaining fragment can be allocated to another file as needed.

Space is allocated to a file when a program does a *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased*. If the file needs to be expanded to hold the new data, one of three conditions exists:

1) There is enough space left in an already allocated block or fragment to hold the new data. The new data is written into the available space.

2) The file contains no fragmented blocks (and the last block in the file contains insufficient space to hold the new data). If space exists in a block already allocated, the space is filled with new data. If the remainder of the new data contains more than a full block of data, a full block is allocated and the

* A program may be overwriting data in the middle of an existing file in which case space would already have been allocated.

first full block of new data is written there. This process is repeated until less than a full block of new data remains. If the remaining new data to be written will fit in less than a full block, a block with the necessary fragments is located, otherwise a full block is located. The remaining new data is written into the located space.

3) The file contains one or more fragments (and the fragments contain insufficient space to hold the new data). If the size of the new data plus the size of the data already in the fragments exceeds the size of a full block, a new block is allocated. The contents of the fragments are copied to the beginning of the block and the remainder of the block is filled with new data. The process then continues as in (2) above. Otherwise, if the new data to be written will fit in less than a full block, a block with the necessary fragments is located, otherwise a full block is located. The contents of the existing fragments appended with the new data are written into the allocated space.

The problem with expanding a file one fragment at a a time is that data may be copied many times as a fragmented block expands to a full block. Fragment reallocation can be minimized if the user program writes a full block at a time, except for a partial block at the end of the file. Since file systems with different block sizes may reside on the same system, the file system interface has been extended to provide application programs the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The amount of wasted space in the 4096/1024 byte new file system organization is empirically observed to be about the same as in the 1024 byte old file system organization. A file system with 4096 byte blocks and 512 byte fragments has about the same amount of wasted space as the 512 byte block UNIX file system. The new file system uses less space than the 512 byte or 1024 byte file systems for indexing information for large files and the same amount of space for small files. These savings are offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilization when a new file system's fragment size equals an old file system's block size.

In order for the layout policies to be effective, a file system cannot be kept completely full. For each file system there is a parameter, termed the free space reserve, that gives the minimum acceptable percentage of file system blocks that should be free. If the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter may be changed at any time, even when the file system is mounted and active. The transfer rates that appear in section 4 were measured on file systems kept less than 90% full (a reserve of 10%). If the number of free blocks falls to zero, the file system throughput tends to be cut in half, because of the inability of the file system to localize blocks in a file. If a file system's performance degrades because of overfilling, it may be restored by removing files until the amount of free space once again reaches the minimum acceptable level. Access rates for files created during periods of little free space may be restored by moving their data once enough space is available. The free space reserve must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, the percentage of waste in an old 1024 byte UNIX file system is roughly comparable to a new 4096/512 byte file system with the free space reserve set at 5%. (Compare 11.8% wasted with the old file system to 6.9% waste + 5% reserved space in the new file system.)

3.2. File system parameterization

Except for the initial creation of the free list, the old file system ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass storage device, or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration-dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given installation can have several different disk technologies running on a single processor. Each file system is parameterized so that it can be adapted to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be rotationally well

positioned. The distance between "rotationally optimal" blocks varies greatly; it can be a consecutive block or a rotationally delayed block depending on system characteristics. On a processor with an input/output channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks can often be accessed without suffering lost time because of an intervening disk revolution. For processors without input/output channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to service an interrupt and schedule a new disk transfer. Given a block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in the file will come into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the available blocks in a cylinder group at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive. The super-block contains a vector of lists called *rotational layout tables*. The vector is indexed by rotational position. Each component of the vector lists the index into the block map for every data block contained in its rotational position. When looking for an allocatable block, the system first looks through the summary counts for a rotational position with a non-zero block count. It then uses the index of the rotational position to find the appropriate list to use to index through only the relevant parts of the block map to find a free block.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with a rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

3.3. Layout policies

The file system layout policies are divided into two distinct parts. At the top level are global policies that use file system wide summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because there are insufficient blocks left in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80], and to improve the layout of data to make larger transfers possible as described by [Nevalainen77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space forcing the data to be scattered to non-local cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes are used to describe both files and directories. Inodes of files in the same directory are frequently accessed together. For example, the "list directory" command

often accesses the inode for each file in a directory. The layout policy tries to place all the inodes of files in a directory in the same cylinder group. To ensure that files are distributed throughout the disk, a different policy is used for directory allocation. A new directory is placed in a cylinder group that has a greater than average number of free inodes, and the smallest number of directories already in it. The intent of this policy is to allow the inode clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for a particular cylinder group can be read with 8 to 16 disk transfers. (At most 16 disk transfers are required because a cylinder group may have no more than 2048 inodes.) This puts a small and constant upper bound on the number of disk transfers required to access the inodes for all the files in a directory. In contrast, the old file system typically requires one disk transfer to fetch the inode for each file in a directory.

The other major resource is data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all data blocks for a file in the same cylinder group, preferably at rotationally optimal positions in the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Further, using all the space in a cylinder group causes future allocations for any file in the cylinder group to also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The heuristic solution chosen is to redirect block allocation to a different cylinder group when a file exceeds 48 kilobytes, and at every megabyte thereafter.* The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free, otherwise it allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristics that employ only partial information.

If a requested block is not available, the local allocator uses a four level allocation strategy:

1) Use the next available block rotationally closest to the requested block on the same cylinder. It is assumed here that head switching time is zero. On disk controllers where this is not the case, it may be possible to incorporate the time required to switch between disk platters when constructing the rotational layout tables. This, however, has not yet been tried.

2) If there are no blocks available on the same cylinder, use a block within the same cylinder group.

3) If that cylinder group is entirely full, quadratically hash the cylinder group number to choose another cylinder group to look for a free block.

4) Finally if the hash fails, apply an exhaustive search to all cylinder groups.

Quadratic hash is used because of its speed in finding unused slots in nearly full hash tables [Knuth75]. File systems that are parameterized to maintain at least 10% free space rarely use this strategy. File systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random; the most important characteristic of the strategy used under such conditions is that the strategy be fast.

* The first spill over point at 48 kilobytes is the point at which a file on a 4096 byte block file system first requires a single indirect block. This appears to be a natural first point at which to redirect block allocation. The other spillover points are chosen with the intent of forcing block allocation to be redirected when a file has used about 25% of the data blocks in a cylinder group. In observing the new file system in day to day use, the heuristics appear to work well in minimizing the number of completely filled cylinder groups.

4. Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long term performance of the new file system.

Our empirical studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories (to force the system to access inodes in multiple cylinder groups), the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate at which user programs can transfer data to or from a file without performing any processing on it. These programs must read and write enough data to insure that buffering in the operating system does not affect the results. They are also run at least three times in succession; the first to get the system into a known state and the second two to insure that the experiment has stabilized and is repeatable. The tests used and their results are discussed in detail in [Kridle83]†. The systems were running multi-user but were otherwise quiescent. There was no contention for either the CPU or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an AMPEX Capricorn 330 megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured. The same number of system calls were performed in all tests; the basic system call overhead was a negligible portion of the total running time of the tests.

Type of File System	Processor and Bus Measured	Read Speed	Read Bandwidth	Read % CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/983 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/983 22%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/983 24%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/983 47%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/983 47%	54%

Table 2a – Reading rates of the old and new UNIX file systems.

Type of File System	Processor and Bus Measured	Write Speed	Write Bandwidth	Write % CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/983 5%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/983 14%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/983 22%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/983 33%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/983 47%	95%

Table 2b – Writing rates of the old and new UNIX file systems.

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in Table 2 were based on a file system with a 10% free space reserve. Synthetic work loads suggest that throughput deteriorates to about half the rates given in Table 2 when the file systems are full.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is calculated by multiplying the number of bytes on a track by the number of revolutions of the disk per second. The bandwidth is calculated by comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3–5% of the disk bandwidth, while the new file system uses up to 47% of the bandwidth.

† A UNIX command that is similar to the reading test that we used is "cp file /dev/null", where "file" is eight megabytes long.

Both reads and writes are faster in the new system than in the old system. The biggest factor in this speedup is because of the larger block size used by the new file system. The overhead of allocating blocks in the new system is greater than the overhead of allocating blocks in the old system, however fewer blocks need to be allocated in the new system because they are bigger. The net effect is that the cost per byte allocated is about the same for both systems.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must do more work when allocating blocks than when simply reading them. Note that the write rates are about the same as the read rates in the 8192 byte block file system; the write rates are slower than the read rates in the 4096 byte block file system. The slower write rates occur because the kernel has to do twice as many disk allocations per second, making the processor unable to keep up with the disk transfer rate.

In contrast the old file system is about 50% faster at writing files than reading them. This is because the write system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced, hence disk transfers queue up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek distance, the average seek between the scheduled disk writes is much less than it would be if the data blocks were written out in the random disk order in which they are generated. However when the file is read, the read system call is processed synchronously so the disk blocks must be retrieved from the disk in the non-optimal seek order in which they are requested. This forces the disk scheduler to do long seeks resulting in a lower throughput rate.

In the new system the blocks of a file are more optimally ordered on the disk. Even though reads are still synchronous, the requests are presented to the disk in a much better order. Even though the writes are still asynchronous, they are already presented to the disk in minimum seek order so there is no gain to be had by reordering them. Hence the disk seek latencies that limited the old file system have little effect in the new file system. The cost of allocation is the factor in the new system that causes writes to be slower than reads.

The performance of the new file system is currently limited by memory to memory copy operations required to move data from disk buffers in the system's address space to data buffers in the user's address space. These copy operations account for about 40% of the time spent performing an input/output operation. If the buffers in both address spaces were properly aligned, this transfer could be performed without copying by using the VAX virtual memory management hardware. This would be especially desirable when transferring large amounts of data. We did not implement this because it would change the user interface to the file system in two major ways: user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow contiguous disk blocks to be read in a single disk transaction. Many disks used with UNIX systems contain either 32 or 48 512 byte sectors per track. Each track holds exactly two or three 8192 byte file system blocks, or four or six 4096 byte file system blocks. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than 50% of the available bandwidth. If the next block for a file cannot be laid out contiguously, then the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, then it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. The reason that block chaining has not been implemented is because it would require rewriting all the disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up allocations, the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79]. This technique was not included because block allocation currently accounts for less than 10% of the time spent in a write system call and, once again, the current throughput rates are already limited by the speed of the available processors.

5. File system functional enhancements

The performance enhancements to the UNIX file system did not require any changes to the semantics or data structures visible to application programs. However, several changes had been generally desired for some time but had not been introduced because they would require users to dump and restore all their file systems. Since the new file system already required all existing file systems to be dumped and restored, these functional enhancements were introduced at this time.

5.1. Long file names

File names can now be of nearly arbitrary length. Only programs that read directories are affected by this change. To promote portability to UNIX systems that are not running the new file system, a set of directory access routines have been introduced to provide a consistent interface to directories on both old and new systems.

Directories are allocated in 512 byte units called chunks. This size is chosen so that each allocation can be transferred to disk in a single operation. Chunks are broken up into variable length records termed directory entries. A directory entry contains the information necessary to map the name of a file to its associated inode. No directory entry is allowed to span multiple chunks. The first three fields of a directory entry are fixed length and contain: an inode number, the size of the entry, and the length of the file name contained in the entry. The remainder of an entry is variable length and contains a null terminated file name, padded to a 4 byte boundary. The maximum length of a file name in a directory is currently 255 characters.

Available space in a directory is recorded by having one or more entries accumulate the free space in their entry size fields. This results in directory entries that are larger than required to hold the entry name plus fixed length fields. Space allocated to a directory should always be completely accounted for by totaling up the sizes of its entries. When an entry is deleted from a directory, its space is returned to a previous entry in the same directory chunk by increasing the size of the previous entry by the size of the deleted entry. If the first entry of a directory chunk is free, then the entry's inode number is set to zero to indicate that it is unallocated.

5.2. File locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to use a separate ''lock'' file. A process would try to create a ''lock'' file. If the creation succeeded, then the process could proceed with its update; if the creation failed, then the process would wait and try again. This mechanism had three drawbacks. Processes consumed CPU time by looping over attempts to create locks. Locks left lying around because of system crashes had to be manually removed (normally in a system startup command script). Finally, processes running as system administrator are always permitted to create files, so were forced to use a different mechanism. While it is possible to get around all these problems, the solutions are not straight forward, so a mechanism for locking files has been added.

The most general schemes allow multiple processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to serialize access to a file with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the standard system applications, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the extent of enforcement. A hard lock is always enforced when a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks there must be some override policy implemented in the kernel. With advisory locks the policy is left to the user programs. In the UNIX system, programs with system administrator privilege are allowed override any protection scheme. Because many of the programs that need to use locks must also run as the system administrator, we chose to implement advisory locks rather than create an additional protection scheme that was inconsistent with the UNIX philosophy or could not be used by system

administration programs.

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process may have an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the lock request will block until the lock can be obtained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process may access the file.

Locks are applied or removed only on open files. This means that locks can be manipulated without needing to close and reopen a file. This is useful, for example, when a process wishes to apply a shared lock, read some information and determine whether an update is required, then apply an exclusive lock and update the file.

A request for a lock will cause a process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to conditionally request a lock is useful to "daemon" processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since locks exist only while the locking processes exist, lock files can never be left active after the processes exit or if the system crashes.

Almost no deadlock detection is attempted. The only deadlock detection done by the system is that the file to which a lock is applied must not already have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail).

5.3. Symbolic links

The traditional UNIX file system allows multiple directory entries in the same file system to reference a single file. Each directory entry "links" a file's name to an inode and its contents. The link concept is fundamental; inodes do not reside in directories, but exist separately and are referenced by links. When all the links to an inode are removed, the inode is deallocated. This style of referencing an inode does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations *symbolic links* similar to the scheme used by Multics [Feiertag71] have been added.

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. In UNIX, pathnames are specified relative to the root of the file system hierarchy, or relative to a process's current working directory. Pathnames specified relative to the root are called absolute pathnames. Pathnames specified relative to the current working directory are termed relative pathnames. If a symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname that they are using. However certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links; seven system utilities required changes to use these calls.

In future Berkeley software distributions it may be possible to reference file systems located on remote machines using pathnames. When this occurs, it will be possible to create symbolic links that span machines.

5.4. Rename

Programs that create a new version of an existing file typically create the new version as a temporary file and then rename the temporary file with the name of the target file. In the old UNIX file system renaming required three calls to the system. If a program were interrupted or the system crashed between these calls, the target file could be left with only its temporary name. To eliminate this possibility the *rename* system call has been added. The rename call does the rename operation in a fashion that guarantees the

existence of the target name.

Rename works both on data files and directories. When renaming directories, the system must do special validation checks to insure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the descendents of the target directory to insure that it does not include the directory being moved.

5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of inodes and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Resources are given both a hard and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more resources while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If users fails to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

Acknowledgements

We thank Robert Elz for his ongoing interest in the new file system, and for adding disk quotas in a rational and efficient manner. We also acknowledge Dennis Ritchie for his suggestions on the appropriate modifications to the user interface. We appreciate Michael Powell's explanations on how the DEMOS file system worked; many of his ideas were used in this implementation. Special commendation goes to Peter Kessler and Robert Henry for acting like real users during the early debugging stage when file systems were less stable than they should have been. The criticisms and suggestions by the reviews contributed significantly to the coherence of the paper. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under ARPA Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

References

[Almes78] Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.

[Bass81] Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.

[Feiertag71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 35-41

[Ferrin82a] Ferrin, T.E., "Performance and Robustness Improvements in Version 7 UNIX", Computer Graphics Laboratory Technical Report 2, School of Pharmacy, University of California, San Francisco, January 1982. Presented at the 1982 Winter Usenix Conference, Santa Monica, California.

[Ferrin82b] Ferrin, T.E., "Performance Issuses of VMUNIX Revisited", ;login: (The Usenix Association Newsletter), Vol 7, #5, November 1982. pp 3-6

[Kridle83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group,

Dept of EECS, Berkeley, CA 94720, Technical Report #8.

[Kowalski78] Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978

[Knuth75] Knuth, D. "The Art of Computer Programming", Volume 3 - Sorting and Searching, Addison-Wesley Publishing Company Inc, Reading, Mass, 1975. pp 506-549

[Maruyama76] Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", CACM, 19, 11. Nov 1976. pp 634-642

[Nevalainen77] Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247

[Pechura83] Pechura, M., and Schoeffler, J. "Estimating File Access Time of Floppy Disks", CACM, 26, 10. Oct 1983. pp 754-763

[Peterson83] Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55

[Powell79] Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42

[Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

[Smith81a] Smith, A. "Input/Output Optimization and Disk Architectures: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117

[Smith81b] Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54

[Symbolics81] "Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.

[Thompson78] Thompson, K. "UNIX Implementation", Bell System Technical Journal, 57, 6, part 2. pp 1931-1946 July-August 1978.

[Thompson80] Thompson, M. "Spice File System", Carnegie-Mellon University, Department of Computer Science, Pittsburg, PA 15213 #CMU-CS-80, Sept 1980.

[Trivedi80] Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473

[White80] White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.

Analysis and Evolution of Journaling File Systems

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison
{vijayan, dusseau, remzi}@cs.wisc.edu

Abstract

We develop and apply two new methods for analyzing file system behavior and evaluating file system changes. First, *semantic block-level analysis (SBA)* combines knowledge of on-disk data structures with a trace of disk traffic to infer file system behavior; in contrast to standard benchmarking approaches, SBA enables users to understand *why* the file system behaves as it does. Second, *semantic trace playback (STP)* enables traces of disk traffic to be easily modified to represent changes in the file system implementation; in contrast to directly modifying the file system, STP enables users to rapidly gauge the benefits of new policies. We use SBA to analyze Linux ext3, ReiserFS, JFS, and Windows NTFS; in the process, we uncover many strengths and weaknesses of these journaling file systems. We also apply STP to evaluate several modifications to ext3, demonstrating the benefits of various optimizations without incurring the costs of a real implementation.

1 Introduction

Modern file systems are journaling file systems [4, 22, 29, 32]. By writing information about pending updates to a write-ahead log [12] before committing the updates to disk, journaling enables fast file system recovery after a crash. Although the basic techniques have existed for many years (*e.g.*, in Cedar [13] and Episode [9]), journaling has increased in popularity and importance in recent years; due to ever-increasing disk capacities, scan-based recovery (*e.g.*, via fsck [16]) is prohibitively slow on modern drives and RAID volumes. However, despite the popularity and importance of journaling file systems such as ext3 [32], ReiserFS [22], JFS [4], and NTFS [27] little is known about their internal policies.

Understanding how these file systems behave is important for developers, administrators, and application writers. Therefore, we believe it is time to perform a detailed analysis of journaling file systems. Most previous work has analyzed file systems from above; by writing user-level programs and measuring the time taken for various file system operations, one can elicit some salient aspects of file system performance [6, 8, 19, 26]. However, it is difficult to discover the underlying reasons for the observed performance with this approach.

In this paper we employ a novel benchmarking methodology called *semantic block-level analysis* (SBA) to trace and analyze file systems. With SBA, we induce controlled workload patterns from above the file system, but focus our analysis not only on the time taken for said operations, but also on the resulting stream of read and write requests *below* the file system. This analysis is *semantic* because we leverage information about block type (*e.g.*, whether a block request is to the journal or to an inode); this analysis is *block-level* because it interposes on the block interface to storage. By analyzing the low-level block stream in a semantically meaningful way, one can understand *why* the file system behaves as it does.

Analysis hints at how the file system could be improved, but does not reveal whether the change is worth implementing. Traditionally, for each potential improvement to the file system, one must implement the change and measure performance under various workloads; if the change gives little improvement, the implementation effort is wasted. In this paper, we introduce and apply a complementary technique to SBA called *semantic trace playback* (STP). STP enables us to rapidly suggest and evaluate file system modifications without a large implementation or simulation effort. Using real workloads and traces, we show how STP can be used effectively.

We have applied a detailed analysis to both Linux ext3 and ReiserFS and a preliminary analysis to Linux JFS and Windows NTFS. In each case, we focus on the journaling aspects of each file system. For example, we determine the events that cause data and metadata to be written to the journal or their fixed locations. We also examine how the characteristics of the workload and configuration parameters (*e.g.*, the size of the journal and the values of commit timers) impact this behavior.

Our analysis has uncovered design flaws, performance problems, and even correctness bugs in these file systems. For example, ext3 and ReiserFS make the design decision to group unrelated traffic into the same compound transaction; the result of this *tangled synchrony* is that a single disk-intensive process forces *all* write traffic to disk, particularly affecting the performance of otherwise asynchronous writers. (§3.2.1). Further, we find that both ext3 and ReiserFS artificially *limit parallelism*, by preventing the overlap of pre-commit journal writes and fixed-place updates (§3.2.2). Our analysis also reveals that in ordered and data journaling modes, ext3 exhibits *eager writing*, forcing data blocks to disk much sooner than the typical 30-second delay (§3.2.3). In addition, we find that JFS

has an *infinite write delay*, as it does not utilize commit timers and indefinitely postpones journal writes until another trigger forces writes to occur, such as memory pressure (§5). Finally, we identify four previously unknown bugs in ReiserFS that will be fixed in subsequent releases (§4.3).

The main contributions of this paper are:

• A new methodology, semantic block analysis (SBA), for understanding the internal behavior of file systems.

• A new methodology, semantic trace playback (STP), for rapidly gauging the benefits of file system modifications without a heavy implementation effort.

• A detailed analysis using SBA of two important journaling file systems, ext3 and ReiserFS, and a preliminary analysis of JFS and NTFS.

• An evaluation using STP of different design and implementation alternatives for ext3.

The rest of this paper is organized as follows. In §2 we describe our new techniques for SBA and STP. We apply these techniques to ext3, ReiserFS, JFS, and NTFS in §3, §4, §5, and §6 respectively. We discuss related work in §7 and conclude in §8.

2 Methodology

We introduce two techniques for evaluating file systems. First, semantic block analysis (SBA) enables users to understand the internal behavior and policies of the file system. Second, semantic trace playback (STP) allows users to quantify how changing the file system will impact the performance of real workloads.

2.1 Semantic Block-Level Analysis

File systems have traditionally been evaluated using one of two approaches; either one applies synthetic or real workloads and measures the resulting file system performance [6, 14, 17, 19, 20] or one collects traces to understand how file systems are used [1, 2, 21, 24, 35, 37]. However, performing each in isolation misses an interesting opportunity: by correlating the observed disk traffic with the running workload and with performance, one can often answer *why* a given workload behaves as it does.

Block-level tracing of disk traffic allows one to analyze a number of interesting properties of the file system and workload. At the coarsest granularity, one can record the *quantity* of disk traffic and how it is divided between reads and writes; for example, such information is useful for understanding how file system caching and write buffering affect performance. At a more detailed level, one can track the *block number* of each block that is read or written; by analyzing the block numbers, one can see the extent to which traffic is sequential or random. Finally, one can analyze the *timing* of each block; with timing information, one can understand when the file system initiates a burst of traffic.

By combining block-level analysis with *semantic* information about those blocks, one can infer much more about the behavior of the file system. The main difference between *semantic block analysis* (SBA) and more standard block-level tracing is that SBA analysis understands the on-disk format of the file system under test. SBA enables us to understand new properties of the file system. For example, SBA allows us to distinguish between traffic to the journal versus to in-place data and to even track individual transactions to the journal.

	Ext3	ReiserFS	JFS	NTFS
SBA Generic	1289	1289	1289	1289
SBA FS Specific	181	48	20	15
SBA Total	1470	1337	1309	1304

Table 1: **Code size of SBA drivers.** *The number of C statements (counted as the number of semicolons) needed to implement SBA for ext3 and ReiserFS and a preliminary SBA for JFS and NTFS.*

2.1.1 Implementation

The infrastructure for performing SBA is straightforward. One places a pseudo-device driver in the kernel, associates it with an underlying disk, and mounts the file system of interest (*e.g.*, ext3) on the pseudo device; we refer to this as the SBA driver. One then runs controlled microbenchmarks to generate disk traffic. As the SBA driver passes the traffic to and from the disk, it also efficiently tracks each request and response by storing a small record in a fixed-sized circular buffer. Note that by tracking the ordering of requests and responses, the pseudo-device driver can infer the order in which the requests were scheduled at lower levels of the system.

SBA requires that one interpret the *contents* of the disk block traffic. For example, one must interpret the contents of the journal to infer the type of journal block (*e.g.*, a descriptor or commit block) and one must interpret the journal descriptor block to know which data blocks are journaled. As a result, it is most efficient to semantically interpret block-level traces on-line; performing this analysis off-line would require exporting the contents of blocks, greatly inflating the size of the trace.

An SBA driver is customized to the file system under test. One concern is the amount of information that must be embedded within the SBA driver for each file system. Given that the focus of this paper is on understanding journaling file systems, our SBA drivers are embedded with enough information to interpret the placement and contents of journal blocks, metadata, and data blocks. We now analyze the complexity of the SBA driver for four journaling file systems, ext3, ReiserFS, JFS, and NTFS.

Journaling file systems have both a journal, where transactions are temporarily recorded, and fixed-location data structures, where data permanently reside. Our SBA driver distinguishes between the traffic sent to the journal and to the fixed-location data structures. This traffic is simple to distinguish in ReiserFS, JFS, and NTFS because the journal is a set of contiguous blocks, separate from the rest of the file system. However, to be backward

compatible with ext2, ext3 can treat the journal as a regular file. Thus, to determine which blocks belong to the journal, SBA uses its knowledge of inodes and indirect blocks; given that the journal does not change location after it has been created, this classification remains efficient at run-time. SBA is also able to classify the different types of journal blocks such as the descriptor block, journal data block, and commit block.

To perform useful analysis of journaling file systems, the SBA driver does not have to understand many details of the file system. For example, our driver does not understand the directory blocks or superblock of ext3 or the B+ tree structure of ReiserFS or JFS. However, if one wishes to infer additional file system properties, one may need to embed the SBA driver with more knowledge. Nevertheless, the SBA driver does not know anything about the policies or parameters of the file system; in fact, SBA can be used to infer these policies and parameters.

Table 1 reports the number of C statements required to implement the SBA driver. These numbers show that most of the code in the SBA driver (*i.e.*, 1289 statements) is for general infrastructure; only between approximately 50 and 200 statements are needed to support different journaling file systems. The ext3 specific code is more than that of the other file systems because in ext3, journal is created as a file and can span multiple block groups. In order to find the blocks belonging to the journal file, we parse the journal inode and journal indirect blocks. In Reiserfs, JFS, and NTFS the journal is contiguous and finding its blocks is trivial (even though the journal is a file in NTFS, for small journals they are contiguously allocated).

2.1.2 Workloads

SBA analysis can be used to gather useful information for any workload. However, the focus of this paper is on understanding the internal policies and behavior of the file system. As a result, we wish to construct synthetic workloads that uncover decisions made by the file system. More realistic workloads will be considered only when we apply semantic trace playback.

When constructing synthetic workloads that stress the file system, previous research has revealed a range of parameters that impact performance [8]. We have created synthetic workloads varying these parameters: the amount of data written, sequential versus random accesses, the interval between calls to `fsync`, and the amount of concurrency. We focus exclusively on write-based workloads because reads are directed to their fixed-place location, and thus do not impact the journal. When we analyze each file system, we only report results for those workloads which revealed file system policies and parameters.

2.1.3 Overhead of SBA

The processing and memory overheads of SBA are minimal for the workloads we ran as they did not generate high I/O rates. For every I/O request, the SBA driver performs the following operations to collect detailed traces:

- A gettimeofday() call during the start and end of I/O.
- A block number comparison to see if the block is a journal or fixed-location block.
- A check for a magic number on journal blocks to distinguish journal metadata from journal data.

SBA stores the trace records with details like read or write, block number, block type, time of issue and completion in an internal circular buffer. All these operations are performed only if one needs detailed traces. But for many of our analyses, it is sufficient to have cumulative statistics like the total number of journal writes and fixed-location writes. These numbers are easy to collect and require less processing within the SBA driver.

2.1.4 Alternative Approaches

One might believe that directly instrumenting a file system to obtain timing information and disk traces would be equivalent or superior to performing SBA analysis. We believe this is not the case for several reasons. First, to directly instrument the file system, one needs source code for that file system and one must re-instrument new versions as they are released; in contrast, SBA analysis does not require file system source and much of the SBA driver code can be reused across file systems and versions. Second, when directly instrumenting the file system, one may accidentally miss some of the conditions for which disk blocks are written; however, the SBA driver is guaranteed to see all disk traffic. Finally, instrumenting existing code may accidentally change the behavior of that code [36]; an efficient SBA driver will likely have no impact on file system behavior.

2.2 Semantic Trace Playback

In this section we describe semantic trace playback (STP). STP can be used to rapidly evaluate certain kinds of new file system designs, both without a heavy implementation investment and without a detailed file system simulator.

We now describe how STP functions. STP is built as a user-level process; it takes as input a trace (described further below), parses it, and issues I/O requests to the disk using the raw disk interface. Multiple threads are employed to allow for concurrency.

Ideally, STP would function by only taking a block-level trace as input (generated by the SBA driver), and indeed this is sufficient for some types of file system modifications. For example, it is straightforward to model different layout schemes by simply mapping blocks to different on-disk locations.

However, it was our desire to enable more powerful emulations with STP. For example, one issue we explore later is the effect of using byte differences in the journal, instead of storing entire blocks therein. One complication that arises is that by changing the contents of the journal,

the *timing* of block I/O changes; the thresholds that initiate I/O are triggered at a different time.

To handle emulations that alter the timing of disk I/O, more information is needed than is readily available in the low-level block trace. Specifically, STP needs to observe two high-level activities. First, STP needs to observe any file-system level operations that create dirty buffers in memory. The reason for this requirement is found in §3.2.2; when the number of uncommitted buffers reaches a threshold (in ext3, $\frac{1}{4}$ of the journal size), a commit is enacted. Similarly, when one of the interval timers expires, these blocks may have to be flushed to disk.

Second, STP needs to observe application-level calls to `fsync`; without doing so, STP cannot understand whether an I/O operation in the SBA trace is there due to a `fsync` call or due to normal file system behavior (*e.g.*, thresholds being crossed, timers going off, etc.). Without such differentiation, STP cannot emulate behaviors that are timing sensitive.

Both of these requirements are met by giving a file-system level trace as input to STP, in addition to the SBA-generated block-level trace. We currently use library-level interpositioning to trace the application of interest.

We can now qualitatively compare STP to two other standard approaches for file system evolution. In the first approach, when one has an idea for improving a file system, one simply implements the idea within the file system and measures the performance of the real system. This approach is attractive because it gives a reliable answer as to whether the idea was a real improvement, assuming that the workload applied is relevant. However, it is time consuming, particularly if the modification to the file system is non-trivial.

In the second approach, one builds an accurate simulation of the file system, and evaluates a new idea within the domain of the file system before migrating it to the real system. This approach is attractive because one can often avoid some of the details of building a real implementation and thus more quickly understand whether the idea is a good one. However, it requires a detailed and accurate simulator, the construction and maintenance of which is certainly a challenging endeavor.

STP avoids the difficulties of both of these approaches by using the low-level traces as the "truth" about how the file system behaves, and then modifying file system output (*i.e.*, the block stream) based on its simple internal models of file system behavior; these models are based on our empirical analysis found in §3.2.

Despite its advantages over traditional implementation and simulation, STP is limited in some important ways. For example, STP is best suited for evaluating design alternatives under simpler benchmarks; if the workload exhibits complex virtual memory behavior whose interactions with the file system are not modeled, the results may not be meaningful. Also, STP is limited to evaluating file system changes that are not too radical; the basic operation of the file system should remain intact. Finally, STP does not provide a means to evaluate *how* to implement a given change; rather, it should be used to understand *whether* a certain modification improves performance.

2.3 Environment

All measurements are taken on a machine running Linux 2.4.18 with a 600 MHz Pentium III processor and 1 GB of main memory. The file system under test is created on a single IBM 9LZX disk, which is separate from the root disk. Where appropriate, each data point reports the average of 30 trials; in all cases, variance is quite low.

3 The Ext3 File System

In this section, we analyze the popular Linux filesystem, ext3. We begin by giving a brief overview of ext3, and then apply semantic block-level analysis and semantic trace playback to understand its internal behavior.

3.1 Background

Linux ext3 [33, 34] is a journaling file system, built as an extension to the ext2 file system. In ext3, data and metadata are eventually placed into the standard ext2 structures, which are the fixed-location structures. In this organization (which is loosely based on FFS [15]), the disk is split into a number of *block groups*; within each block group are bitmaps, inode blocks, and data blocks. The ext3 journal (or log) is commonly stored as a file within the file system, although it can be stored on a separate device or partition. Figure 1 depicts the ext3 on-disk layout.

Information about pending file system updates is written to the journal. By forcing journal updates to disk *before* updating complex file system structures, this write-ahead logging technique [12] enables efficient crash recovery; a simple scan of the journal and a redo of any incomplete committed operations bring the file system to a consistent state. During normal operation, the journal is treated as a circular buffer; once the necessary information has been propagated to its fixed location in the ext2 structures, journal space can be reclaimed.

Journaling Modes: Linux ext3 includes three flavors of journaling: *writeback mode*, *ordered mode*, and *data journaling mode*; Figure 2 illustrates the differences between these modes. The choice of mode is made at mount time and can be changed via a remount.

In *writeback mode*, only file system metadata is journaled; data blocks are written directly to their fixed location. This mode does not enforce any ordering between the journal and fixed-location data writes, and because of this, writeback mode has the weakest consistency semantics of the three modes. Although it guarantees consistent file system metadata, it does not provide any guarantee as to the consistency of data blocks.

In *ordered journaling mode*, again only metadata writes are journaled; however, data writes to their fixed location are ordered *before* the journal writes of the metadata. In

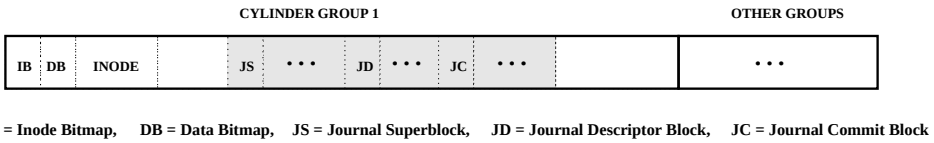


Figure 1: **Ext3 On-Disk Layout.** *The picture shows the layout of an ext3 file system. The disk address space is broken down into a series of block groups (akin to FFS cylinder groups), each of which has bitmaps to track allocations and regions for inodes and data blocks. The ext3 journal is depicted here as a file within the first block group of the file system; it contains a superblock, various descriptor blocks to describe its contents, and commit blocks to denote the ends of transactions.*

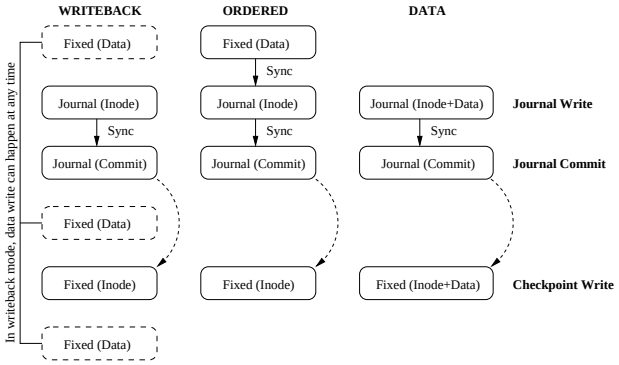


Figure 2: **Ext3 Journaling Modes.** *The diagram depicts the three different journaling modes of ext3: writeback, ordered, and data. In the diagram, time flows downward. Boxes represent updates to the file system, e.g., "Journal (Inode)" implies the write of an inode to the journal; the other destination for writes is labeled "Fixed", which is a write to the fixed in-place ext2 structures. An arrow labeled with a "Sync" implies that the two blocks are written out in immediate succession synchronously, hence guaranteeing the first completes before the second. A curved arrow indicates ordering but not immediate succession; hence, the second write will happen at some later time. Finally, for writeback mode, the dashed box around the "Fixed (Data)" block indicates that it may happen at any time in the sequence. In this example, we consider a data block write and its inode as the updates that need to be propagated to the file system; the diagrams show how the data flow is different for each of the ext3 journaling modes.*

contrast to writeback mode, this mode provides more sensible consistency semantics, where both the data and the metadata are guaranteed to be consistent after recovery.

In full *data journaling mode*, ext3 logs *both* metadata and data to the journal. This decision implies that when a process writes a data block, it will typically be written out to disk *twice*: once to the journal, and then later to its fixed ext2 location. Data journaling mode provides the same strong consistency guarantees as ordered journaling mode; however, it has different performance characteristics, in some cases worse, and surprisingly, in some cases, better. We explore this topic further (§3.2).

Transactions: Instead of considering each file system update as a separate transaction, ext3 groups many updates into a single *compound transaction* that is periodically committed to disk. This approach is relatively simple to implement [33]. Compound transactions may have better performance than more fine-grained transactions when the same structure is frequently updated in a short period of time (*e.g.*, a free space bitmap or an inode of a file that is constantly being extended) [13].

Journal Structure: Ext3 uses additional metadata structures to track the list of journaled blocks. The *journal superblock* tracks summary information for the journal, such as the block size and head and tail pointers. A *journal descriptor block* marks the beginning of a transaction and describes the subsequent journaled blocks, including their final fixed on-disk location. In data journaling mode, the descriptor block is followed by the data and metadata blocks; in ordered and writeback mode, the descriptor block is followed by the metadata blocks. In all modes, ext3 logs full blocks, as opposed to differences from old versions; thus, even a single bit change in a bitmap results in the entire bitmap block being logged. Depending upon the size of the transaction, multiple descriptor blocks each followed by the corresponding data and metadata blocks may be logged. Finally, a *journal commit block* is written to the journal at the end of the transaction; once the commit block is written, the journaled data can be recovered without loss.

Checkpointing: The process of writing journaled metadata and data to their fixed-locations is known as *checkpointing*. Checkpointing is triggered when various thresholds are crossed, *e.g.*, when file system buffer space is low, when there is little free space left in the journal, or when a timer expires.

Crash Recovery: Crash recovery is straightforward in ext3 (as it is in many journaling file systems); a basic form of *redo logging* is used. Because new updates (whether to data or just metadata) are written to the log, the process of restoring in-place file system structures is easy. During recovery, the file system scans the log for committed complete transactions; incomplete transactions are discarded. Each update in a completed transaction is simply replayed into the fixed-place ext2 structures.

3.2 Analysis of ext3 with SBA

We now perform a detailed analysis of ext3 using our SBA framework. Our analysis is divided into three categories. First, we analyze the basic behavior of ext3 as a function of the workload and the three journaling modes. Second, we isolate the factors that control when data is committed to the journal. Third, we isolate the factors that control when data is checkpointed to its fixed-place location.

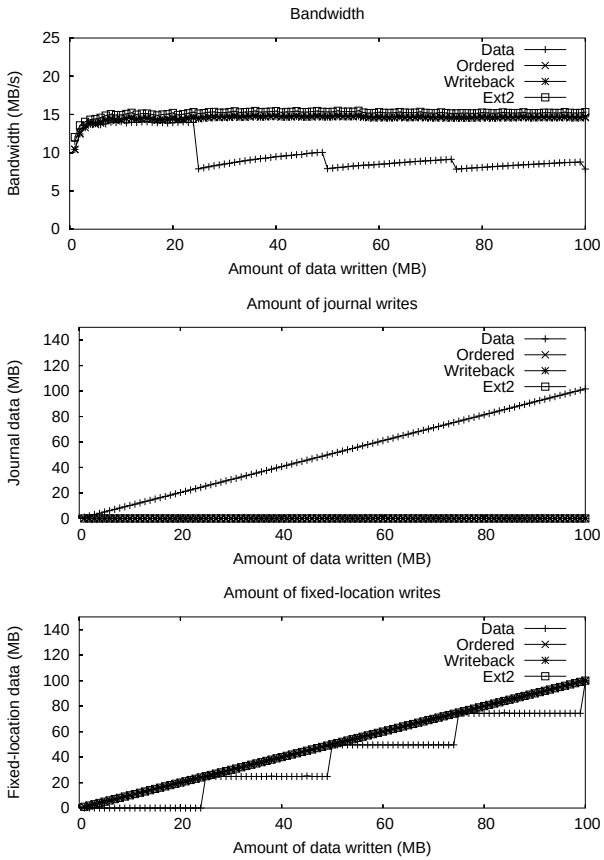


Figure 3: **Basic Behavior for Sequential Workloads in ext3.** *Within each graph, we evaluate ext2 and the three ext3 journaling modes. We increase the size of the written file along the x-axis. The workload writes to a single file sequentially and then performs an* `fsync`*. Each graph examines a different metric: the top graph shows the achieved bandwidth; the middle graph uses SBA to report the amount of journal traffic; the bottom graph uses SBA to report the amount of fixed-location traffic. The journal size is set to 50 MB.*

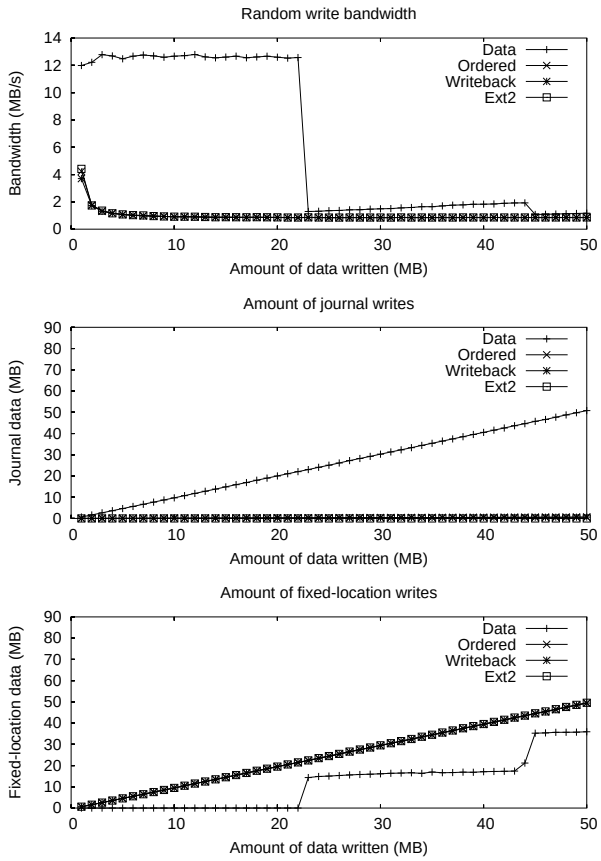


Figure 4: **Basic Behavior for Random Workloads in ext3.** *This figure is similar to Figure 3. The workload issues 4 KB writes to random locations in a single file and calls* `fsync` *once for every 256 writes. Top graph shows the bandwidth, middle graph shows the journal traffic, and the bottom graph reports the fixed-location traffic. The journal size is set to 50 MB.*

3.2.1 Basic Behavior: Modes and Workload

We begin by analyzing the basic behavior of ext3 as a function of the workload and journaling mode (*i.e.*, writeback, ordered, and full data journaling). Our goal is to understand the workload conditions that trigger ext3 to write data and metadata to the journal and to their fixed locations. We explored a range of workloads by varying the amount of data written, the sequentiality of the writes, the synchronization interval between writes, and the number of concurrent writers.

Sequential and Random Workloads: We begin by showing our results for three basic workloads. The first workload writes to a single file sequentially and then performs an `fsync` to flush its data to disk (Figure 3); the second workload issues 4 KB writes to random locations in a single file and calls `fsync` once for every 256 writes (Figure 4); the third workload again issues 4 KB random writes but calls `fsync` for every write (Figure 5). In each workload, we increase the total amount of data that it writes and observe how the behavior of ext3 changes.

The top graphs in Figures 3, 4, and 5 plot the achieved bandwidth for the three workloads; within each graph, we compare the three different journaling modes and ext2. From these bandwidth graphs we make four observations. First, the achieved bandwidth is extremely sensitive to the workload: as expected, a sequential workload achieves much higher throughput than a random workload and calling `fsync` more frequently further reduces throughput for random workloads. Second, for sequential traffic, ext2 performs slightly better than the highest performing ext3 mode: there is a small but noticeable cost to journaling for sequential streams. Third, for all workloads, ordered mode and writeback mode achieve bandwidths that are similar to ext2. Finally, the performance of data journaling is quite irregular, varying in a sawtooth pattern with the amount of data written.

These graphs of file system throughput allow us to compare performance across workloads and journaling modes, but do not enable us to infer the *cause* of the differences. To help us infer the internal behavior of the file system, we apply semantic analysis to the underlying block stream;

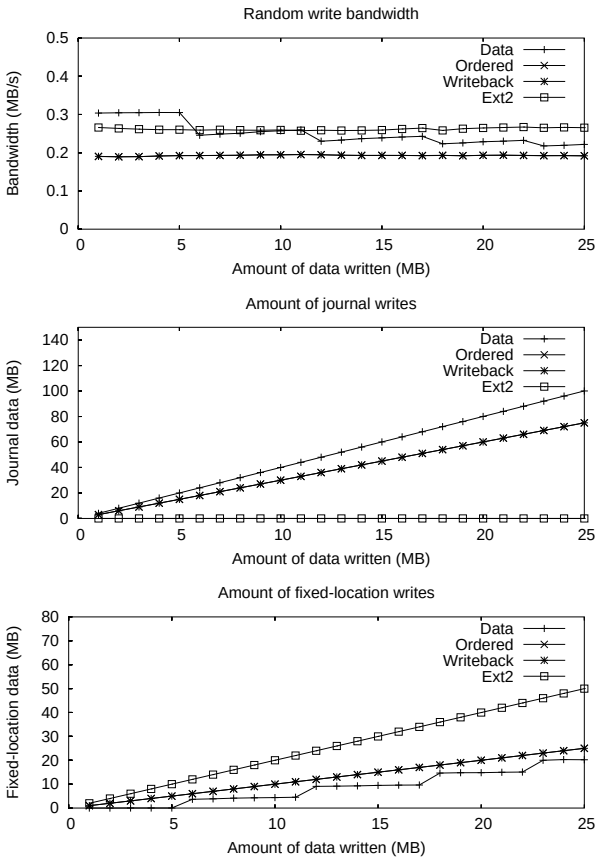


Figure 5: **Basic Behavior for Random Workloads in ext3.** *This figure is similar to Figure 3. The workload issues 4 KB random writes and calls* `fsync` *for every write. Bandwidth is shown in the first graph; journal writes and fixed-location writes are reported in the second and third graph using SBA. The journal size is set to 50 MB.*

in particular, we record the amount of journal and fixed-location traffic. This accounting is shown in the bottom two graphs of Figures 3, 4, and 5.

The second row of graphs in Figures 3, 4, and 5 quantify the amount of traffic flushed to the journal and help us to infer the events which cause this traffic. We see that, in data journaling mode, the total amount of data written to the journal is high, proportional to the amount of data written by the application; this is as expected, since both data and metadata are journaled. In the other two modes, only metadata is journaled; therefore, the amount of traffic to the journal is quite small.

The third row of Figures 3, 4, and 5 shows the traffic to the fixed location. For writeback and ordered mode the amount of traffic written to the fixed location is equal to the amount of data written by the application. However, in data journaling mode, we observe a stair-stepped pattern in the amount of data written to the fixed location. For example, with a file size of 20 MB, even though the process has called `fsync` to force the data to disk, no data is written to the fixed location by the time the application terminates; because all data is logged, the expected consistency semantics are still preserved. However, even though it is not necessary for consistency, when the application writes more data, checkpointing does occur at regular intervals; this extra traffic leads to the sawtooth bandwidth measured in the first graph. In this particular experiment with sequential traffic and a journal size of 50 MB, a checkpoint occurs when 25 MB of data is written; we explore the relationship between checkpoints and journal size more carefully in §3.2.3.

The SBA graphs also reveal why data journaling mode performs better than the other modes for asynchronous random writes. With data journaling mode, all data is written first to the log, and thus even random writes become *logically* sequential and achieve sequential bandwidth. As the journal is filled, checkpointing causes extra disk traffic, which reduces bandwidth; in this particular experiment, the checkpointing occurs near 23 MB. Finally, SBA analysis reveals that synchronous 4 KB writes do not perform well, even in data journaling mode. Forcing each small 4 KB write to the log, even in logical sequence, incurs a delay between sequential writes (not shown) and thus each write incurs a disk rotation.

Concurrency: We now report our results from running workloads containing multiple processes. We construct a workload containing two diverse classes of traffic: an asynchronous foreground process in competition with a background process. The foreground process writes out a 50 MB file without calling `fsync`, while the background process repeatedly writes a 4 KB block to a random location, optionally calls `fsync`, and then sleeps for some period of time (*i.e.*, the "sync interval"). We focus on data journaling mode, but the effect holds for ordered journaling mode too (not shown).

In Figure 6 we show the impact of varying the mean "sync interval" of the background process on the performance of the foreground process. The first graph plots the bandwidth achieved by the foreground asynchronous process, depending upon whether it competes against an asynchronous or synchronous background process. As expected, when the foreground process runs with an asynchronous background process, its bandwidth is uniformly high and matches in-memory speeds. However, when the foreground process competes with a synchronous background process, its bandwidth drops to disk speeds.

The SBA analysis in the second graph reports the amount of journal data, revealing that the more frequently the background process calls `fsync`, the more traffic is sent to the journal. In fact, the amount of journal traffic is equal to the sum of the foreground and background process traffic written in that interval, not that of only the background process. This effect is due to the implementation of compound transactions in ext3: all file system updates add their changes to a global transaction, which is eventually committed to disk.

This workload reveals the potentially disastrous consequences of grouping unrelated updates into the same com-

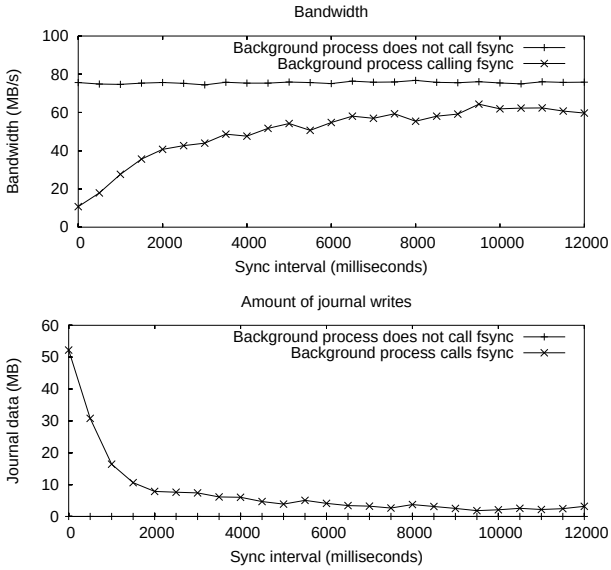


Figure 6: **Basic Behavior for Concurrent Writes in ext3.** *Two processes compete in this workload: a foreground process writing a sequential file of size 50 MB and a background process writing out 4 KB, optionally calling* `fsync`*, sleeping for the "sync interval", and then repeating. Along the x-axis, we increase the sync interval. In the top graph, we plot the bandwidth achieved by the foreground process in two scenarios: with the background process either calling or not calling* `fsync` *after each write. In the bottom graph, the amount of data written to disk during both sets of experiments is shown.*

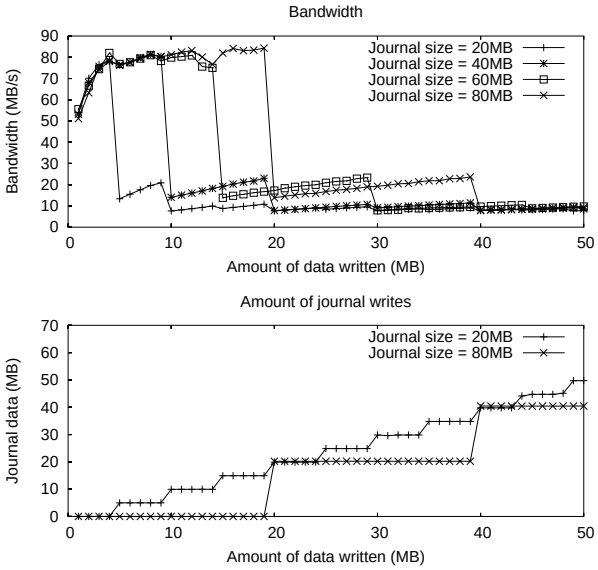


Figure 7: **Impact of Journal Size on Commit Policy in ext3.** *The topmost figure plots the bandwidth of data journaling mode under different-sized file writes. Four lines are plotted representing four different journal sizes. The second graph shows the amount of log traffic generated for each of the experiments (for clarity, only two of the four journal sizes are shown).*

pound transaction: all traffic is committed to disk at the same rate. Thus, even asynchronous traffic must wait for synchronous updates to complete. We refer to this negative effect as *tangled synchrony* and explore the benefits of untangling transactions in §3.3.3 using STP.

3.2.2 Journal Commit Policy

We next explore the conditions under which ext3 commits transactions to its on-disk journal. As we will see, two factors influence this event: the size of the journal and the settings of the commit timers.

In these experiments, we focus on data journaling mode; since this mode writes both metadata and data to the journal, the traffic sent to the journal is most easily seen in this mode. However, writeback and ordered modes commit transactions using the same policies. To exercise log commits, we examine workloads in which data is not explicitly forced to disk by the application (*i.e.*, the process does not call `fsync`); further, to minimize the amount of metadata overhead, we write to a single file.

Impact of Journal Size: The size of the journal is a configurable parameter in ext3 that contributes to when updates are committed. By varying the size of the journal and the amount of data written in the workload, we can infer the amount of data that triggers a log commit. Figure 7 shows the resulting bandwidth and the amount of journal traffic, as a function of file size and journal size. The first graph shows that when the amount of data written by the application (to be precise, the number of dirty uncommitted buffers, which includes both data and metadata) reaches $\frac{1}{4}$ the size of the journal, bandwidth drops considerably. In fact, in the first performance regime, the observed bandwidth is equal to in-memory speeds.

Our semantic analysis, shown in the second graph, reports the amount of traffic to the journal. This graph reveals that metadata and data are forced to the journal when it is equal to $\frac{1}{4}$ the journal size. Inspection of Linux ext3 code confirms this threshold. Note that the threshold is the same for ordered and writeback modes (not shown); however, it is triggered much less frequently since only metadata is logged.

Impact of Timers: In Linux 2.4 ext3, three timers have some control over when data is written: the metadata commit timer and the data commit timer, both managed by the kupdate daemon, and the commit timer managed by the kjournal daemon. The system-wide kupdate daemon is responsible for flushing dirty buffers to disk; the kjournal daemon is specialized for ext3 and is responsible for committing ext3 transactions. The strategy for ext2 is to flush metadata frequently (*e.g.*, every 5 seconds) while delaying data writes for a longer time (*e.g.*, every 30 seconds). Flushing metadata frequently has the advantage that the file system can approach FFS-like consistency without a severe performance penalty; delaying data writes has the advantage that files that are deleted quickly do not tax the disk. Thus, mapping the ext2 goals to the ext3 timers leads to default values of 5 seconds for the kupdate metadata timer, 5 seconds for the kjournal timer,

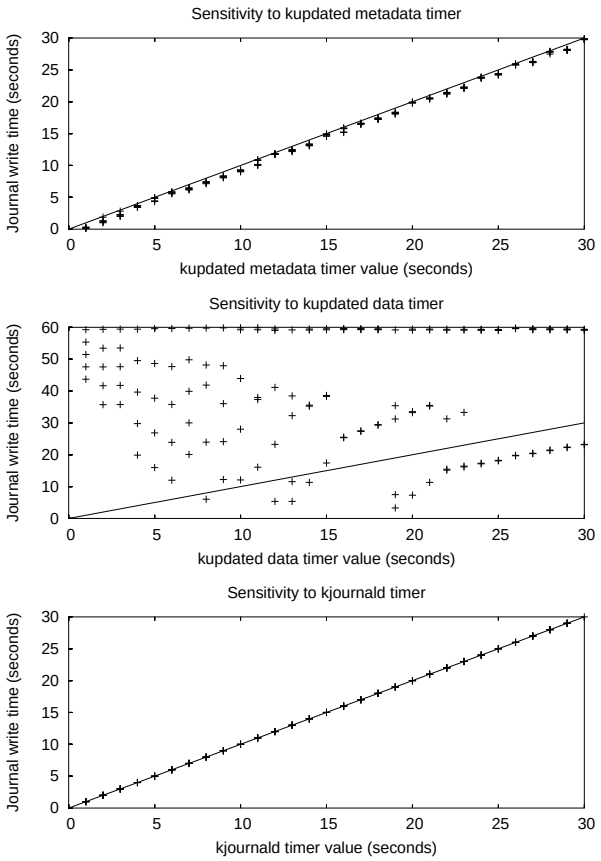


Figure 8: **Impact of Timers on Commit Policy in ext3.** *In each graph, the value of one timer is varied across the x-axis, and the time of the first write to the journal is recorded along the y-axis. When measuring the impact of a particular timer, we set the other timers to 60 seconds and the journal size to 50 MB so that they do not affect the measurements.*

and 30 seconds for the kupdate data timer.

We measure how these timers affect when transactions are committed to the journal. To ensure that a specific timer influences journal commits, we set the journal size to be sufficiently large and set the other timers to a large value (*i.e.*, 60 s). For our analysis, we observe when the first write appears in the journal. Figure 8 plots our results varying one of the timers along the x-axis, and plotting the time that the first log write occurs along the y-axis.

The first graph and the third graph show that the kupdate daemon metadata commit timer and the kjournal daemon commit timer control the timing of log writes: the data points along $y = x$ indicate that the log write occurred precisely when the timer expired. Thus, traffic is sent to the log at the minimum of those two timers. The second graph shows that the kupdate daemon data timer does not influence the timing of log writes: the data points are not correlated with the x-axis. As we will see, this timer influences when data is written to its fixed location.

Interaction of Journal and Fixed-Location Traffic: The timing between writes to the journal and to the fixed-location data must be managed carefully for consistency. In fact, the difference between writeback and ordered mode is in this timing: writeback mode does not enforce any ordering between the two, whereas ordered mode ensures that the data is written to its fixed location before the commit block for that transaction is written to the journal. When we performed our SBA analysis, we found a performance deficiency in how ordered mode is implemented.

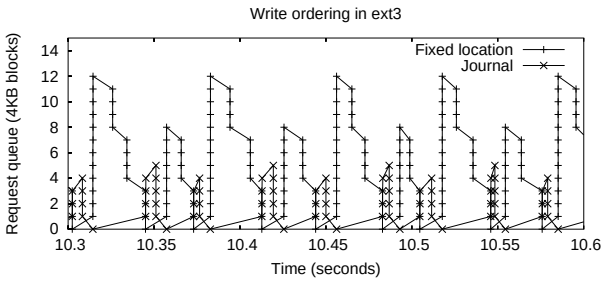


Figure 9: **Interaction of Journal and Fixed-Location Traffic in ext3.** *The figure plots the number of outstanding writes to the journal and fixed-location disks. In this experiment, we run five processes, each of which issues 16 KB random synchronous writes. The file system has a 50 MB journal and is running in ordered mode; the journal is configured to run on a separate disk.*

We consider a workload that synchronously writes a large number of random 16 KB blocks and use the SBA driver to separate journal and fixed-location data. Figure 9 plots the number of concurrent writes to each data type over time. The figure shows that writes to the journal and fixed-place data do *not* overlap. Specifically, ext3 issues the data writes to the fixed location and waits for completion, then issues the journal writes to the journal and again waits for completion, and finally issues the final commit block and waits for completion. We observe this behavior irrespective of whether the journal is on a separate device or on the same device as the file system. Inspection of the ext3 code confirms this observation. However, the first wait is not needed for correctness. In those cases where the journal is configured on a separate device, this extra wait can severely limit concurrency and performance. Thus, ext3 has *falsely limited parallelism*. We will use STP to fix this timing problem in §3.3.4.

3.2.3 Checkpoint Policy

We next turn our attention to checkpointing, the process of writing data to its fixed location within the ext2 structures. We will show that checkpointing in ext3 is again a function of the journal size and the commit timers, as well as the synchronization interval in the workload. We focus on data journaling mode since it is the most sensitive to journal size. To understand when checkpointing occurs, we construct workloads that periodically force data to the journal (*i.e.*, call `fsync`) and we observe when data is subsequently written to its fixed location.

Impact of Journal Size: Figure 10 shows our SBA results

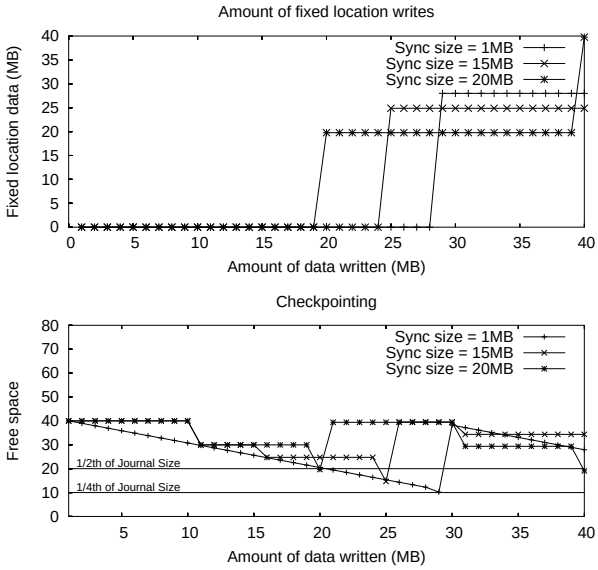


Figure 10: **Impact of Journal Size on Checkpoint Policy in ext3.** *We consider a workload where a certain amount of data (as indicated by the x-axis value) is written sequentially, with a* `fsync` *issued after every 1, 15, or 20 MB. The first graph uses SBA to plot the amount of fixed-location traffic. The second graph uses SBA to plot the amount of free space in the journal.*

as a function of file size and synchronization interval for a single journal size of 40 MB. The first graph shows the amount of data written to its fixed ext2 location at the end of each experiment. We can see that the point at which checkpointing occurs varies across the three sync intervals; for example, with a 1 MB sync interval (*i.e.*, when data is forced to disk after every 1 MB worth of writes), checkpoints occur after approximately 28 MB has been committed to the log, whereas with a 20 MB sync interval, checkpoints occur after 20 MB. To illustrate what triggers a checkpoint, in the second graph, we plot the amount of journal free space immediately preceding the checkpoint. By correlating the two graphs, we see that checkpointing occurs when the amount of free space is between $\frac{1}{4}$-th and $\frac{1}{2}$-th of the journal size. The precise fraction depends upon the synchronization interval, where smaller sync amounts allow checkpointing to be postponed until there is less free space in the journal.[1] We have confirmed this same relationship for other journal sizes (not shown).

Impact of Timers: We examine how the system timers impact the timing of checkpoint writes to the fixed locations using the same workload as above. Here, we vary the kupdate data timer while setting the other timers to five seconds. Figure 11 shows how the kupdate data timer impacts when data is written to its fixed location. First, as seen previously in Figure 8, the log is updated after the five second timers expire. Then, the checkpoint write occurs later by the amount specified by the kupdate data timer, at a five second granularity; further experiments (not shown here) reveal that this granularity is controlled by the kupdate metadata timer.

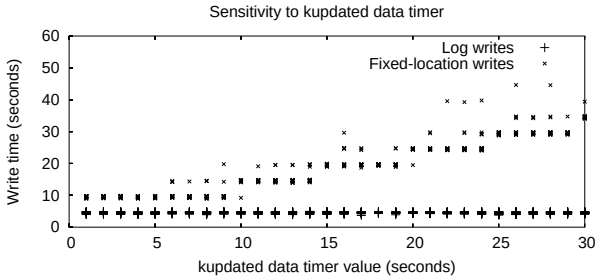


Figure 11: **Impact of Timers on Checkpoint Policy in ext3.** *The figure plots the relationship between the time that data is first written to the log and then checkpointed as dependent on the value of the kupdate data timer. The scatter plot shows the results of multiple (30) runs. The process that is running writes 1 MB of data (no* `fsync`*); data journaling mode is used, with other timers set to 5 seconds and a journal size of 50 MB.*

Our analysis reveals that the ext3 timers do not lead to the same timing of data and metadata traffic as in ext2. Ordered and data journaling modes force data to disk either before or at the time of metadata writes. Thus, *both* data and metadata are flushed to disk frequently. This timing behavior is the largest potential performance differentiator between ordered and writeback modes. Interestingly, this frequent flushing has a potential advantage; by forcing data to disk in a more timely manner, large disk queues can be avoided and overall performance improved [18]. The disadvantage of early flushing, however, is that temporary files may be written to disk before subsequent deletion, increasing the overall load on the I/O system.

3.2.4 Summary of Ext3

Using SBA, we have isolated a number of features within ext3 that can have a strong impact on performance.

- The journaling mode that delivers the best performance depends strongly on the workload. It is well known that random workloads perform better with logging [25]; however, the relationship between the size of the journal and the amount of data written by the application can have an even larger impact on performance.
- Ext3 implements compound transactions in which unrelated concurrent updates are placed into the same transaction. The result of this *tangled synchrony* is that all traffic in a transaction is committed to disk at the same rate, which results in disastrous performance for asynchronous traffic when combined with synchronous traffic.

[1] The exact amount of free space that triggers a checkpoint is not straightforward to derive for two reasons. First, ext3 reserves some amount of journal space for overhead such as descriptor and commit blocks. Second, ext3 reserves space in the journal for the currently committing transaction (*i.e.*, the synchronization interval). Although we have derived the free space function more precisely, we do not feel this very detailed information is particularly enlightening; therefore, we simply say that checkpointing occurs when free space is somewhere between $\frac{1}{4}$-th and $\frac{1}{2}$-th of the journal size.

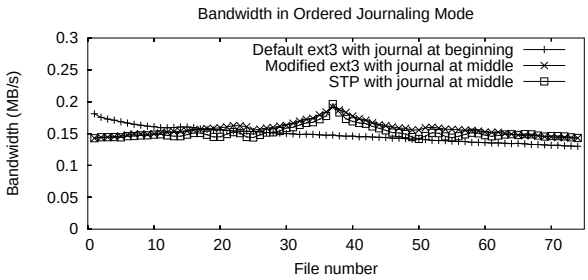


Figure 12: **Improved Journal Placement with STP.** *We compare three placements of the journal: at the beginning of the partition (the ext3 default), modeled in the middle of the file system using STP, and in the middle of the file system. 50 MB files are created across the file system; a file is chosen, as indicated by the number along the x-axis, and the workload issues 4 KB synchronous writes to that file.*

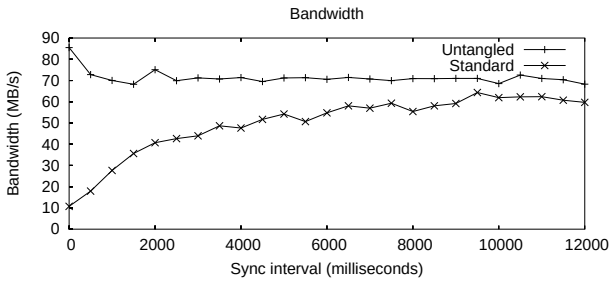


Figure 13: **Untangling Transaction Groups with STP.** *This experiment is identical to that described in Figure 6, with one addition: we show performance of the foreground process with untangled transactions as emulated with STP.*

• In ordered mode, ext3 does not overlap any of the writes to the journal and fixed-place data. Specifically, ext3 issues the data writes to the fixed location and waits for completion, then issues the journal writes to the journal and again waits for completion, and finally issues the final commit block and waits for completion; however, the first wait is not needed for correctness. When the journal is placed on a separate device, this *falsely limited parallelism* can harm performance.

• In ordered and data journaling modes, when a timer flushes meta-data to disk, the corresponding data must be flushed as well. The disadvantage of this *eager writing* is that temporary files may be written to disk, increasing the I/O load.

3.3 Evolving ext3 with STP

In this section, we apply STP and use a wider range of workloads and traces to evaluate various modifications to ext3. To demonstrate the accuracy of the STP approach, we begin with a simple modification that varies the placement of the journal. Our SBA analysis pointed to a number of improvements for ext3, which we can quantify with STP: the value of using different journaling modes depending upon the workload, having separate transactions for each update, and overlapping pre-commit journal writes with data updates in ordered mode. Finally, we use STP to evaluate *differential journaling*, in which block differences are written to the journal.

3.3.1 Journal Location

Our first experiment with STP quantifies the impact of changing a simple policy: the placement of the journal. The default ext3 creates the journal as a regular file at the beginning of the partition. We start with this policy because we are able to validate STP: the results we obtain with STP are quite similar to those when we implement the change within ext3 itself.

We construct a workload that stresses the placement of the journal: a 4 GB partition is filled with 50 MB files and the benchmark process issues random, synchronous 4 KB writes to a chosen file. In Figure 12 we vary which file is chosen along the x-axis. The first line in the graph shows the performance for ordered mode in default ext3: bandwidth drops by nearly 30% when the file is located far from the journal. SBA analysis (not shown) confirms that this performance drop occurs as the seek distance increases between the writes to the file and the journal.

To evaluate the benefit of placing the journal in the middle of the disk, we use STP to remap blocks. For validation, we also coerce ext3 to allocate its journal in the middle of the disk, and compare results. Figure 12 shows that the STP predicted performance is nearly identical to this version of ext3. Furthermore, we see that worst-case behavior is avoided; by placing the journal in the middle of the file system instead of at the beginning, the longest seeks across the entire volume are avoided during synchronous workloads (*i.e.*, workloads that frequently seek between the journal and the ext2 structures).

3.3.2 Journaling Mode

As shown in §3.2.1, different workloads perform better with different journaling modes. For example, random writes perform better in data journaling mode as the random writes are written sequentially into the journal, but large sequential writes perform better in ordered mode as it avoids the extra traffic generated by data journaling mode. However, the journaling mode in ext3 is set at mount time and remains fixed until the next mount.

Using STP, we evaluate a new adaptive journaling mode that chooses the journaling mode for each transaction according to writes that are in the transaction. If a transaction is sequential, it uses ordered journaling; otherwise, it uses data journaling.

To demonstrate the potential performance benefits of adaptive journaling, we run a portion of a trace from HP Labs [23] after removing the inter-arrival times between the I/O calls and compare ordered mode, data journaling mode, and our adaptive approach. The trace completes in 83.39 seconds and 86.67 seconds, in ordered and data journaling modes, respectively; however, with STP adaptive journaling, the trace completes in only 51.75 seconds. Because the trace has both sequential and random write

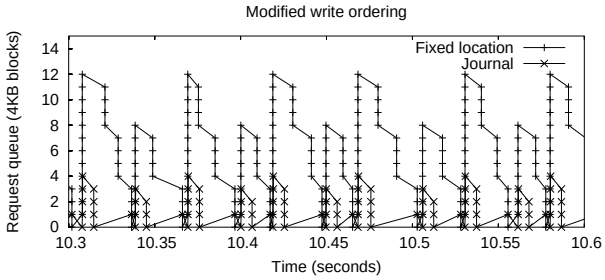


Figure 14: **Changing the Interaction of Journal and Fixed-Location Traffic with STP.** *The same experiment is run as in Figure 9; however, in this run, we use STP to issue the pre-commit journal writes and data writes concurrently. We plot the STP emulated performance, and also made this change to ext3 directly, obtaining the same resultant performance.*

phases, adaptive journaling out performs any single-mode approach.

3.3.3 Transaction Grouping

Linux ext3 groups all updates into system-wide compound transactions and commits them to disk periodically. However, as we have shown in 3.2.1, if just a single update stream is synchronous, it can have a dramatic impact on the performance of other asynchronous streams, by transforming in-memory updates into disk-bound ones.

Using STP, we show the performance of a file system that *untangles* these traffic streams, only forcing the process that issues the `fsync` to commit its data to disk. Figure 13 plots the performance of an asynchronous sequential stream in the presence of a random synchronous stream. Once again, we vary the interval of updates from the synchronous process, and from the graph, we can see that segregated transaction grouping is effective; the asynchronous I/O stream is unaffected by synchronous traffic.

3.3.4 Timing

We show that STP can quantify the cost of falsely limited parallelism, as discovered in 3.2.2, where pre-commit journal writes are not overlapped with data updates in ordered mode. With STP, we modify the timing so that journal and fixed-location writes are all initiated simultaneously; the commit transaction is written only after the previous writes complete. We consider the same workload of five processes issuing 16 KB random synchronous writes and with the journal on a separate disk.

Figure 14 shows that STP can model this implementation change by modifying the timing of the requests. For this workload, STP predicts an improvement of about 18%; this prediction matches what we achieve when ext3 is changed directly. Thus, as expected, increasing the amount of concurrency improves performance when the journal is on a separate device.

3.3.5 Journal Contents

Ext3 uses physical logging and writes new blocks in their entirety to the log. However, if whole blocks are journaled irrespective of how many bytes have changed in the block, journal space fills quickly, increasing both commit and checkpoint frequency.

Using STP, we investigate *differential journaling*, where the file system writes block differences to the journal instead of new blocks in their entirety. This approach can potentially reduce disk traffic noticeably, if dirty blocks are not substantially different from their previous versions. We focus on data journaling mode, as it generates by far the most journal traffic; differential journaling is less useful for the other modes.

To evaluate whether differential journaling matters for real workloads, we analyze SBA traces underneath two database workloads modeled on TPC-B [30] and TPC-C [31]. The former is a simple application-level implementation of a debit-credit benchmark, and the latter a realistic implementation of order-entry built on top of Postgres. With data journaling mode, the amount of data written to the journal is reduced by a factor of 200 for TPC-B and a factor of 6 under TPC-C. In contrast, for ordered and writeback modes, the difference is minimal (less than 1%); in these modes, only metadata is written to the log, and applying differential journaling to said metadata blocks makes little difference in total I/O volume.

4 ReiserFS

We now focus on a second Linux journaling filesystem, ReiserFS. In this section, we focus on the chief differences between ext3 and ReiserFS. Due to time constraints, we do not use STP to explore changes to ReiserFS.

4.1 Background

The general behavior of ReiserFS is similar to ext3. For example, both file systems have the same three journaling modes and both have compound transactions. However, ReiserFS differs from ext3 in three primary ways.

First, the two file systems use different on-disk structures to track their fixed-location data. Ext3 uses the same structures as ext2; for improved scalability, ReiserFS uses a B+ tree, in which data is stored on the leaves of the tree and the metadata is stored on the internal nodes. Since the impact of the fixed-location data structures is not the focus of this paper, this difference is largely irrelevant.

Second, the format of the journal is slightly different. In ext3, the journal can be a file, which may be anywhere in the partition and may not be contiguous. The ReiserFS journal is not a file and is instead a contiguous sequence of blocks at the beginning of the file system; as in ext3, the ReiserFS journal can be put on a different device. Further, ReiserFS limits the journal to a maximum of 32 MB.

Third, ext3 and ReiserFS differ slightly in their journal contents. In ReiserFS, the fixed locations for the blocks in the transaction are stored not only in the descriptor block but also in the commit block. Also, unlike ext3, ReiserFS uses only one descriptor block in every compound

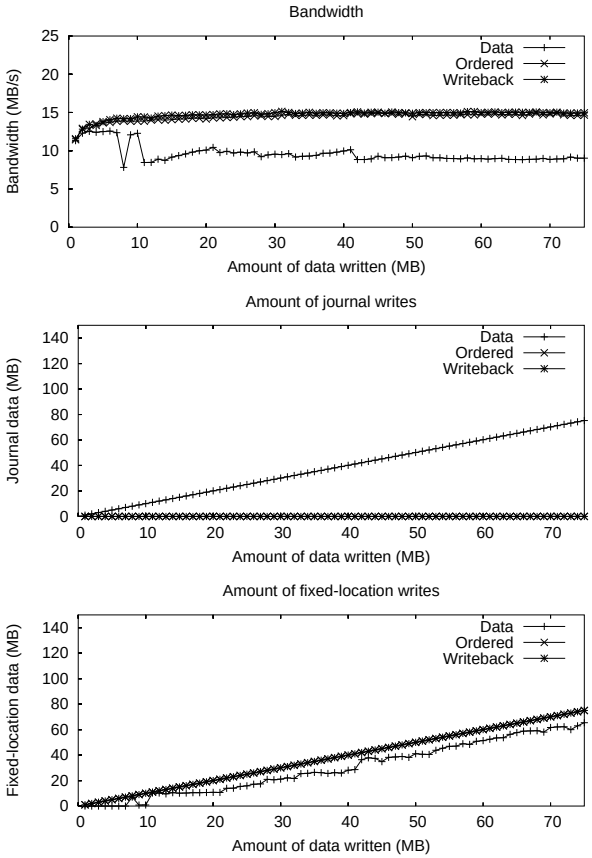


Figure 15: **Basic Behavior for Sequential Workloads in ReiserFS.** *Within each graph, we evaluate the three ReiserFS journaling modes. We consider a single workload in which the size of the sequentially written file is increased along the x-axis. Each graph examines a different metric: the first hows the achieved bandwidth; the second uses SBA to report the amount of journal traffic; the third uses SBA to report the amount of fixed-location traffic. The journal size is set to 32 MB.*

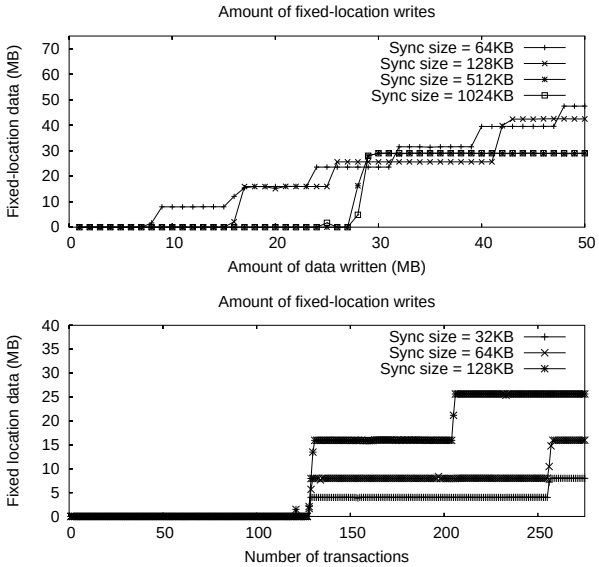


Figure 16: **Impact of Journal Size and Transactions on Checkpoint Policy in ReiserFS.** *We consider workloads where data is sequentially written and an* `fsync` *is issued after a specified amount of data. We use SBA to report the amount of fixed-location traffic. In the first graph, we vary the amount of data written; in the second graph, we vary the number of transactions, defined as the number of calls to* `fsync`.

transaction, which limits the number of blocks that can be grouped in a transaction.

4.2 Semantic Analysis of ReiserFS

We have performed identical experiments on ReiserFS as we have on ext3. Due to space constraints, we present only those results which reveal significantly different behavior across the two file systems.

4.2.1 Basic Behavior: Modes and Workload

Qualitatively, the performance of the three journaling modes in ReiserFS is similar to that of ext3: random workloads with infrequent synchronization perform best with data journaling; otherwise, sequential workloads generally perform better than random ones and writeback and ordered modes generally perform better than data journaling. Furthermore, ReiserFS groups concurrent transactions into a single compound transaction, as did ext3. The primary difference between the two file systems occurs for sequential workloads with data journaling. As shown in the first graph of Figure 15, the throughput of data journaling mode in ReiserFS does not follow the sawtooth pattern. An initial reason for this is found through SBA analysis. As seen in the second and third graphs of Figure 15, almost all of the data is written not only to the journal, but is also checkpointed to its in-place location. Thus, ReiserFS appears to checkpoint data much more aggressively than ext3, which we will explore in §4.2.3.

4.2.2 Journal Commit Policy

We explore the factors that impact when ReiserFS commits transactions to the log. Again, we focus on data journaling, since it is the most sensitive. We postpone exploring the impact of the timers until §4.2.3.

We previously saw that ext3 commits data to the log when approximately $\frac{1}{4}$ of the log is filled or when a timer expires. Running the same workload that does not force data to disk (*i.e.*, does not call `fsync`) on ReiserFS and performing SBA analysis, we find that ReiserFS uses a different threshold: depending upon whether the journal size is below or above 8 MB, ReiserFS commits data when about 450 blocks (*i.e.*, 1.7 MB) or 900 blocks (*i.e.*, 3.6 MB) are written. Given that ReiserFS limits journal size to at most 32 MB, these fixed thresholds appear sufficient.

Finally, we note that ReiserFS also has falsely limited parallelism in ordered mode. Like ext3, ReiserFS forces the data to be flushed to its fixed location before it issues any writes to the journal.

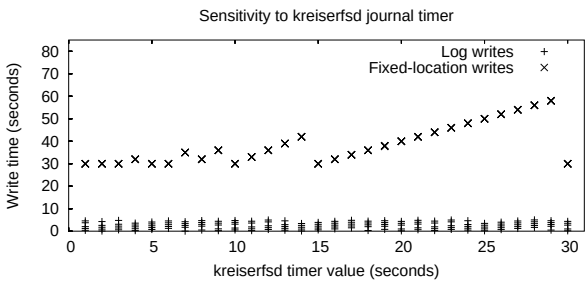


Figure 17: **Impact of Timers in ReiserFS.** *The figure plots the relationship between the time that data is written and the value of the kreiserfs timer. The scatter plot shows the results of multiple (30) runs. The process that is running writes 1 MB of data (no* `fsync`*); data journaling mode is used, with other timers set to 5 seconds and a journal size of 32 MB.*

4.2.3 Checkpoint Policy

We also investigate the conditions which trigger ReiserFS to checkpoint data to its fixed-place location; this policy is more complex in ReiserFS. In ext3, we found that data was checkpointed when the journal was $\frac{1}{4}$ to $\frac{1}{2}$ full. In ReiserFS, the point at which data is checkpointed depends not only on the free space in the journal, but also on the number of concurrent transactions. We again consider workloads that periodically force data to the journal by calling `fsync` at different intervals.

Our results are shown in Figure 16. The first graph shows the amount of data checkpointed as a function of the amount of data written; in all cases, data is checkpointed before $\frac{7}{8}$ of the journal is filled. The second graph shows the amount of data checkpointed as a function of the number of transactions. This graph shows that data is checkpointed at least at intervals of 128 transactions; running a similar workload on ext3 reveals no relationship between the number of transactions and checkpointing. Thus, ReiserFS checkpoints data whenever either journal free space drops below 4 MB or when there are 128 transactions in the journal.

As with ext3, timers control when data is written to the journal and to the fixed locations, but with some differences: in ext3, the kjournal daemon is responsible for committing transactions, whereas in ReiserFS, the kreiserfs daemon has this role. Figure 17 shows the time at which data is written to the journal and to the fixed location as the kreiserfs timer is increased; we make two conclusions. First, log writes always occur within the first five seconds of the data write by the application, regardless of the timer value. Second, the fixed-location writes occur only when the elapsed time is both greater than 30 seconds and a multiple of the kreiserfs timer value. Thus, the ReiserFS timer policy is simpler than that of ext3.

4.3 Finding Bugs

SBA analysis is useful not only for inferring the policies of filesystems, but also for finding cases that have not been implemented correctly. With SBA analysis, we have found a number of problems with the ReiserFS implementation that have not been reported elsewhere. In each case, we identified the problem because the SBA driver did not observe some disk traffic that it expected. To verify these problems, we have also examined the code to find the cause and have suggested corresponding fixes to the ReiserFS developers.

- In the first transaction after a mount, the `fsync` call returns before any of the data is written. We tracked this aberrant behavior to an incorrect initialization.
- When a file block is overwritten in writeback mode, its stat information is not updated. This error occurs due to a failure to update the inode's transaction information.
- When committing old transactions, dirty data is not always flushed. We tracked this to erroneously applying a condition to prevent data flushing during journal replay.
- Irrespective of changing the journal thread's wake up interval, dirty data is not flushed. This problem occurs due to a simple coding error.

5 The IBM Journaled File System

In this section, we describe our experience performing a preliminary SBA analysis of the Journaled File System (JFS). We began with a rudimentary understanding of JFS from what we were able to obtain through documentation [3]; for example, we knew that the journal is located by default at the end of the partition and is treated as a contiguous sequence of blocks and that one cannot specify the journaling mode.

Due to the fact that we knew less about this file system before we began, we found we needed to apply a new analysis technique as well: in some cases we filtered out traffic and then rebooted the system so that we could infer whether the filtered traffic was necessary for consistency or not. For example, we used this technique to understand the journaling mode of JFS. From this basic starting point, and without examining JFS code, we were able to learn a number of interesting properties about JFS.

First, we inferred that JFS uses ordered journaling mode. Due to the small amount of traffic to the journal, it was obvious that it was not employing data journaling. To differentiate between writeback and ordered modes, we observed that the ordering of writes matched that of ordered mode. That is, when a data block is written by the application, JFS orders the write such that the data block is written successfully before the metadata writes are issued.

Second, we determined that JFS does logging at the record level. That is, whenever an inode, index tree, or directory tree structure changes, only that structure is logged instead of the entire block containing the structure. As a result, JFS writes fewer journal blocks than ext3 and ReiserFS for the same operations.

Third, JFS does not by default group concurrent updates into a single compound transaction. Running the same experiment as we performed in Figure 6, we see that

the bandwidth of the asynchronous traffic is very high irrespective of whether there is a synchronous traffic in the background. However, there are circumstances in which transactions are grouped: for example, if the write commit records are on the same log page.

Finally, there are no commit timers in JFS and the fixed-location writes happen whenever the kupdate daemon's timer expires. However, the journal writes are never triggered by the timer: journal writes are indefinitely postponed until there is another trigger such as memory pressure or an unmount operation. This *infinite write delay* limits reliability, as a crash can result in data loss even for data that was written minutes or hours before.

6 Windows NTFS

In this section, we explain our analysis of NTFS. NTFS is a journaling file system that is used as the default file system on Windows operating systems such as XP, 2000, and NT. Although the source code or documentation of NTFS is not publicly available, tools for finding the NTFS file layout exist [28].

We ran the Windows XP operating system on top of VMware on a Linux machine. The pseudo device driver was exported as a SCSI disk to the Windows and a NTFS file system was constructed on top of the pseudo device. We ran simple workloads on NTFS and observed traffic within the SBA driver for our analysis.

Every object in NTFS is a file. Even metadata is stored in terms of files. The journal itself is a file and is located almost at the center of the file system. We used the ntfsprogs tools to discover journal file boundaries. Using the journal boundaries we were able to distinguish journal traffic from fixed-location traffic.

From our analysis, we found that NTFS does not do data journaling. This can be easily verified by the amount of data traffic observed by the SBA driver. We also found that NTFS, similar to JFS, does not do block-level journaling. It journals metadata in terms of records. We verified that whole blocks are not journaled in NTFS by matching the contents of the fixed-location traffic to the contents of the journal traffic.

We also inferred that NTFS performs ordered journaling. On data writes, NTFS waits until the data block writes to the fixed-location complete before writing the metadata blocks to the journal. We confirmed this ordering by using the SBA driver to delay the data block writes upto 10 seconds and found that the following metadata writes to the journal are delayed by the corresponding amount.

7 Related Work

Journaling Studies: Journaling file systems have been studied in detail. Most notably, Seltzer *et al.* [26] compare two variants of a journaling FFS to soft updates [11], a different technique for managing metadata consistency for file systems. Although the authors present no direct observation of low-level traffic, they are familiar enough with the systems (indeed, they are the implementors!) to explain behavior and make "semantic" inferences. For example, to explain why journaling performance drops in a delete benchmark, the authors report that the file system is "forced to read the first indirect block in order to reclaim the disk blocks it references" ([26], Section 8.1). A tool such as SBA makes such expert observations more readily available to all. Another recent study compares a range of Linux file systems, including ext2, ext3, ReiserFS, XFS, and JFS [7]. This work evaluates which file systems are fastest for different benchmarks, but gives little explanation as to *why* one does well for a given workload.

File System Benchmarks: There are many popular file system benchmarks, such as IOzone [19], Bonnie [6], lmbench [17], the modified Andrew benchmark [20], and PostMark [14]. Some of these (IOZone, Bonnie, lmbench) perform synthetic read/write tests to determine throughput; others (Andrew, Postmark) are intended to model "realistic" application workloads. Uniformly, all measure overall throughput or runtime to draw high-level conclusions about the file system. In contrast to SBA, none are intended to yield low-level insights about the internal policies of the file system.

Perhaps the most related to our work is Chen and Patterson's self-scaling benchmark [8]. In this work, the benchmarking framework conducts a search over the space of possible workload parameters (*e.g.*, sequentiality, request size, total workload size, and concurrency), and hones in on interesting parts of the workload space. Interestingly, some conclusions about file system behavior can be drawn from the resultant output, such as the size of the file cache. Our approach is not nearly as automated; instead, we construct benchmarks that exercise certain file system behaviors in a controlled manner.

File System Tracing: Many previous studies have traced file system activity. For example, Zhou *et al.* [37], Ousterhout *et al.* [21], Baker *et al.* [2], and Roselli *et al.* [24] all record various file system operations to later deduce file-level access patterns. Vogels [35] performs a similar study but inside the NT file system driver framework, where more information is available (*e.g.*, mapped I/O is not missed, as it is in most other studies). A recent example of a tracing infrastructure is TraceFS [1], which traces file systems at the VFS layer; however, TraceFS does not enable the low-level tracing that SBA provides. Finally, Blaze [5] and later Ellard *et al.* [10] show how low-level packet tracing can be useful in an NFS environment. By recording network-level protocol activity, network file system behavior can be carefully analyzed. This type of packet analysis is analogous to SBA since they are both positioned at a low level and thus must reconstruct higher-level behaviors to obtain a complete view.

8 Conclusions

As systems grow in complexity, there is a need for techniques and approaches that enable both users and system architects to understand in detail how such systems operate. We have presented semantic block-level analysis (SBA), a new methodology for file system benchmarking that uses block-level tracing to provide insight about the internal behavior of a file system. The block stream annotated with semantic information (*e.g.*, whether a block belongs to the journal or to another data structure) is an excellent source of information.

In this paper, we have focused on how the behavior of journaling file systems can be understood with SBA. In this case, using SBA is very straightforward: the user must know only how the journal is allocated on disk. Using SBA, we have analyzed in detail two Linux journaling file systems: ext3 and ReiserFS. We also have performed a preliminary analysis of Linux JFS and Windows NTFS. In all cases, we have uncovered behaviors that would be difficult to discover using more conventional approaches.

We have also developed and presented semantic trace playback (STP) which enables the rapid evaluation of new ideas for file systems. Using STP, we have demonstrated the potential benefits of numerous modifications to the current ext3 implementation for real workloads and traces. Of these modifications, we believe the transaction grouping mechanism within ext3 should most seriously be reevaluated; an untangled approach enables asynchronous processes to obtain in-memory bandwidth, despite the presence of other synchronous I/O streams in the system.

Acknowledgments

We thank Theodore Ts'o, Jiri Schindler and the members of the ADSL research group for their insightful comments. We also thank Mustafa Uysal for his excellent shepherding, and the anonymous reviewers for their thoughtful suggestions. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM and EMC.

References

[1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *FAST '04*, San Francisco, CA, April 2004.

[2] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *SOSP '91*, pages 198–212, Pacific Grove, CA, October 1991.

[3] S. Best. JFS Log. How the Journaled File System performs logging. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 163–168, Atlanta, 2000.

[4] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2004.

[5] M. Blaze. NFS tracing by passive network monitoring. In *USENIX Winter '92*, pages 333–344, San Francisco, CA, January 1992.

[6] T. Bray. The Bonnie File System Benchmark. http://www.textuality.com/bonnie/.

[7] R. Bryant, R. Forester, and J. Hawkes. Filesystem Performance and Scalability in Linux 2.4.17. In *FREENIX '02*, Monterey, CA, June 2002.

[8] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *SIGMETRICS '93*, pages 1–12, Santa Clara, CA, May 1993.

[9] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode File System. In *USENIX Winter '92*, pages 43–60, San Francisco, CA, January 1992.

[10] D. Ellard and M. I. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *LISA '03*, pages 73–85, San Diego, California, October 2003.

[11] G. R. Ganger and Y. N. Patt. Metadata Update Performance in File Systems. In *OSDI '94*, pages 49–60, Monterey, CA, November 1994.

[12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, Austin, Texas, November 1987.

[14] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[15] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[16] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[17] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX 1996*, San Diego, CA, January 1996.

[18] J. C. Mogul. A Better Update Policy. In *USENIX Summer '94*, Boston, MA, June 1994.

[19] W. Norcutt. The IOzone Filesystem Benchmark. http://www.iozone.org/.

[20] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, CA, June 1990.

[21] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *SOSP '85*, pages 15–24, Orcas Island, WA, December 1985.

[22] H. Reiser. ReiserFS. www.namesys.com, 2004.

[23] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *FAST '02*, pages 14–29, Monterey, CA, January 2002.

[24] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *USENIX '00*, pages 41–54, San Diego, California, June 2000.

[25] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[26] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *USENIX '00*, pages 71–84, San Diego, California, June 2000.

[27] D. A. Solomon. *Inside Windows NT (Microsoft Programming Series)*. Microsoft Press, 1998.

[28] SourceForge. The Linux NTFS Project. http://linux-ntfs.sf.net/, 2004.

[29] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *USENIX 1996*, San Diego, CA, January 1996.

[30] Transaction Processing Council. TPC Benchmark B Standard Specification, Revision 3.2. Technical Report, 1990.

[31] Transaction Processing Council. TPC Benchmark C Standard Specification, Revision 5.2. Technical Report, 1992.

[32] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX '02*, Monterey, CA, June 2002.

[33] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[34] S. C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[35] W. Vogels. File system usage in Windows NT 4.0. In *SOSP '99*, pages 93–109, Kiawah Island Resort, SC, December 1999.

[36] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.

[37] S. Zhou, H. D. Costa, and A. Smith. A File System Tracing Package for Berkeley UNIX. In *USENIX Summer '84*, pages 407–419, Salt Lake City, UT, June 1984.

The Design and Implementation of a Log-Structured File System

Mendel Rosenblum and John K. Ousterhout

Electrical Engineering and Computer Sciences, Computer Science Division
University of California
Berkeley, CA 94720
mendel@sprite.berkeley.edu, ouster@sprite.berkeley.edu

Abstract

This paper presents a new technique for disk storage management called a *log-structured file system*. A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. The log is the only structure on disk; it contains indexing information so that files can be read back from the log efficiently. In order to maintain large free areas on disk for fast writing, we divide the log into *segments* and use a *segment cleaner* to compress the live information from heavily fragmented segments. We present a series of simulations that demonstrate the efficiency of a simple cleaning policy based on cost and benefit. We have implemented a prototype log-structured file system called Sprite LFS; it outperforms current Unix file systems by an order of magnitude for small-file writes while matching or exceeding Unix performance for reads and large writes. Even when the overhead for cleaning is included, Sprite LFS can use 70% of the disk bandwidth for writing, whereas Unix file systems typically can use only 5-10%.

1. Introduction

Over the last decade CPU speeds have increased dramatically while disk access times have only improved slowly. This trend is likely to continue in the future and it will cause more and more applications to become disk-bound. To lessen the impact of this problem, we have devised a new disk storage management technique called a *log-structured file system*, which uses disks an order of magnitude more efficiently than current file systems.

Log-structured file systems are based on the assumption that files are cached in main memory and that increasing memory sizes will make the caches more and more effective at satisfying read requests[1]. As a result, disk traffic will become dominated by writes. A log-structured file system writes all new information to disk in a sequential structure called the *log*. This approach increases write performance dramatically by eliminating almost all seeks. The sequential nature of the log also permits much faster crash recovery: current Unix file systems typically must scan the entire disk to restore consistency after a crash, but a log-structured file system need only examine the most recent portion of the log.

The notion of logging is not new, and a number of recent file systems have incorporated a log as an auxiliary structure to speed up writes and crash recovery[2, 3]. However, these other systems use the log only for temporary storage; the permanent home for information is in a traditional random-access storage structure on disk. In contrast, a log-structured file system stores data permanently in the log: there is no other structure on disk. The log contains indexing information so that files can be read back with efficiency comparable to current file systems.

For a log-structured file system to operate efficiently, it must ensure that there are always large extents of free space available for writing new data. This is the most difficult challenge in the design of a log-structured file system. In this paper we present a solution based on large extents called *segments*, where a *segment cleaner* process continually regenerates empty segments by compressing the live data from heavily fragmented segments. We used a simulator to explore different cleaning policies and discovered a simple but effective algorithm based on cost and benefit: it segregates older, more slowly changing data from young rapidly-changing data and treats them differently during cleaning.

We have constructed a prototype log-structured file system called Sprite LFS, which is now in production use as part of the Sprite network operating system[4]. Benchmark programs demonstrate that the raw writing speed of Sprite LFS is more than an order of magnitude greater than that of Unix for small files. Even for other workloads, such

The work described here was supported in part by the National Science Foundation under grant CCR-8900029, and in part by the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591.

This paper will appear in the *Proceedings of the 13th ACM Symposium on Operating Systems Principles* and the February 1992 *ACM Transactions on Computer Systems*.

as those including reads and large-file accesses, Sprite LFS is at least as fast as Unix in all cases but one (files read sequentially after being written randomly). We also measured the long-term overhead for cleaning in the production system. Overall, Sprite LFS permits about 65-75% of a disk's raw bandwidth to be used for writing new data (the rest is used for cleaning). For comparison, Unix systems can only utilize 5-10% of a disk's raw bandwidth for writing new data; the rest of the time is spent seeking.

The remainder of this paper is organized into six sections. Section 2 reviews the issues in designing file systems for computers of the 1990's. Section 3 discusses the design alternatives for a log-structured file system and derives the structure of Sprite LFS, with particular focus on the cleaning mechanism. Section 4 describes the crash recovery system for Sprite LFS. Section 5 evaluates Sprite LFS using benchmark programs and long-term measurements of cleaning overhead. Section 6 compares Sprite LFS to other file systems, and Section 7 concludes.

2. Design for file systems of the 1990's

File system design is governed by two general forces: technology, which provides a set of basic building blocks, and workload, which determines a set of operations that must be carried out efficiently. This section summarizes technology changes that are underway and describes their impact on file system design. It also describes the workloads that influenced the design of Sprite LFS and shows how current file systems are ill-equipped to deal with the workloads and technology changes.

2.1. Technology

Three components of technology are particularly significant for file system design: processors, disks, and main memory. Processors are significant because their speed is increasing at a nearly exponential rate, and the improvements seem likely to continue through much of the 1990's. This puts pressure on all the other elements of the computer system to speed up as well, so that the system doesn't become unbalanced.

Disk technology is also improving rapidly, but the improvements have been primarily in the areas of cost and capacity rather than performance. There are two components of disk performance: transfer bandwidth and access time. Although both of these factors are improving, the rate of improvement is much slower than for CPU speed. Disk transfer bandwidth can be improved substantially with the use of disk arrays and parallel-head disks[5] but no major improvements seem likely for access time (it is determined by mechanical motions that are hard to improve). If an application causes a sequence of small disk transfers separated by seeks, then the application is not likely to experience much speedup over the next ten years, even with faster processors.

The third component of technology is main memory, which is increasing in size at an exponential rate. Modern file systems cache recently-used file data in main memory, and larger main memories make larger file caches possible. This has two effects on file system behavior. First, larger file caches alter the workload presented to the disk by absorbing a greater fraction of the read requests[1, 6]. Most write requests must eventually be reflected on disk for safety, so disk traffic (and disk performance) will become more and more dominated by writes.

The second impact of large file caches is that they can serve as write buffers where large numbers of modified blocks can be collected before writing any of them to disk. Buffering may make it possible to write the blocks more efficiently, for example by writing them all in a single sequential transfer with only one seek. Of course, write-buffering has the disadvantage of increasing the amount of data lost during a crash. For this paper we will assume that crashes are infrequent and that it is acceptable to lose a few seconds or minutes of work in each crash; for applications that require better crash recovery, non-volatile RAM may be used for the write buffer.

2.2. Workloads

Several different file system workloads are common in computer applications. One of the most difficult workloads for file system designs to handle efficiently is found in office and engineering environments. Office and engineering applications tend to be dominated by accesses to small files; several studies have measured mean file sizes of only a few kilobytes[1, 6-8]. Small files usually result in small random disk I/Os, and the creation and deletion times for such files are often dominated by updates to file system ''metadata'' (the data structures used to locate the attributes and blocks of the file).

Workloads dominated by sequential accesses to large files, such as those found in supercomputing environments, also pose interesting problems, but not for file system software. A number of techniques exist for ensuring that such files are laid out sequentially on disk, so I/O performance tends to be limited by the bandwidth of the I/O and memory subsystems rather than the file allocation policies. In designing a log-structured file system we decided to focus on the efficiency of small-file accesses, and leave it to hardware designers to improve bandwidth for large-file accesses. Fortunately, the techniques used in Sprite LFS work well for large files as well as small ones.

2.3. Problems with existing file systems

Current file systems suffer from two general problems that make it hard for them to cope with the technologies and workloads of the 1990's. First, they spread information around the disk in a way that causes too many small accesses. For example, the Berkeley Unix fast file system (Unix FFS)[9] is quite effective at laying out each file sequentially on disk, but it physically separates different files. Furthermore, the attributes (''inode'') for a file are separate from the file's contents, as is the directory entry containing the file's name. It takes at least five separate disk I/Os, each preceded by a seek, to create a new file in

Unix FFS: two different accesses to the file's attributes plus one access each for the file's data, the directory's data, and the directory's attributes. When writing small files in such a system, less than 5% of the disk's potential bandwidth is used for new data; the rest of the time is spent seeking.

The second problem with current file systems is that they tend to write synchronously: the application must wait for the write to complete, rather than continuing while the write is handled in the background. For example even though Unix FFS writes file data blocks asynchronously, file system metadata structures such as directories and inodes are written synchronously. For workloads with many small files, the disk traffic is dominated by the synchronous metadata writes. Synchronous writes couple the application's performance to that of the disk and make it hard for the application to benefit from faster CPUs. They also defeat the potential use of the file cache as a write buffer. Unfortunately, network file systems like NFS[10] have introduced additional synchronous behavior where it didn't used to exist. This has simplified crash recovery, but it has reduced write performance.

Throughout this paper we use the Berkeley Unix fast file system (Unix FFS) as an example of current file system design and compare it to log-structured file systems. The Unix FFS design is used because it is well documented in the literature and used in several popular Unix operating systems. The problems presented in this section are not unique to Unix FFS and can be found in most other file systems.

3. Log-structured file systems

The fundamental idea of a log-structured file system is to improve write performance by buffering a sequence of file system changes in the file cache and then writing all the changes to disk sequentially in a single disk write operation. The information written to disk in the write operation includes file data blocks, attributes, index blocks, directories, and almost all the other information used to manage the file system. For workloads that contain many small files, a log-structured file system converts the many small synchronous random writes of traditional file systems into large asynchronous sequential transfers that can utilize nearly 100% of the raw disk bandwidth.

Although the basic idea of a log-structured file system is simple, there are two key issues that must be resolved to achieve the potential benefits of the logging approach. The first issue is how to retrieve information from the log; this is the subject of Section 3.1 below. The second issue is how to manage the free space on disk so that large extents of free space are always available for writing new data. This is a much more difficult issue; it is the topic of Sections 3.2-3.6. Table 1 contains a summary of the on-disk data structures used by Sprite LFS to solve the above problems; the data structures are discussed in detail in later sections of the paper.

3.1. File location and reading

Although the term ''log-structured'' might suggest that sequential scans are required to retrieve information from the log, this is not the case in Sprite LFS. Our goal was to match or exceed the read performance of Unix FFS. To accomplish this goal, Sprite LFS outputs index structures in the log to permit random-access retrievals. The basic structures used by Sprite LFS are identical to those used in Unix FFS: for each file there exists a data structure called an *inode*, which contains the file's attributes (type, owner, permissions, etc.) plus the disk addresses of the first ten blocks of the file; for files larger than ten blocks, the inode also contains the disk addresses of one or more *indirect blocks*, each of which contains the addresses of more data or indirect blocks. Once a file's inode has been found, the number of disk I/Os required to read the file is identical in Sprite LFS and Unix FFS.

In Unix FFS each inode is at a fixed location on disk; given the identifying number for a file, a simple calculation

Data structure	Purpose	Location	Section
Inode	Locates blocks of file, holds protection bits, modify time, etc.	Log	3.1
Inode map	Locates position of inode in log, holds time of last access plus version number.	Log	3.1
Indirect block	Locates blocks of large files.	Log	3.1
Segment summary	Identifies contents of segment (file number and offset for each block).	Log	3.2
Segment usage table	Counts live bytes still left in segments, stores last write time for data in segments.	Log	3.6
Superblock	Holds static configuration information such as number of segments and segment size.	Fixed	None
Checkpoint region	Locates blocks of inode map and segment usage table, identifies last checkpoint in log.	Fixed	4.1
Directory change log	Records directory operations to maintain consistency of reference counts in inodes.	Log	4.2

Table 1 — Summary of the major data structures stored on disk by Sprite LFS.
For each data structure the table indicates the purpose served by the data structure in Sprite LFS. The table also indicates whether the data structure is stored in the log or at a fixed position on disk and where in the paper the data structure is discussed in detail. Inodes, indirect blocks, and superblocks are similar to the Unix FFS data structures with the same names. Note that Sprite LFS contains neither a bitmap nor a free list.

yields the disk address of the file's inode. In contrast, Sprite LFS doesn't place inodes at fixed positions; they are written to the log. Sprite LFS uses a data structure called an *inode map* to maintain the current location of each inode. Given the identifying number for a file, the inode map must be indexed to determine the disk address of the inode. The inode map is divided into blocks that are written to the log; a fixed checkpoint region on each disk identifies the locations of all the inode map blocks. Fortunately, inode maps are compact enough to keep the active portions cached in main memory: inode map lookups rarely require disk accesses.

Figure 1 shows the disk layouts that would occur in Sprite LFS and Unix FFS after creating two new files in different directories. Although the two layouts have the same logical structure, the log-structured file system produces a much more compact arrangement. As a result, the write performance of Sprite LFS is much better than Unix FFS, while its read performance is just as good.

3.2. Free space management: segments

The most difficult design issue for log-structured file systems is the management of free space. The goal is to maintain large free extents for writing new data. Initially all the free space is in a single extent on disk, but by the time the log reaches the end of the disk the free space will have been fragmented into many small extents corresponding to the files that were deleted or overwritten.

From this point on, the file system has two choices: threading and copying. These are illustrated in Figure 2. The first alternative is to leave the live data in place and thread the log through the free extents. Unfortunately, threading will cause the free space to become severely fragmented, so that large contiguous writes won't be possible and a log-structured file system will be no faster than traditional file systems. The second alternative is to copy live data out of the log in order to leave large free extents for writing. For this paper we will assume that the live data is written back in a compacted form at the head of the log; it could also be moved to another log-structured file system to form a hierarchy of logs, or it could be moved to some totally different file system or archive. The disadvantage of copying is its cost, particularly for long-lived files; in the simplest case where the log works circularly across the disk and live data is copied back into the log, all of the long-lived files will have to be copied in every pass of the log across the disk.

Sprite LFS uses a combination of threading and copying. The disk is divided into large fixed-size extents called *segments*. Any given segment is always written sequentially from its beginning to its end, and all live data must be copied out of a segment before the segment can be rewritten. However, the log is threaded on a segment-by-segment basis; if the system can collect long-lived data together into segments, those segments can be skipped over so that the data doesn't have to be copied repeatedly. The segment size is chosen large enough that the transfer time to read or write a whole segment is much greater than the cost of a seek to the beginning of the segment. This allows whole-segment operations to run at nearly the full bandwidth of the disk, regardless of the order in which segments are accessed. Sprite LFS currently uses segment sizes of either 512 kilobytes or one megabyte.

3.3. Segment cleaning mechanism

The process of copying live data out of a segment is called *segment cleaning*. In Sprite LFS it is a simple three-step process: read a number of segments into memory, identify the live data, and write the live data back to a smaller number of clean segments. After this

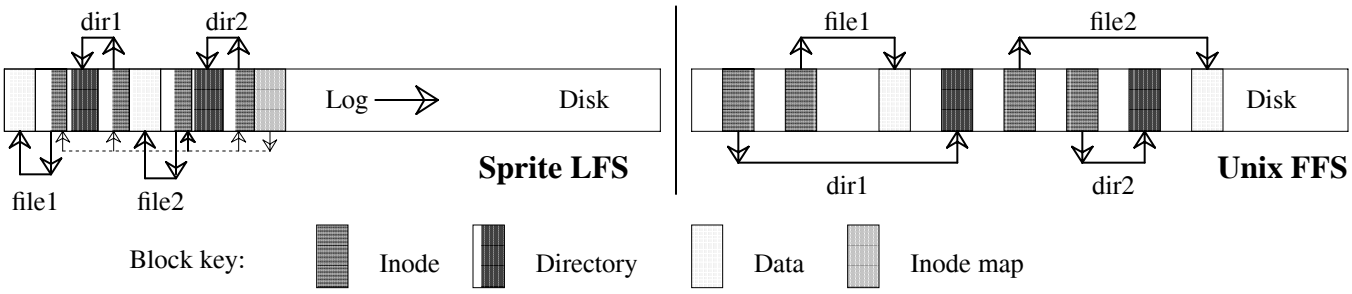


Figure 1 — A comparison between Sprite LFS and Unix FFS.
This example shows the modified disk blocks written by Sprite LFS and Unix FFS when creating two single-block files named `dir1/file1` and `dir2/file2`. Each system must write new data blocks and inodes for `file1` and `file2`, plus new data blocks and inodes for the containing directories. Unix FFS requires ten non-sequential writes for the new information (the inodes for the new files are each written twice to ease recovery from crashes), while Sprite LFS performs the operations in a single large write. The same number of disk accesses will be required to read the files in the two systems. Sprite LFS also writes out new inode map blocks to record the new inode locations.

operation is complete, the segments that were read are marked as clean, and they can be used for new data or for additional cleaning.

As part of segment cleaning it must be possible to identify which blocks of each segment are live, so that they can be written out again. It must also be possible to identify the file to which each block belongs and the position of the block within the file; this information is needed in order to update the file's inode to point to the new location of the block. Sprite LFS solves both of these problems by writing a *segment summary block* as part of each segment. The summary block identifies each piece of information that is written in the segment; for example, for each file data block the summary block contains the file number and block number for the block. Segments can contain multiple segment summary blocks when more than one log write is needed to fill the segment. (Partial-segment writes occur when the number of dirty blocks buffered in the file cache is insufficient to fill a segment.) Segment summary blocks impose little overhead during writing, and they are useful during crash recovery (see Section 4) as well as during cleaning.

Sprite LFS also uses the segment summary information to distinguish live blocks from those that have been overwritten or deleted. Once a block's identity is known, its liveness can be determined by checking the file's inode or indirect block to see if the appropriate block pointer still refers to this block. If it does, then the block is live; if it doesn't, then the block is dead. Sprite LFS optimizes this check slightly by keeping a version number in the inode map entry for each file; the version number is incremented whenever the file is deleted or truncated to length zero. The version number combined with the inode number form an unique identifier (uid) for the contents of the file. The segment summary block records this uid for each block in the segment; if the uid of a block does not match the uid currently stored in the inode map when the segment is cleaned, the block can be discarded immediately without examining the file's inode.

This approach to cleaning means that there is no free-block list or bitmap in Sprite. In addition to saving memory and disk space, the elimination of these data structures also simplifies crash recovery. If these data structures existed, additional code would be needed to log changes to the structures and restore consistency after crashes.

3.4. Segment cleaning policies

Given the basic mechanism described above, four policy issues must be addressed:

(1) When should the segment cleaner execute? Some possible choices are for it to run continuously in background at low priority, or only at night, or only when disk space is nearly exhausted.

(2) How many segments should it clean at a time? Segment cleaning offers an opportunity to reorganize data on disk; the more segments cleaned at once, the more opportunities to rearrange.

(3) Which segments should be cleaned? An obvious choice is the ones that are most fragmented, but this turns out not to be the best choice.

(4) How should the live blocks be grouped when they are written out? One possibility is to try to enhance the locality of future reads, for example by grouping files in the same directory together into a single output segment. Another possibility is to sort the blocks by the time they were last modified and group blocks of similar age together into new segments; we call this approach *age sort*.

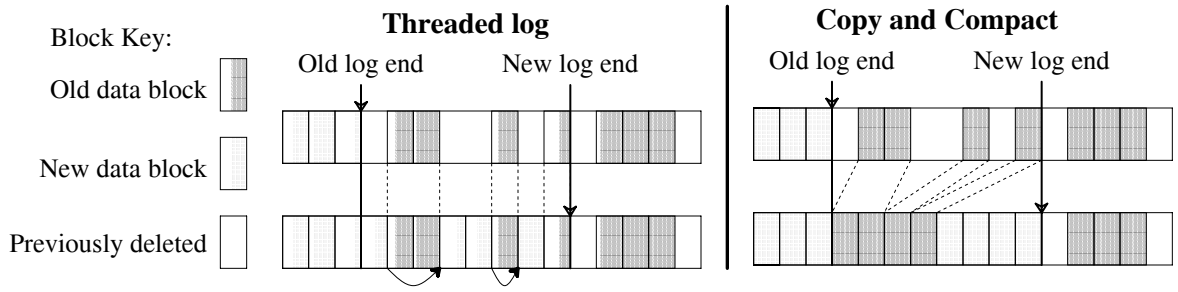


Figure 2 — Possible free space management solutions for log-structured file systems.
In a log-structured file system, free space for the log can be generated either by copying the old blocks or by threading the log around the old blocks. The left side of the figure shows the threaded log approach where the log skips over the active blocks and overwrites blocks of files that have been deleted or overwritten. Pointers between the blocks of the log are maintained so that the log can be followed during crash recovery. The right side of the figure shows the copying scheme where log space is generated by reading the section of disk after the end of the log and rewriting the active blocks of that section along with the new data into the newly generated space.

In our work so far we have not methodically addressed the first two of the above policies. Sprite LFS starts cleaning segments when the number of clean segments drops below a threshold value (typically a few tens of segments). It cleans a few tens of segments at a time until the number of clean segments surpasses another threshold value (typically 50-100 clean segments). The overall performance of Sprite LFS does not seem to be very sensitive to the exact choice of the threshold values. In contrast, the third and fourth policy decisions are critically important: in our experience they are the primary factors that determine the performance of a log-structured file system. The remainder of Section 3 discusses our analysis of which segments to clean and how to group the live data.

We use a term called *write cost* to compare cleaning policies. The write cost is the average amount of time the disk is busy per byte of new data written, including all the cleaning overheads. The write cost is expressed as a multiple of the time that would be required if there were no cleaning overhead and the data could be written at its full bandwidth with no seek time or rotational latency. A write cost of 1.0 is perfect: it would mean that new data could be written at the full disk bandwidth and there is no cleaning overhead. A write cost of 10 means that only one-tenth of the disk's maximum bandwidth is actually used for writing new data; the rest of the disk time is spent in seeks, rotational latency, or cleaning.

For a log-structured file system with large segments, seeks and rotational latency are negligible both for writing and for cleaning, so the write cost is the total number of bytes moved to and from the disk divided by the number of those bytes that represent new data. This cost is determined by the utilization (the fraction of data still live) in the segments that are cleaned. In the steady state, the cleaner must generate one clean segment for every segment of new data written. To do this, it reads N segments in their entirety and writes out N*u segments of live data (where u is the utilization of the segments and $0 \le u < 1$). This creates N*(1−u) segments of contiguous free space for new data. Thus

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}}$$

$$= \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}}$$

$$= \frac{N + N^*u + N^*(1-u)}{N^*(1-u)} = \frac{2}{1-u} \qquad (1)$$

In the above formula we made the conservative assumption that a segment must be read in its entirety to recover the live blocks; in practice it may be faster to read just the live blocks, particularly if the utilization is very low (we haven't tried this in Sprite LFS). If a segment to be cleaned has no live blocks ($u = 0$) then it need not be read at all and the write cost is 1.0.

Figure 3 graphs the write cost as a function of u. For reference, Unix FFS on small-file workloads utilizes at most 5-10% of the disk bandwidth, for a write cost of 10-20 (see [11] and Figure 8 in Section 5.1 for specific measurements). With logging, delayed writes, and disk request sorting this can probably be improved to about 25% of the bandwidth[12] or a write cost of 4. Figure 3 suggests that the segments cleaned must have a utilization of less than .8 in order for a log-structured file system to outperform the current Unix FFS; the utilization must be less than .5 to outperform an improved Unix FFS.

It is important to note that the utilization discussed above is not the overall fraction of the disk containing live data; it is just the fraction of live blocks in segments that are cleaned. Variations in file usage will cause some segments to be less utilized than others, and the cleaner can choose the least utilized segments to clean; these will have lower utilization than the overall average for the disk.

Even so, the performance of a log-structured file system can be improved by reducing the overall utilization of the disk space. With less of the disk in use the segments that are cleaned will have fewer live blocks resulting in a lower write cost. Log-structured file systems provide a cost-performance tradeoff: if disk space is underutilized, higher performance can be achieved but at a high cost per usable byte; if disk capacity utilization is increased, storage costs are reduced but so is performance. Such a tradeoff

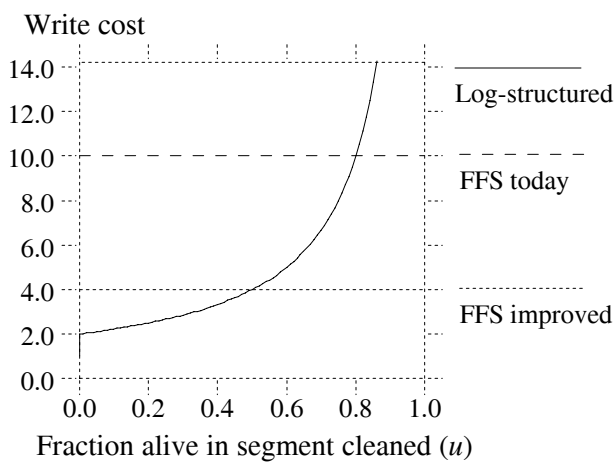


Figure 3 — Write cost as a function of u for small files.
In a log-structured file system, the write cost depends strongly on the utilization of the segments that are cleaned. The more live data in segments cleaned the more disk bandwidth that is needed for cleaning and not available for writing new data. The figure also shows two reference points: ''FFS today'', which represents Unix FFS today, and ''FFS improved'', which is our estimate of the best performance possible in an improved Unix FFS. Write cost for Unix FFS is not sensitive to the amount of disk space in use.

between performance and space utilization is not unique to log-structured file systems. For example, Unix FFS only allows 90% of the disk space to be occupied by files. The remaining 10% is kept free to allow the space allocation algorithm to operate efficiently.

The key to achieving high performance at low cost in a log-structured file system is to force the disk into a bimodal segment distribution where most of the segments are nearly full, a few are empty or nearly empty, and the cleaner can almost always work with the empty segments. This allows a high overall disk capacity utilization yet provides a low write cost. The following section describes how we achieve such a bimodal distribution in Sprite LFS.

3.5. Simulation results

We built a simple file system simulator so that we could analyze different cleaning policies under controlled conditions. The simulator's model does not reflect actual file system usage patterns (its model is much harsher than reality), but it helped us to understand the effects of random access patterns and locality, both of which can be exploited to reduce the cost of cleaning. The simulator models a file system as a fixed number of 4-kbyte files, with the number chosen to produce a particular overall disk capacity utilization. At each step, the simulator overwrites one of the files with new data, using one of two pseudo-random access patterns:

Uniform — Each file has equal likelihood of being selected in each step.

Hot-and-cold — Files are divided into two groups. One group contains 10% of the files; it is called *hot* because its files are selected 90% of the time. The other group is called *cold*; it contains 90% of the files but they are selected only 10% of the time. Within groups each file is equally likely to be selected. This access pattern models a simple form of locality.

In this approach the overall disk capacity utilization is constant and no read traffic is modeled. The simulator runs until all clean segments are exhausted, then simulates the actions of a cleaner until a threshold number of clean segments is available again. In each run the simulator was allowed to run until the write cost stabilized and all cold-start variance had been removed.

Figure 4 superimposes the results from two sets of simulations onto the curves of Figure 3. In the ''LFS uniform'' simulations the uniform access pattern was used. The cleaner used a simple greedy policy where it always chose the least-utilized segments to clean. When writing out live data the cleaner did not attempt to re-organize the data: live blocks were written out in the same order that they appeared in the segments being cleaned (for a uniform access pattern there is no reason to expect any improvement from re-organization).

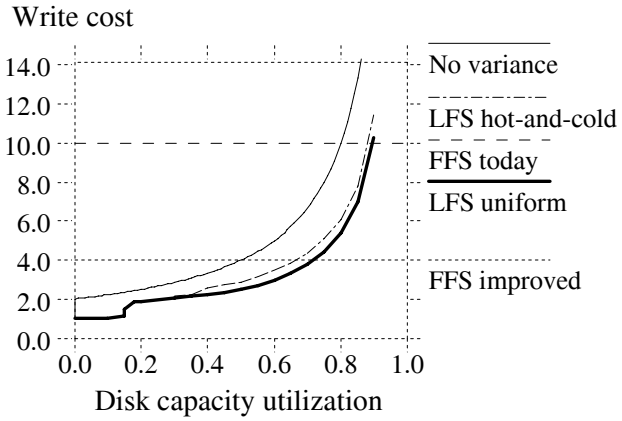


Figure 4 — Initial simulation results.
The curves labeled ''FFS today'' and ''FFS improved'' are reproduced from Figure 3 for comparison. The curve labeled ''No variance'' shows the write cost that would occur if all segments always had exactly the same utilization. The ''LFS uniform'' curve represents a log-structured file system with uniform access pattern and a greedy cleaning policy: the cleaner chooses the least-utilized segments. The ''LFS hot-and-cold'' curve represents a log-structured file system with locality of file access. It uses a greedy cleaning policy and the cleaner also sorts the live data by age before writing it out again. The x-axis is overall disk capacity utilization, which is not necessarily the same as the utilization of the segments being cleaned.

Even with uniform random access patterns, the variance in segment utilization allows a substantially lower write cost than would be predicted from the overall disk capacity utilization and formula (1). For example, at 75% overall disk capacity utilization, the segments cleaned have an average utilization of only 55%. At overall disk capacity utilizations under 20% the write cost drops below 2.0; this means that some of the cleaned segments have no live blocks at all and hence don't need to be read in.

The ''LFS hot-and-cold'' curve shows the write cost when there is locality in the access patterns, as described above. The cleaning policy for this curve was the same as for ''LFS uniform'' except that the live blocks were sorted by age before writing them out again. This means that long-lived (cold) data tends to be segregated in different segments from short-lived (hot) data; we thought that this approach would lead to the desired bimodal distribution of segment utilizations.

Figure 4 shows the surprising result that locality and ''better'' grouping result in *worse* performance than a system with no locality! We tried varying the degree of locality (e.g. 95% of accesses to 5% of data) and found that performance got worse and worse as the locality increased. Figure 5 shows the reason for this non-intuitive result. Under the greedy policy, a segment doesn't get cleaned until it becomes the least utilized of all segments. Thus every segment's utilization eventually drops to the cleaning threshold, including the cold segments. Unfortunately, the

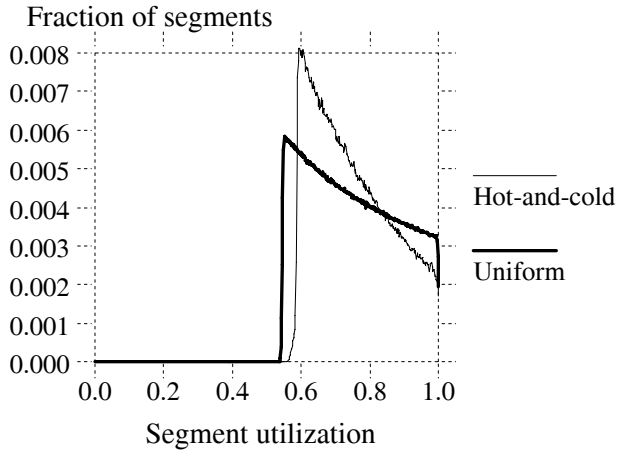


Figure 5 — Segment utilization distributions with greedy cleaner.
These figures show distributions of segment utilizations of the disk during the simulation. The distribution is computed by measuring the utilizations of all segments on the disk at the points during the simulation when segment cleaning was initiated. The distribution shows the utilizations of the segments available to the cleaning algorithm. Each of the distributions corresponds to an overall disk capacity utilization of 75%. The ''Uniform'' curve corresponds to ''LFS uniform'' in Figure 4 and ''Hot-and-cold'' corresponds to ''LFS hot-and-cold'' in Figure 4. Locality causes the distribution to be more skewed towards the utilization at which cleaning occurs; as a result, segments are cleaned at a higher average utilization.

utilization drops very slowly in cold segments, so these segments tend to linger just above the cleaning point for a very long time. Figure 5 shows that many more segments are clustered around the cleaning point in the simulations with locality than in the simulations without locality. The overall result is that cold segments tend to tie up large numbers of free blocks for long periods of time.

After studying these figures we realized that hot and cold segments must be treated differently by the cleaner. Free space in a cold segment is more valuable than free space in a hot segment because once a cold segment has been cleaned it will take a long time before it re-accumulates the unusable free space. Said another way, once the system reclaims the free blocks from a segment with cold data it will get to ''keep'' them a long time before the cold data becomes fragmented and ''takes them back again.'' In contrast, it is less beneficial to clean a hot segment because the data will likely die quickly and the free space will rapidly re-accumulate; the system might as well delay the cleaning a while and let more of the blocks die in the current segment. The value of a segment's free space is based on the stability of the data in the segment. Unfortunately, the stability cannot be predicted without knowing future access patterns. Using an assumption that the older the data in a segment the longer it is likely to remain unchanged, the stability can be estimated by the age of data.

To test this theory we simulated a new policy for selecting segments to clean. The policy rates each segment according to the benefit of cleaning the segment and the cost of cleaning the segment and chooses the segments with the highest ratio of benefit to cost. The benefit has two components: the amount of free space that will be reclaimed and the amount of time the space is likely to stay free. The amount of free space is just $1-u$, where u is the utilization of the segment. We used the most recent modified time of any block in the segment (ie. the age of the youngest block) as an estimate of how long the space is likely to stay free. The benefit of cleaning is the space-time product formed by multiplying these two components. The cost of cleaning the segment is $1+u$ (one unit of cost to read the segment, u to write back the live data). Combining all these factors, we get

$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated * age of data}}{\text{cost}} = \frac{(1-u)*\text{age}}{1+u}$$

We call this policy the *cost-benefit* policy; it allows cold segments to be cleaned at a much higher utilization than hot segments.

We re-ran the simulations under the hot-and-cold access pattern with the cost-benefit policy and age-sorting

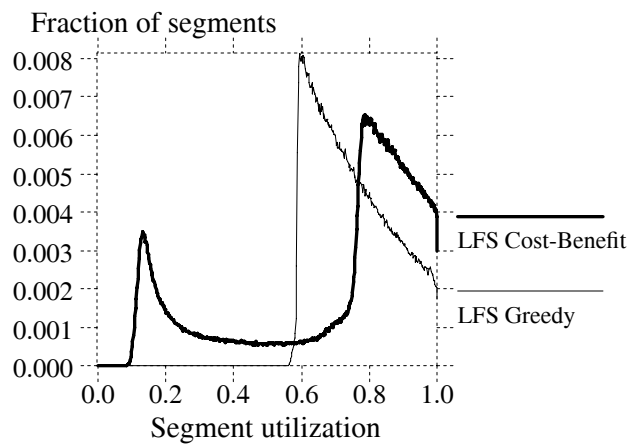


Figure 6 — Segment utilization distribution with cost-benefit policy.
This figure shows the distribution of segment utilizations from the simulation of a hot-and-cold access pattern with 75% overall disk capacity utilization. The ''LFS Cost-Benefit'' curve shows the segment distribution occurring when the cost-benefit policy is used to select segments to clean and live blocks grouped by age before being re-written. Because of this bimodal segment distribution, most of the segments cleaned had utilizations around 15%. For comparison, the distribution produced by the greedy method selection policy is shown by the ''LFS Greedy'' curve reproduced from Figure 5.

on the live data. As can be seen from Figure 6, the cost-benefit policy produced the bimodal distribution of segments that we had hoped for. The cleaning policy cleans cold segments at about 75% utilization but waits until hot segments reach a utilization of about 15% before cleaning them. Since 90% of the writes are to hot files, most of the segments cleaned are hot. Figure 7 shows that the cost-benefit policy reduces the write cost by as much as 50% over the greedy policy, and a log-structured file system out-performs the best possible Unix FFS even at relatively high disk capacity utilizations. We simulated a number of other degrees and kinds of locality and found that the cost-benefit policy gets even better as locality increases.

The simulation experiments convinced us to implement the cost-benefit approach in Sprite LFS. As will be seen in Section 5.2, the behavior of actual file systems in Sprite LFS is even better than predicted in Figure 7.

3.6. Segment usage table

In order to support the cost-benefit cleaning policy, Sprite LFS maintains a data structure called the *segment usage table*. For each segment, the table records the number of live bytes in the segment and the most recent modified time of any block in the segment. These two values are used by the segment cleaner when choosing segments to clean. The values are initially set when the segment is written, and the count of live bytes is decremented when files are deleted or blocks are overwritten. If the count falls to zero then the segment can be reused without cleaning. The blocks of the segment usage table are written to the log, and the addresses of the blocks are stored in the checkpoint regions (see Section 4 for details).

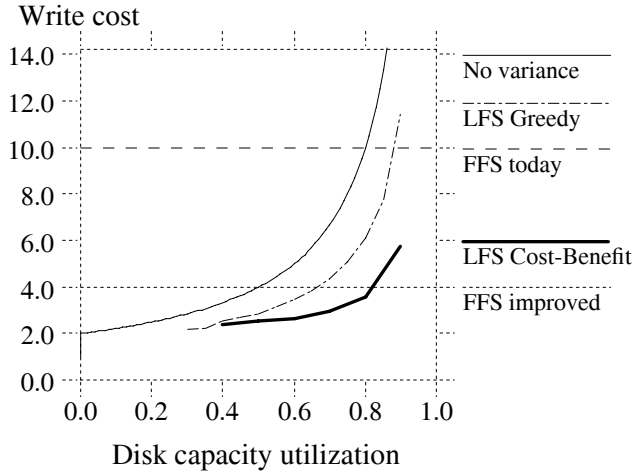


Figure 7 — Write cost, including cost-benefit policy.
This graph compares the write cost of the greedy policy with that of the cost-benefit policy for the hot-and-cold access pattern. The cost-benefit policy is substantially better than the greedy policy, particularly for disk capacity utilizations above 60%.

In order to sort live blocks by age, the segment summary information records the age of the youngest block written to the segment. At present Sprite LFS does not keep modified times for each block in a file; it keeps a single modified time for the entire file. This estimate will be incorrect for files that are not modified in their entirety. We plan to modify the segment summary information to include modified times for each block.

4. Crash recovery

When a system crash occurs, the last few operations performed on the disk may have left it in an inconsistent state (for example, a new file may have been written without writing the directory containing the file); during reboot the operating system must review these operations in order to correct any inconsistencies. In traditional Unix file systems without logs, the system cannot determine where the last changes were made, so it must scan all of the metadata structures on disk to restore consistency. The cost of these scans is already high (tens of minutes in typical configurations), and it is getting higher as storage systems expand.

In a log-structured file system the locations of the last disk operations are easy to determine: they are at the end of the log. Thus it should be possible to recover very quickly after crashes. This benefit of logs is well known and has been used to advantage both in database systems[13] and in other file systems[2, 3, 14]. Like many other logging systems, Sprite LFS uses a two-pronged approach to recovery: *checkpoints*, which define consistent states of the file system, and *roll-forward*, which is used to recover information written since the last checkpoint.

4.1. Checkpoints

A checkpoint is a position in the log at which all of the file system structures are consistent and complete. Sprite LFS uses a two-phase process to create a checkpoint. First, it writes out all modified information to the log, including file data blocks, indirect blocks, inodes, and blocks of the inode map and segment usage table. Second, it writes a *checkpoint region* to a special fixed position on disk. The checkpoint region contains the addresses of all the blocks in the inode map and segment usage table, plus the current time and a pointer to the last segment written.

During reboot, Sprite LFS reads the checkpoint region and uses that information to initialize its main-memory data structures. In order to handle a crash during a checkpoint operation there are actually two checkpoint regions, and checkpoint operations alternate between them. The checkpoint time is in the last block of the checkpoint region, so if the checkpoint fails the time will not be updated. During reboot, the system reads both checkpoint regions and uses the one with the most recent time.

Sprite LFS performs checkpoints at periodic intervals as well as when the file system is unmounted or the system

is shut down. A long interval between checkpoints reduces the overhead of writing the checkpoints but increases the time needed to roll forward during recovery; a short checkpoint interval improves recovery time but increases the cost of normal operation. Sprite LFS currently uses a checkpoint interval of thirty seconds, which is probably much too short. An alternative to periodic checkpointing is to perform checkpoints after a given amount of new data has been written to the log; this would set a limit on recovery time while reducing the checkpoint overhead when the file system is not operating at maximum throughput.

4.2. Roll-forward

In principle it would be safe to restart after crashes by simply reading the latest checkpoint region and discarding any data in the log after that checkpoint. This would result in instantaneous recovery but any data written since the last checkpoint would be lost. In order to recover as much information as possible, Sprite LFS scans through the log segments that were written after the last checkpoint. This operation is called *roll-forward*.

During roll-forward Sprite LFS uses the information in segment summary blocks to recover recently-written file data. When a summary block indicates the presence of a new inode, Sprite LFS updates the inode map it read from the checkpoint, so that the inode map refers to the new copy of the inode. This automatically incorporates the file's new data blocks into the recovered file system. If data blocks are discovered for a file without a new copy of the file's inode, then the roll-forward code assumes that the new version of the file on disk is incomplete and it ignores the new data blocks.

The roll-forward code also adjusts the utilizations in the segment usage table read from the checkpoint. The utilizations of the segments written since the checkpoint will be zero; they must be adjusted to reflect the live data left after roll-forward. The utilizations of older segments will also have to be adjusted to reflect file deletions and overwrites (both of these can be identified by the presence of new inodes in the log).

The final issue in roll-forward is how to restore consistency between directory entries and inodes. Each inode contains a count of the number of directory entries referring to that inode; when the count drops to zero the file is deleted. Unfortunately, it is possible for a crash to occur when an inode has been written to the log with a new reference count while the block containing the corresponding directory entry has not yet been written, or vice versa.

To restore consistency between directories and inodes, Sprite LFS outputs a special record in the log for each directory change. The record includes an operation code (create, link, rename, or unlink), the location of the directory entry (i-number for the directory and the position within the directory), the contents of the directory entry (name and i-number), and the new reference count for the inode named in the entry. These records are collectively called the *directory operation log*; Sprite LFS guarantees that each directory operation log entry appears in the log before the corresponding directory block or inode.

During roll-forward, the directory operation log is used to ensure consistency between directory entries and inodes: if a log entry appears but the inode and directory block were not both written, roll-forward updates the directory and/or inode to complete the operation. Roll-forward operations can cause entries to be added to or removed from directories and reference counts on inodes to be updated. The recovery program appends the changed directories, inodes, inode map, and segment usage table blocks to the log and writes a new checkpoint region to include them. The only operation that can't be completed is the creation of a new file for which the inode is never written; in this case the directory entry will be removed. In addition to its other functions, the directory log made it easy to provide an atomic rename operation.

The interaction between the directory operation log and checkpoints introduced additional synchronization issues into Sprite LFS. In particular, each checkpoint must represent a state where the directory operation log is consistent with the inode and directory blocks in the log. This required additional synchronization to prevent directory modifications while checkpoints are being written.

5. Experience with the Sprite LFS

We began the implementation of Sprite LFS in late 1989 and by mid-1990 it was operational as part of the Sprite network operating system. Since the fall of 1990 it has been used to manage five different disk partitions, which are used by about thirty users for day-to-day computing. All of the features described in this paper have been implemented in Sprite LFS, but roll-forward has not yet been installed in the production system. The production disks use a short checkpoint interval (30 seconds) and discard all the information after the last checkpoint when they reboot.

When we began the project we were concerned that a log-structured file system might be substantially more complicated to implement than a traditional file system. In reality, however, Sprite LFS turns out to be no more complicated than Unix FFS[9]: Sprite LFS has additional complexity for the segment cleaner, but this is compensated by the elimination of the bitmap and layout policies required by Unix FFS; in addition, the checkpointing and roll-forward code in Sprite LFS is no more complicated than the *fsck* code[15] that scans Unix FFS disks to restore consistency. Logging file systems like Episode[2] or Cedar[3] are likely to be somewhat more complicated than either Unix FFS or Sprite LFS, since they include both logging and layout code.

In everyday use Sprite LFS does not feel much different to the users than the Unix FFS-like file system in Sprite. The reason is that the machines being used are not fast enough to be disk-bound with the current workloads. For example on the modified Andrew benchmark[11],

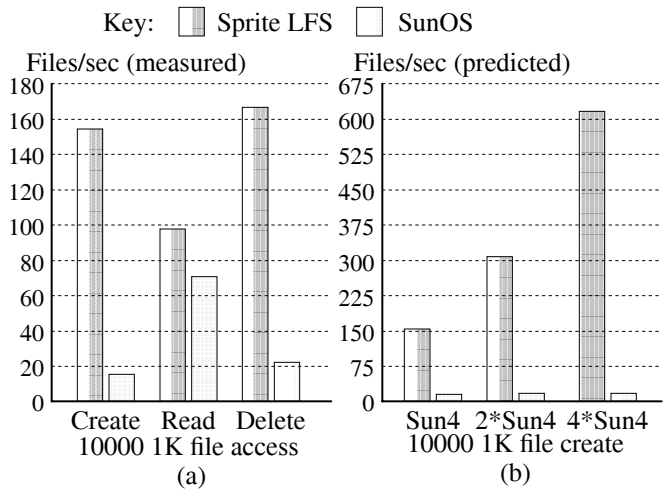


Figure 8 — Small-file performance under Sprite LFS and SunOS.
Figure (a) measures a benchmark that created 10000 one-kilobyte files, then read them back in the same order as created, then deleted them. Speed is measured by the number of files per second for each operation on the two file systems. The logging approach in Sprite LFS provides an order-of-magnitude speedup for creation and deletion. Figure (b) estimates the performance of each system for creating files on faster computers with the same disk. In SunOS the disk was 85% saturated in (a), so faster processors will not improve performance much. In Sprite LFS the disk was only 17% saturated in (a) while the CPU was 100% utilized; as a consequence I/O performance will scale with CPU speed.

Sprite LFS is only 20% faster that SunOS using the configuration presented in Section 5.1. Most of the speedup is attributable to the removal of the synchronous writes in Sprite LFS. Even with the synchronous writes of Unix FFS, the benchmark has a CPU utilization of over 80%, limiting the speedup possible from changes in the disk storage management.

5.1. Micro-benchmarks

We used a collection of small benchmark programs to measure the best-case performance of Sprite LFS and compare it to SunOS 4.0.3, whose file system is based on Unix FFS. The benchmarks are synthetic so they do not represent realistic workloads, but they illustrate the strengths and weaknesses of the two file systems. The machine used for both systems was a Sun-4/260 (8.7 integer SPECmarks) with 32 megabytes of memory, a Sun SCSI3 HBA, and a Wren IV disk (1.3 MBytes/sec maximum transfer bandwidth, 17.5 milliseconds average seek time). For both LFS and SunOS, the disk was formatted with a file system having around 300 megabytes of usable storage. An eight-kilobyte block size was used by SunOS while Sprite LFS used a four-kilobyte block size and a one-megabyte segment size. In each case the system was running multiuser but was otherwise quiescent during the test. For Sprite LFS no cleaning occurred during the benchmark runs so the measurements represent best-case performance; see Section 5.2 below for measurements of cleaning overhead.

Figure 8 shows the results of a benchmark that creates, reads, and deletes a large number of small files. Sprite LFS is almost ten times as fast as SunOS for the create and delete phases of the benchmark. Sprite LFS is also faster for reading the files back; this is because the files are read in the same order created and the log-structured file system packs the files densely in the log. Furthermore, Sprite LFS only kept the disk 17% busy during the create phase while saturating the CPU. In contrast, SunOS kept the disk busy 85% of the time during the create phase, even though only about 1.2% of the disk's potential bandwidth was used for new data. This means that the performance of Sprite LFS will improve by another factor of 4-6 as CPUs get faster (see Figure 8(b)). Almost no improvement can be expected in SunOS.

Although Sprite was designed for efficiency on workloads with many small file accesses, Figure 9 shows that it also provides competitive performance for large files. Sprite LFS has a higher write bandwidth than SunOS in all cases. It is substantially faster for random writes because it turns them into sequential writes to the log; it is also faster for sequential writes because it groups many blocks into a single large I/O, whereas SunOS performs

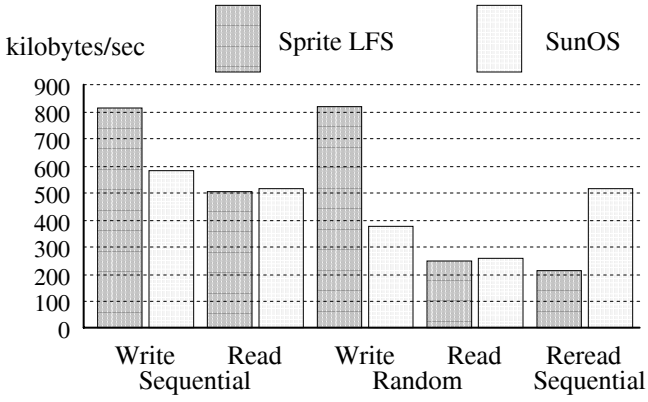


Figure 9 — Large-file performance under Sprite LFS and SunOS.
The figure shows the speed of a benchmark that creates a 100-Mbyte file with sequential writes, then reads the file back sequentially, then writes 100 Mbytes randomly to the existing file, then reads 100 Mbytes randomly from the file, and finally reads the file sequentially again. The bandwidth of each of the five phases is shown separately. Sprite LFS has a higher write bandwidth and the same read bandwidth as SunOS with the exception of sequential reading of a file that was written randomly.

individual disk operations for each block (a newer version of SunOS groups writes [16] and should therefore have performance equivalent to Sprite LFS). The read performance is similar in the two systems except for the case of reading a file sequentially after it has been written randomly; in this case the reads require seeks in Sprite LFS, so its performance is substantially lower than SunOS.

Figure 9 illustrates the fact that a log-structured file system produces a different form of locality on disk than traditional file systems. A traditional file system achieves *logical locality* by assuming certain access patterns (sequential reading of files, a tendency to use multiple files within a directory, etc.); it then pays extra on writes, if necessary, to organize information optimally on disk for the assumed read patterns. In contrast, a log-structured file system achieves *temporal locality*: information that is created or modified at the same time will be grouped closely on disk. If temporal locality matches logical locality, as it does for a file that is written sequentially and then read sequentially, then a log-structured file system should have about the same performance on large files as a traditional file system. If temporal locality differs from logical locality then the systems will perform differently. Sprite LFS handles random writes more efficiently because it writes them sequentially on disk. SunOS pays more for the random writes in order to achieve logical locality, but then it handles sequential re-reads more efficiently. Random reads have about the same performance in the two systems, even though the blocks are laid out very differently. However, if the nonsequential reads occurred in the same order as the nonsequential writes then Sprite would have been much faster.

5.2. Cleaning overheads

The micro-benchmark results of the previous section give an optimistic view of the performance of Sprite LFS because they do not include any cleaning overheads (the write cost during the benchmark runs was 1.0). In order to assess the cost of cleaning and the effectiveness of the cost-benefit cleaning policy, we recorded statistics about our production log-structured file systems over a period of several months. Five systems were measured:

`/user6` Home directories for Sprite developers. Workload consists of program development, text processing, electronic communication, and simulations.

`/pcs` Home directories and project area for research on parallel processing and VLSI circuit design.

`/src/kernel` Sources and binaries for the Sprite kernel.

`/swap2` Sprite client workstation swap files. Workload consists of virtual memory backing store for 40 diskless Sprite workstations. Files tend to be large, sparse, and accessed nonsequentially.

`/tmp` Temporary file storage area for 40 Sprite workstations.

Table 2 shows statistics gathered during cleaning over a four-month period. In order to eliminate start-up effects we waited several months after putting the file systems into use before beginning the measurements. The behavior of the production file systems has been substantially better than predicted by the simulations in Section 3. Even though the overall disk capacity utilizations ranged from 11-75%, more than half of the segments cleaned were totally empty. Even the non-empty segments have utilizations far less than the average disk utilizations. The overall write costs ranged from 1.2 to 1.6, in comparison to write costs of 2.5-3 in the corresponding simulations. Figure 10 shows the distribution of segment utilizations, gathered in a recent snapshot of the `/user6` disk.

We believe that there are two reasons why cleaning costs are lower in Sprite LFS than in the simulations. First,

Write cost in Sprite LFS file systems								
File system	Disk Size	Avg File Size	Avg Write Traffic	In Use	Segments Cleaned	Segments Empty	*u* Avg	Write Cost
`/user6`	1280 MB	23.5 KB	3.2 MB/hour	75%	10732	69%	.133	1.4
`/pcs`	990 MB	10.5 KB	2.1 MB/hour	63%	22689	52%	.137	1.6
`/src/kernel`	1280 MB	37.5 KB	4.2 MB/hour	72%	16975	83%	.122	1.2
`/tmp`	264 MB	28.9 KB	1.7 MB/hour	11%	2871	78%	.130	1.3
`/swap2`	309 MB	68.1 KB	13.3 MB/hour	65%	4701	66%	.535	1.6

Table 2 - Segment cleaning statistics and write costs for production file systems.
For each Sprite LFS file system the table lists the disk size, the average file size, the average daily write traffic rate, the average disk capacity utilization, the total number of segments cleaned over a four-month period, the fraction of the segments that were empty when cleaned, the average utilization of the non-empty segments that were cleaned, and the overall write cost for the period of the measurements. These write cost figures imply that the cleaning overhead limits the long-term write performance to about 70% of the maximum sequential write bandwidth.

all the files in the simulations were just a single block long. In practice, there are a substantial number of longer files, and they tend to be written and deleted as a whole. This results in greater locality within individual segments. In the best case where a file is much longer than a segment, deleting the file will produce one or more totally empty segments. The second difference between simulation and reality is that the simulated reference patterns were evenly distributed within the hot and cold file groups. In practice there are large numbers of files that are almost never written (cold segments in reality are much colder than the cold segments in the simulations). A log-structured file system will isolate the very cold files in segments and never clean them. In the simulations, every segment eventually received modifications and thus had to be cleaned.

If the measurements of Sprite LFS in Section 5.1 were a bit over-optimistic, the measurements in this section are, if anything, over-pessimistic. In practice it may be possible to perform much of the cleaning at night or during other idle periods, so that clean segments are available during bursts of activity. We do not yet have enough experience with Sprite LFS to know if this can be done. In addition, we expect the performance of Sprite LFS to improve as we gain experience and tune the algorithms. For example, we have not yet carefully analyzed the policy issue of how many segments to clean at a time, but we think it may impact the system's ability to segregate hot data from cold data.

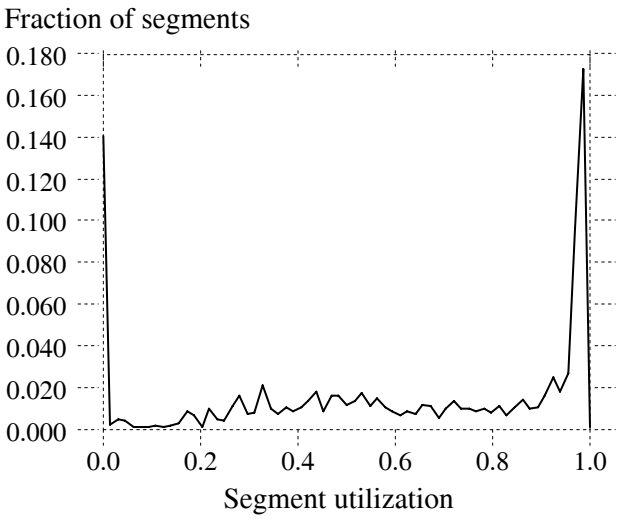


Figure 10 — Segment utilization in the /user6 file system
This figure shows the distribution of segment utilizations in a recent snapshot of the `/user6` disk. The distribution shows large numbers of fully utilized segments and totally empty segments.

5.3. Crash recovery

Although the crash recovery code has not been installed on the production system, the code works well enough to time recovery of various crash scenarios. The time to recover depends on the checkpoint interval and the rate and type of operations being performed. Table 3 shows the recovery time for different file sizes and amounts of file data recovered. The different crash configurations were generated by running a program that created one, ten, or fifty megabytes of fixed-size files before the system was crashed. A special version of Sprite LFS was used that had an infinite checkpoint interval and never wrote directory changes to disk. During the recovery roll-forward, the created files had to be added to the inode map, the directory entries created, and the segment usage table updated.

Table 3 shows that recovery time varies with the number and size of files written between the last checkpoint and the crash. Recovery times can be bounded by limiting the amount of data written between checkpoints. From the average file sizes and daily write traffic in Table 2, a checkpoint interval as large as an hour would result in average recovery times of around one second. Using the maximum observered write rate of 150 megabytes/hour, maximum recovery time would grow by one second for every 70 seconds of checkpoint interval length.

5.4. Other overheads in Sprite LFS

Table 4 shows the relative importance of the various kinds of data written to disk, both in terms of how much of the live blocks they occupy on disk and in terms of how much of the data written to the log they represent. More than 99% of the live data on disk consists of file data blocks and indirect blocks. However, about 13% of the information written to the log consists of inodes, inode map blocks, and segment map blocks, all of which tend to be overwritten quickly. The inode map alone accounts for more than 7% of all the data written to the log. We suspect that this is because of the short checkpoint interval currently used in Sprite LFS, which forces metadata to disk

Sprite LFS recovery time in seconds			
File Size	File Data Recovered		
	1 MB	10 MB	50 MB
1 KB	1	21	132
10 KB	< 1	3	17
100 KB	< 1	1	8

Table 3 — Recovery time for various crash configurations
The table shows the speed of recovery of one, ten, and fifty megabytes of fixed-size files. The system measured was the same one used in Section 5.1. Recovery time is dominated by the number of files to be recovered.

more often than necessary. We expect the log bandwidth overhead for metadata to drop substantially when we install roll-forward recovery and increase the checkpoint interval.

6. Related work

The log-structured file system concept and the Sprite LFS design borrow ideas from many different storage management systems. File systems with log-like structures have appeared in several proposals for building file systems on write-once media[17, 18]. Besides writing all changes in an append-only fashion, these systems maintain indexing information much like the Sprite LFS inode map and inodes for quickly locating and reading files. They differ from Sprite LFS in that the write-once nature of the media made it unnecessary for the file systems to reclaim log space.

The segment cleaning approach used in Sprite LFS acts much like scavenging garbage collectors developed for programming languages[19]. The cost-benefit segment selection and the age sorting of blocks during segment cleaned in Sprite LFS separates files into generations much like generational garbage collection schemes[20]. A significant difference between these garbage collection schemes and Sprite LFS is that efficient random access is possible in the generational garbage collectors, whereas sequential accesses are necessary to achieve high performance in a file system. Also, Sprite LFS can exploit the fact that blocks can belong to at most one file at a time to use much simpler algorithms for identifying garbage than used in the systems for programming languages.

The logging scheme used in Sprite LFS is similar to schemes pioneered in database systems. Almost all database systems use write-ahead logging for crash recovery and high performance[13], but differ from Sprite LFS in how they use the log. Both Sprite LFS and the database systems view the log as the most up to date ''truth'' about the state of the data on disk. The main difference is that database systems do not use the log as the final repository for data: a separate data area is reserved for this purpose. The separate data area of these database systems means that they do not need the segment cleaning mechanisms of the Sprite LFS to reclaim log space. The space occupied by the log in a database system can be reclaimed when the logged changes have been written to their final locations. Since all read requests are processed from the data area, the log can be greatly compacted without hurting read performance. Typically only the changed bytes are written to database logs rather than entire blocks as in Sprite LFS.

The Sprite LFS crash recovery mechanism of checkpoints and roll forward using a ''redo log'' is similar to techniques used in database systems and object repositories[21]. The implementation in Sprite LFS is simplified because the log is the final home of the data. Rather than redoing the operation to the separate data copy, Sprite LFS recovery insures that the indexes point at the newest copy of the data in the log.

Collecting data in the file cache and writing it to disk in large writes is similar to the concept of group commit in database systems[22] and to techniques used in main-memory database systems[23, 24].

Sprite LFS `/user6` file system contents		
Block type	Live data	Log bandwidth
Data blocks*	98.0%	85.2%
Indirect blocks*	1.0%	1.6%
Inode blocks*	0.2%	2.7%
Inode map	0.2%	7.8%
Seg Usage map*	0.0%	2.1%
Summary blocks	0.6%	0.5%
Dir Op Log	0.0%	0.1%

Table 4 — Disk space and log bandwidth usage of /user6
For each block type, the table lists the percentage of the disk space in use on disk (Live data) and the percentage of the log bandwidth consumed writing this block type (Log bandwidth). The block types marked with '*' have equivalent data structures in Unix FFS.

7. Conclusion

The basic principle behind a log-structured file system is a simple one: collect large amounts of new data in a file cache in main memory, then write the data to disk in a single large I/O that can use all of the disk's bandwidth. Implementing this idea is complicated by the need to maintain large free areas on disk, but both our simulation analysis and our experience with Sprite LFS suggest that low cleaning overheads can be achieved with a simple policy based on cost and benefit. Although we developed a log-structured file system to support workloads with many small files, the approach also works very well for large-file accesses. In particular, there is essentially no cleaning overhead at all for very large files that are created and deleted in their entirety.

The bottom line is that a log-structured file system can use disks an order of magnitude more efficiently than existing file systems. This should make it possible to take advantage of several more generations of faster processors before I/O limitations once again threaten the scalability of computer systems.

8. Acknowledgments

Diane Greene, Mary Baker, John Hartman, Mike Kupfer, Ken Shirriff and Jim Mott-Smith provided helpful comments on drafts of this paper.

References

1. John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James

G. Thompson, ''A Trace-Driven Analysis of the Unix 4.2 BSD File System,'' *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 15-24 ACM, (1985).

2. Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas, ''DEcorum File System Architectural Overview,'' *Proceedings of the USENIX 1990 Summer Conference*, pp. 151-164 (Jun 1990).

3. Robert B. Hagmann, ''Reimplementing the Cedar File System Using Logging and Group Commit,'' *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 155-162 (Nov 1987).

4. John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch, ''The Sprite Network Operating System,'' *IEEE Computer* **21**(2) pp. 23-36 (1988).

5. David A. Patterson, Garth Gibson, and Randy H. Katz, ''A Case for Redundant Arrays of Inexpensive Disks (RAID),'' *ACM SIGMOD 88*, pp. 109-116 (Jun 1988).

6. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, ''Measurements of a Distributed File System,'' *Proceedings of the 13th Symposium on Operating Systems Principles*, ACM, (Oct 1991).

7. M. Satyanarayanan, ''A Study of File Sizes and Functional Lifetimes,'' *Proceedings of the 8th Symposium on Operating Systems Principles*, pp. 96-108 ACM, (1981).

8. Edward D. Lazowska, John Zahorjan, David R Cheriton, and Willy Zwaenepoel, ''File Access Performance of Diskless Workstations,'' *Transactions on Computer Systems* **4**(3) pp. 238-268 (Aug 1986).

9. Marshall K. McKusick, ''A Fast File System for Unix,'' *Transactions on Computer Systems* **2**(3) pp. 181-197 ACM, (1984).

10. R. Sandberg, ''Design and Implementation of the Sun Network Filesystem,'' *Proceedings of the USENIX 1985 Summer Conference*, pp. 119-130 (Jun 1985).

11. John K. Ousterhout, ''Why Aren't Operating Systems Getting Faster As Fast as Hardware?,'' *Proceedings of the USENIX 1990 Summer Conference*, pp. 247-256 (Jun 1990).

12. Margo I. Seltzer, Peter M. Chen, and John K. Ousterhout, ''Disk Scheduling Revisited,'' *Proceedings of the Winter 1990 USENIX Technical Conference*, (January 1990).

13. Jim Gray, ''Notes on Data Base Operating Systems,'' in *Operating Systems, An Advanced Course*, Springer-Verlag (1979).

14. A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter, ''Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors,'' *IBM Journal of Research and Development* **34**(1) pp. 105-109 (Jan 1990).

15. Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry, ''Fsck - The UNIX File System Check Program,'' *Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version*, USENIX, (Apr 1986).

16. Larry McVoy and Steve Kleiman, ''Extent-like Performance from a UNIX File System,'' *Proceedings of the USENIX 1991 Winter Conference*, (Jan 1991).

17. D. Reed and Liba Svobodova, ''SWALLOW: A Distributed Data Storage System for a Local Network,'' *Local Networks for Computer Communications*, pp. 355-373 North-Holland, (1981).

18. Ross S. Finlayson and David R. Cheriton, ''Log Files: An Extended File Service Exploiting Write-Once Storage,'' *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 129-148 ACM, (Nov 1987).

19. H. G. Baker, ''List Processing in Real Time on a Serial Computer,'' A.I. Working Paper 139, MIT-AI Lab, Boston, MA (April 1977).

20. Henry Lieberman and Carl Hewitt, ''A Real-Time Garbage Collector Based on the Lifetimes of Objects,'' *Communications of the ACM* **26**(6) pp. 419-429 (1983).

21. Brian M. Oki, Barbara H. Liskov, and Robert W. Scheifler, ''Reliable Object Storage to Support Atomic Actions,'' *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 147-159 ACM, (1985).

22. David J. DeWitt, Randy H. Katz, Frank Olken, L. D. Shapiro, Mike R. Stonebraker, and David Wood, ''Implementation Techniques for Main Memory Database Systems,'' *Proceedings of SIGMOD 1984*, pp. 1-8 (Jun 1984).

23. Kenneth Salem and Hector Garcia-Molina, ''Crash Recovery Mechanisms for Main Storage Database Systems,'' CS-TR-034-86, Princeton University, Princeton, NJ (1986).

24. Robert B. Hagmann, ''A Crash Recovery Scheme for a Memory-Resident Database System,'' *IEEE Transactions on Computers* **C-35**(9)(Sep 1986).

The HP AutoRAID Hierarchical Storage System

JOHN WILKES, RICHARD GOLDING, CARL STAELIN, and TIM SULLIVAN
Hewlett-Packard Laboratories

Configuring redundant disk arrays is a black art. To configure an array properly, a system administrator must understand the details of both the array and the workload it will support. Incorrect understanding of either, or changes in the workload over time, can lead to poor performance. We present a solution to this problem: a two-level storage hierarchy implemented inside a single disk-array controller. In the upper level of this hierarchy, two copies of active data are stored to provide full redundancy and excellent performance. In the lower level, RAID 5 parity protection is used to provide excellent storage cost for inactive data, at somewhat lower performance. The technology we describe in this article, known as HP AutoRAID, automatically and transparently manages migration of data blocks between these two levels as access patterns change. The result is a fully redundant storage system that is extremely easy to use, is suitable for a wide variety of workloads, is largely insensitive to dynamic workload changes, and performs much better than disk arrays with comparable numbers of spindles and much larger amounts of front-end RAM cache. Because the implementation of the HP AutoRAID technology is almost entirely in software, the additional hardware cost for these benefits is very small. We describe the HP AutoRAID technology in detail, provide performance data for an embodiment of it in a storage array, and summarize the results of simulation studies used to choose algorithms implemented in the array.

Categories and Subject Descriptors: B.4.2 [**Input/Output and Data Communications**]: Input/Output Devices—*channels and controllers*; B.4.5 [**Input/Output and Data Communications**]: Reliability, Testing, and Fault-Tolerance—*redundant design*; D.4.2 [**Operating Systems**]: Storage Management—*secondary storage*

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Disk array, RAID, storage hierarchy

1. INTRODUCTION

Modern businesses and an increasing number of individuals depend on the information stored in the computer systems they use. Even though modern disk drives have mean-time-to-failure (MTTF) values measured in hundreds of years, storage needs have increased at an enormous rate, and a sufficiently large collection of such devices can still experience inconveniently frequent

Authors' address: Hewlett-Packard Laboratories, 1501 Page Mill Road, MS 1U13, Palo Alto, CA 94304-1126; email: {wilkes; golding; staelin; sullivan}@hpl.hp.com.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1996 ACM 0734-2071/96/0200-0108$03.50

failures. Worse, completely reloading a large storage system from backup tapes can take hours or even days, resulting in very costly downtime.

For small numbers of disks, the preferred method of providing fault protection is to duplicate (*mirror*) data on two disks with independent failure modes. This solution is simple, and it performs well.

However, once the total number of disks gets large, it becomes more cost effective to employ an array controller that uses some form of partial redundancy (such as parity) to protect the data it stores. Such RAIDs (for Redundant Arrays of Independent Disks) were first described in the early 1980s [Lawlor 1981; Park and Balasubramanian 1986] and popularized by the work of a group at UC Berkeley [Patterson et al. 1988; 1989]. By storing only partial redundancy for the data, the incremental cost of the desired high availability is reduced to as little as 1/N of the total storage-capacity cost (where N is the number of disks in the array), plus the cost of the array controller itself.

The UC Berkeley RAID terminology has a number of different RAID *levels*, each one representing a different amount of redundancy and a placement rule for the redundant data. Most disk array products implement RAID level 3 or 5. In RAID level 3, host data blocks are bit- or byte-interleaved across a set of data disks, and parity is stored on a dedicated data disk (see Figure 1(a)). In RAID level 5, host data blocks are block-interleaved across the disks, and the disk on which the parity block is stored rotates in round-robin fashion for different stripes (see Figure 1(b)). Both hardware and software RAID products are available from many vendors.

Unfortunately, current disk arrays are often difficult to use [Chen and Lee 1993]: the different RAID levels have different performance characteristics and perform well only for a relatively narrow range of workloads. To accommodate this, RAID systems typically offer a great many configuration parameters: data- and parity-layout choice, stripe depth, stripe width, cache sizes and write-back policies, and so on. Setting these correctly is difficult: it requires knowledge of workload characteristics that most people are unable (and unwilling) to acquire. As a result, setting up a RAID array is often a daunting task that requires skilled, expensive people and—in too many cases—a painful process of trial and error.

Making the wrong choice has two costs: the resulting system may perform poorly; and changing from one layout to another almost inevitably requires copying data off to a second device, reformatting the array, and then reloading it. Each step of this process can take hours; it is also an opportunity for inadvertent data loss through operator error—one of the commonest sources of problems in modern computer systems [Gray 1990].

Adding capacity to an existing array is essentially the same problem: taking full advantage of a new disk usually requires a reformat and data reload.

Since RAID 5 arrays suffer reduced performance in "degraded mode"—when one of the drives has failed—many include a provision for one or more spare disks that can be pressed into service as soon as an active disk fails. This allows redundancy reconstruction to commence immediately, thereby reduc-

Fig. 1. Data and parity layout for two different RAID levels.

ing the window of vulnerability to data loss from a second device failure and minimizing the duration of the performance degradation. In the normal case, however, these spare disks are not used and contribute nothing to the performance of the system. (There is also the secondary problem of assuming that a spare disk is still working: because the spare is idle, the array controller may not find out that it has failed until it is too late.)

1.1 The Solution: A Managed Storage Hierarchy

Fortunately, there is a solution to these problems for a great many applications of disk arrays: a redundancy-level storage hierarchy. The basic idea is to combine the performance advantages of mirroring with the cost-capacity benefits of RAID 5 by mirroring active data and storing relatively inactive or read-only data in RAID 5.

To make this solution work, part of the data must be active and part inactive (else the cost performance would reduce to that of mirrored data), and the active subset must change relatively slowly over time (to allow the array to do useful work, rather than just move data between the two levels). Fortunately, studies on I/O access patterns, disk shuffling, and file system restructuring have shown that these conditions are often met in practice [Akyurek and Salem 1993; Deshpandee and Bunt 1988; Floyd and Schlatter Ellis 1989; Geist et al. 1994; Majumdar 1984; McDonald and Bunt 1989; McNutt 1994; Ruemmler and Wilkes 1991; 1993; Smith 1981].

Such a storage hierarchy could be implemented in a number of different ways:

—*Manually*, by the system administrator. (This is how large mainframes have been run for decades. Gelb [1989] discusses a slightly refined version of this basic idea.) The advantage of this approach is that human intelligence can be brought to bear on the problem, and perhaps knowledge that is not available to the lower levels of the I/O and operating systems. However, it is obviously error prone (the wrong choices can be made, and mistakes can be made in moving data from one level to another); it cannot adapt to rapidly changing access patterns; it requires highly skilled people; and it does not allow new resources (such as disk drives) to be added to the system easily.

—*In the file system*, perhaps on a per-file basis. This might well be the best possible place in terms of a good balance of knowledge (the file system can track access patterns on a per-file basis) and implementation freedom.

Unfortunately, there are many different file system implementations in customers' hands, so deployment is a major problem.

—*In a smart array controller*, behind a block-level device interface such as the Small Systems Computer Interface (*SCSI*) *standard* [SCSI 1991]. Although this level has the disadvantage that knowledge about files has been lost, it has the enormous compensating advantage of being easily deployable—strict adherence to the standard means that an array using this approach can look just like a regular disk array, or even just a set of plain disk drives.

Not surprisingly, we are describing an array-controller-based solution here. We use the name "HP AutoRAID" to refer both to the collection of technology developed to make this possible and to its embodiment in an array controller.

1.2 Summary of the Features of HP AutoRAID

We can summarize the features of HP AutoRAID as follows:

Mapping. Host block addresses are internally mapped to their physical locations in a way that allows transparent migration of individual blocks.

Mirroring. Write-active data are mirrored for best performance and to provide single-disk failure redundancy.

RAID 5. Write-inactive data are stored in RAID 5 for best cost capacity while retaining good read performance and single-disk failure redundancy. In addition, large sequential writes go directly to RAID 5 to take advantage of its high bandwidth for this access pattern.

Adaptation to Changes in the Amount of Data Stored. Initially, the array starts out empty. As data are added, internal space is allocated to mirrored storage until no more data can be stored this way. When this happens, some of the storage space is automatically reallocated to the RAID 5 storage class, and data are migrated down into it from the mirrored storage class. Since the RAID 5 layout is a more compact data representation, more data can now be stored in the array. This reapportionment is allowed to proceed until the capacity of the mirrored storage has shrunk to about 10% of the total usable space. (The exact number is a policy choice made by the implementors of the HP AutoRAID firmware to maintain good performance.) Space is apportioned in coarse-granularity 1MB units.

Adaptation to Workload Changes. As the active set of data changes, newly active data are promoted to mirrored storage, and data that have become less active are demoted to RAID 5 in order to keep the amount of mirrored data roughly constant. Because these data movements can usually be done in the background, they do not affect the performance of the array. Promotions and demotions occur completely automatically, in relatively fine-granularity 64KB units.

Hot-Pluggable Disks, Fans, Power Supplies, and Controllers. These allow a failed component to be removed and a new one inserted while the system continues to operate. Although these are relatively commonplace features in

higher-end disk arrays, they are important in enabling the next three features.

On-Line Storage Capacity Expansion. A disk can be added to the array at any time, up to the maximum allowed by the physical packaging—currently 12 disks. The system automatically takes advantage of the additional space by allocating more mirrored storage. As time and the workload permit, the active data are rebalanced across the available drives to even out the workload between the newcomer and the previous disks—thereby getting maximum performance from the system.

Easy Disk Upgrades. Unlike conventional arrays, the disks do not all need to have the same capacity. This has two advantages: first, each new drive can be purchased at the optimal capacity/cost/performance point, without regard to prior selections. Second, the entire array can be upgraded to a new disk type (perhaps with twice the capacity) without interrupting its operation by removing one old disk at a time, inserting a replacement disk, and then waiting for the automatic data reconstruction and rebalancing to complete. To eliminate the reconstruction, data could first be "drained" from the disk being replaced: this would have the advantage of retaining continuous protection against disk failures during this process, but would require enough spare capacity in the system.

Controller Fail-Over. A single array can have two controllers, each one capable of running the entire subsystem. On failure of the primary, the operations are rolled over to the other. A failed controller can be replaced while the system is active. Concurrently active controllers are also supported.

Active Hot Spare. The spare space needed to perform a reconstruction can be spread across all of the disks and used to increase the amount of space for mirrored data—and thus the array's performance—rather than simply being left idle.

If a disk fails, mirrored data are demoted to RAID 5 to provide the space to reconstruct the desired redundancy. Once this process is complete, a second disk failure can be tolerated—and so on, until the physical capacity is entirely filled with data in the RAID 5 storage class.

Simple Administration and Setup. A system administrator can divide the storage space of the array into one or more logical units (LUNs in SCSI terminology) to correspond to the logical groupings of the data to be stored. Creating a new LUN or changing the size of an existing LUN is trivial: it takes about 10 seconds to go through the front-panel menus, select a size, and confirm the request. Since the array does not need to be formatted in the traditional sense, the creation of the LUN does not require a pass over all the newly allocated space to zero it and initialize its parity, an operation that can take hours in a regular array. Instead, all that is needed is for the controller's data structures to be updated.

Log-Structured RAID 5 Writes. A well-known problem of RAID 5 disk arrays is the so-called small-write problem. Doing an update-in-place of part of a stripe takes 4 I/Os: old data and parity have to be read, new parity calculated, and then new data and new parity written back. HP AutoRAID

avoids this overhead in most cases by writing to its RAID 5 storage in a log-structured fashion—that is, only empty areas of disk are written to, so no old-data or old-parity reads are required.

1.3 Related Work

Many papers have been published on RAID reliability, performance, and design variations for parity placement and recovery schemes (see Chen et al. [1994] for an annotated bibliography). The HP AutoRAID work builds on many of these studies: we concentrate here on the architectural issues of using multiple RAID levels (specifically 1 and 5) in a single array controller.

Storage Technology Corporation's Iceberg [Ewing 1993; STK 1995] uses a similar indirection scheme to map logical IBM mainframe disks (count-key-data format) onto an array of 5.25-inch SCSI disk drives (Art Rudeseal, private communication, Nov., 1994). Iceberg has to handle variable-sized records; HP AutoRAID has a SCSI interface and can handle the indirection using fixed-size blocks. The emphasis in the Iceberg project seems to have been on achieving extraordinarily high levels of availability; the emphasis in HP AutoRAID has been on performance once the single-component failure model of regular RAID arrays had been achieved. Iceberg does not include multiple RAID storage levels: it simply uses a single-level modified RAID 6 storage class [Dunphy et al. 1991; Ewing 1993].

A team at IBM Almaden has done extensive work in improving RAID array controller performance and reliability, and several of their ideas have seen application in IBM mainframe storage controllers. Their floating-parity scheme [Menon and Kasson 1989; 1992] uses an indirection table to allow parity data to be written in a nearby slot, not necessarily its original location. This can help to reduce the small-write penalty of RAID 5 arrays. Their distributed sparing concept [Menon and Mattson 1992] spreads the spare space across all the disks in the array, allowing all the spindles to be used to hold data. HP AutoRAID goes further than either of these: it allows both data and parity to be relocated, and it uses the distributed spare capacity to increase the fraction of data held in mirrored form, thereby improving performance still further. Some of the schemes described in Menon and Courtney [1993] are also used in the dual-controller version of the HP AutoRAID array to handle controller failures.

The Loge disk drive controller [English and Stepanov 1992] and its follow-ons Mime [Chao et al. 1992] and Logical Disk [de Jonge et al. 1993] all used a scheme of keeping an indirection table to fixed-sized blocks held on secondary storage. None of these supported multiple storage levels, and none was targeted at RAID arrays. Work on an Extended Function Controller at HP's disk divisions in the 1980s looked at several of these issues, but progress awaited development of suitable controller technologies to make the approach adopted in HP AutoRAID cost effective.

The log-structured writing scheme used in HP AutoRAID owes an intellectual debt to the body of work on log-structured file systems (LFS) [Carson and Setia 1992; Ousterhout and Douglis 1989; Rosenblum and Ousterhout

1992; Seltzer et al. 1993; 1995] and cleaning (garbage collection) policies for them [Blackwell et al. 1995; McNutt 1994; Mogi and Kitsuregawa 1994].

There is a large body of literature on hierarchical storage systems and the many commercial products in this domain (for example, Chen [1973], Cohen et al. [1989], DEC [1993], Deshpandee and Bunt [1988], Epoch Systems [1988], Gelb [1989], Henderson and Poston [1989], Katz et al. [1991], Miller [1991], Misra [1981], Sienknecht et al. [1994], and Smith [1981], together with much of the *Proceedings of the IEEE Symposia on Mass Storage Systems*). Most of this work has been concerned with wider performance disparities between the levels than exist in HP AutoRAID. For example, such systems often use disk and robotic tertiary storage (tape or magneto-optical disk) as the two levels.

Several hierarchical storage systems have used front-end disks to act as a cache for data on tertiary storage. In HP AutoRAID, however, the mirrored storage is not a cache: instead data are moved between the storage classes, residing in precisely one class at a time. This method maximizes the overall storage capacity of a given number of disks.

The Highlight system [Kohl et al. 1993] extended LFS to two-level storage hierarchies (disk and tape) and used fixed-size segments. Highlight's segments were around 1MB in size, however, and therefore were much better suited for tertiary-storage mappings than for two secondary-storage levels.

Schemes in which inactive data are compressed [Burrows et al. 1992; Cate 1990; Taunton 1991] exhibit some similarities to the storage-hierarchy component of HP AutoRAID, but operate at the file system level rather than at the block-based device interface.

Finally, like most modern array controllers, HP AutoRAID takes advantage of the kind of optimizations noted in Baker et al. [1991] and Ruemmler and Wilkes [1993] that become possible with nonvolatile memory.

1.4 Roadmap to Remainder of Article

The remainder of the article is organized as follows. We begin with an overview of the technology: how an HP AutoRAID array controller works. Next come two sets of performance studies. The first is a set of measurements of a product prototype; the second is a set of simulation studies used to evaluate algorithm choices for HP AutoRAID. Finally, we conclude the article with a summary of the benefits of the technology.

2. THE TECHNOLOGY

This section introduces the basic technologies used in HP AutoRAID. It starts with an overview of the hardware, then discusses the layout of data on the disks of the array, including the structures used for mapping data blocks to their locations on disk. This is followed by an overview of normal read and write operations to illustrate the flow of data through the system, and then by descriptions of a series of operations that are usually performed in the background to ensure that the performance of the system remains high over long periods of time.

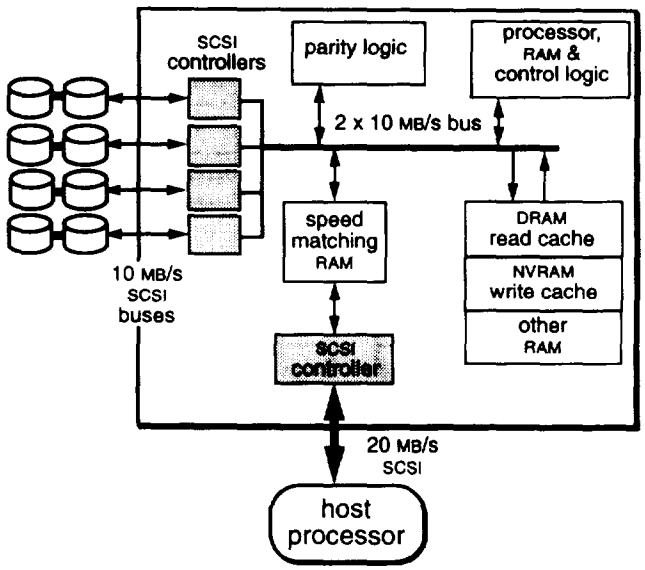


Fig. 2. Overview of HP AutoRAID hardware.

2.1 The HP AutoRAID Array Controller Hardware

An HP AutoRAID array is fundamentally similar to a regular RAID array. That is, it has a set of disks, an intelligent controller that incorporates a microprocessor, mechanisms for calculating parity, caches for staging data (some of which are nonvolatile), a connection to one or more host computers, and appropriate speed-matching buffers. Figure 2 is an overview of this hardware.

The hardware prototype for which we provide performance data uses four back-end SCSI buses to connect to its disks and one or two fast-wide SCSI buses for its front-end host connection. Many other alternatives exist for packaging this technology, but are outside the scope of this article.

The array presents one or more SCSI logical units (LUNs) to its hosts. Each of these is treated as a virtual device inside the array controller: their storage is freely intermingled. A LUN's size may be increased at any time (subject to capacity constraints). Not every block in a LUN must contain valid data—if nothing has been stored at an address, the array controller need not allocate any physical space to it.

2.2 Data Layout

Much of the intelligence in an HP AutoRAID controller is devoted to managing data placement on the disks. A two-level allocation scheme is used.

2.2.1 *Physical Data Layout: PEGs, PEXes, and Segments.* First, the data space on the disks is broken up into large-granularity objects called Physical EXtents (PEXes), as shown in Figure 3. PEXes are typically 1MB in size.

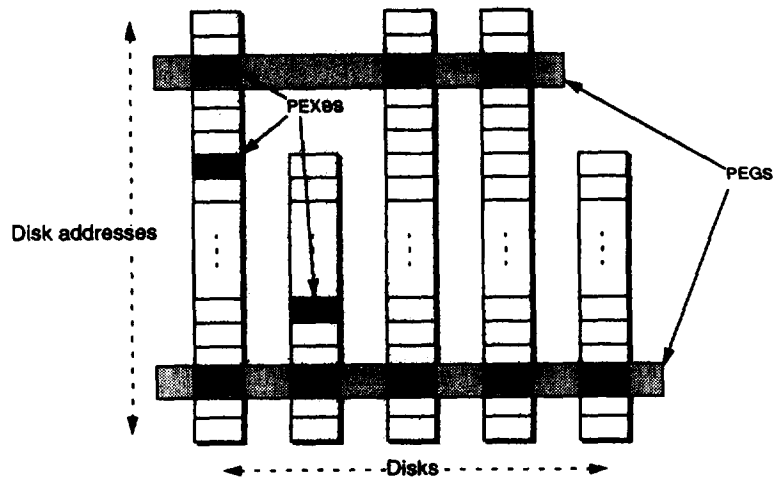


Fig. 3. Mapping of PEGs and PEXes onto disks (adapted from Burkes et al. [1995]).

Table I. A Summary of HP AutoRAID Data Layout Terminology

Term	Meaning	Size
PEX (physical extent)	Unit of physical space allocation.	1MB
PEG (physical extent group)	A group of PEXes, assigned to one storage class.	*
Stripe	One row of parity and data segments in a RAID 5 storage class.	*
Segment	Stripe unit (RAID 5) or half of a mirroring unit.	128KB
RB (relocation block)	Unit of data migration.	64KB
LUN (logical unit)	Host-visible virtual disk.	User settable

* Depends on the number of disks.

Several PEXes can be combined to make a Physical Extent Group (PEG). In order to provide enough redundancy to make it usable by either the mirrored or the RAID 5 storage class, a PEG includes at least three PEXes on different disks. At any given time, a PEG may be assigned to the mirrored storage class or the RAID 5 storage class, or may be unassigned, so we speak of mirrored, RAID 5, and free PEGS. (Our terminology is summarized in Table I.)

PEXes are allocated to PEGs in a manner that balances the amount of data on the disks (and thereby, hopefully, the load on the disks) while retaining the redundancy guarantees (no two PEXes from one disk can be used in the same stripe, for example). Because the disks in an HP AutoRAID array can

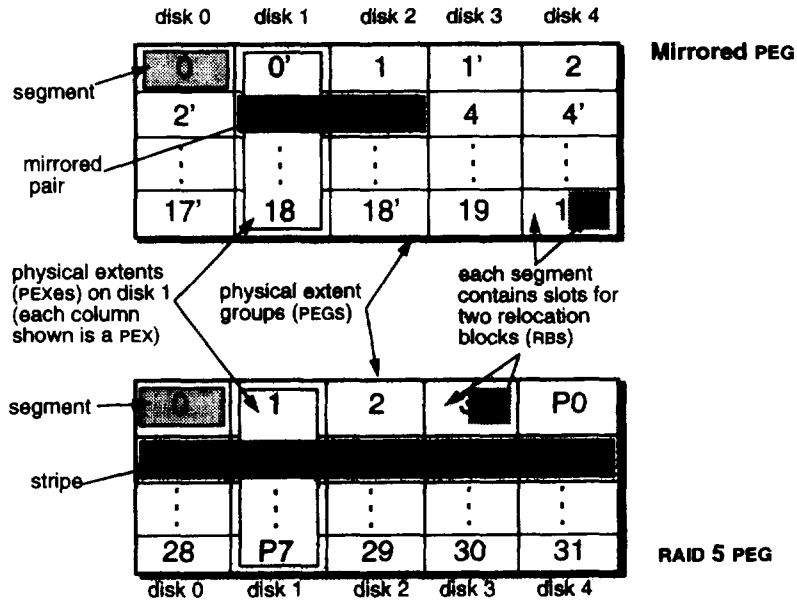


Fig. 4. Layout of two PEGs: one mirrored and one RAID 5. Each PEG is spread out across five disks. The RAID 5 PEG uses segments from all five disks to assemble each of its stripes; the mirrored PEG uses segments from two disks to form mirrored pairs.

be of different sizes, this allocation process may leave uneven amounts of free space on different disks.

Segments are the units of contiguous space on a disk that are included in a stripe or mirrored pair; each PEX is divided into a set of 128KB segments. As Figure 4 shows, mirrored and RAID 5 PEGS are divided into segments in exactly the same way, but the segments are logically grouped and used by the storage classes in different ways: in RAID 5, a segment is the stripe unit; in the mirrored storage class, a segment is the unit of duplication.

2.2.2 *Logical Data Layout*: *RBs*. The logical space provided by the array —that visible to its clients—is divided into relatively small 64KB units called *Relocation Blocks* (RBs). These are the basic units of migration in the system. When a LUN is created or is increased in size, its address space is mapped onto a set of RBs. An RB is not assigned space in a particular PEG until the host issues a write to a LUN address that maps to the RB.

The size of an RB is a compromise between data layout, data migration, and data access costs. Smaller RBs require more mapping information to record where they have been put and increase the time spent on disk seek and rotational delays. Larger RBs will increase migration costs if only small amounts of data are updated in each RB. We report on our exploration of the relationship between RB size and performance in Section 4.1.2.

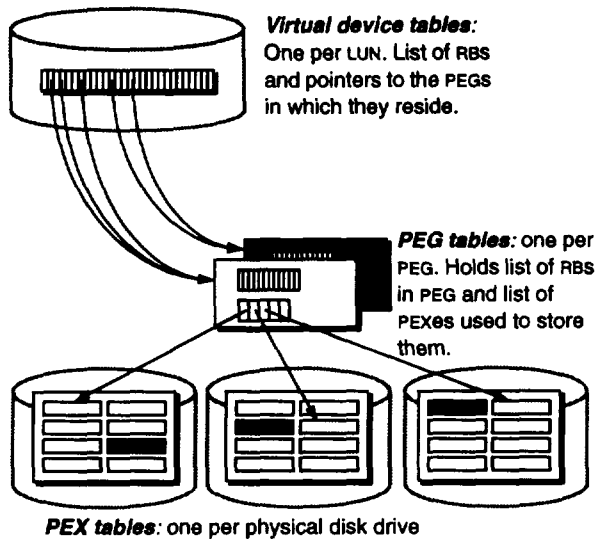


Fig. 5. Structure of the tables that map from addresses in virtual volumes to PEGs, PEXes, and physical disk addresses (simplified).

Each PEG can hold many RBs, the exact number being a function of the PEG's size and its storage class. Unused RB slots in a PEG are marked free until they have an RB (i.e., data) allocated to them.

2.2.3 *Mapping Structures.* A subset of the overall mapping structures is shown in Figure 5. These data structures are optimized for looking up the physical disk address of an RB, given its logical (LUN-relative) address, since that is the most common operation. In addition, data are held about access times and history, the amount of free space in each PEG (for cleaning and garbage collection purposes), and various other statistics. Not shown are various back pointers that allow additional scans.

2.3 Normal Operations

To start a host-initiated read or write operation, the host sends an SCSI Command Descriptor Block (CDB) to the HP AutoRAID array, where it is parsed by the controller. Up to 32 CDBs may be active at a time. An additional 2048 CDBs may be held in a FIFO queue waiting to be serviced; above this limit, requests are queued in the host. Long requests are broken up into 64KB pieces, which are handled sequentially; this method limits the amount of controller resources a single I/O can consume at minimal performance cost.

If the request is a read, and the data are completely in the controller's cache memories, the data are transferred to the host via the speed-matching buffer, and the command then completes once various statistics have been

updated. Otherwise, space is allocated in the front-end buffer cache, and one or more read requests are dispatched to the back-end storage classes.

Writes are handled slightly differently, because the nonvolatile front-end write buffer (NVRAM) allows the host to consider the request complete as soon as a copy of the data has been made in this memory. First a check is made to see if any cached data need invalidating, and then space is allocated in the NVRAM. This allocation may have to wait until space is available; in doing so, it will usually trigger a flush of existing dirty data to a back-end storage class. The data are transferred into the NVRAM from the host, and the host is then told that the request is complete. Depending on the NVRAM cache-flushing policy, a back-end write may be initiated at this point. More often, nothing is done, in the hope that another subsequent write can be coalesced with this one to increase efficiency.

Flushing data to a back-end storage class simply causes a back-end write of the data if they are already in the mirrored storage class. Otherwise, the flush will usually trigger a promotion of the RB from RAID 5 to mirrored. (There are a few exceptions that we describe later.) This promotion is done by calling the migration code, which allocates space in the mirrored storage class and copies the RB from RAID 5. If there is no space in the mirrored storage class (because the background daemons have not had a chance to run, for example), this may in turn provoke a demotion of some mirrored data down to RAID 5. There are some tricky details involved in ensuring that this cannot in turn fail—in brief, the free-space management policies must anticipate the worst-case sequence of such events that can arise in practice.

2.3.1 *Mirrored Reads and Writes.* Reads and writes to the mirrored storage class are straightforward: a read call picks one of the copies and issues a request to the associated disk. A write call causes writes to two disks; it returns only when both copies have been updated. Note that this is a back-end write call that is issued to flush data from the NVRAM and is not synchronous with the host write.

2.3.2 *RAID 5 Reads and Writes.* Back-end reads to the RAID 5 storage class are as simple as for the mirrored storage class: in the normal case, a read is issued to the disk that holds the data. In the recovery case, the data may have to be reconstructed from the other blocks in the same stripe. (The usual RAID 5 recovery algorithms are followed in this case, so we will not discuss the failure case more in this article. Although they are not implemented in the current system, techniques such as parity declustering [Holland and Gibson 1992] could be used to improve recovery-mode performance.)

Back-end RAID 5 writes are rather more complicated, however. RAID 5 storage is laid out as a log: that is, freshly demoted RBs are appended to the end of a "current RAID 5 write PEG," overwriting virgin storage there. Such writes can be done in one of two ways: per-RB writes or batched writes. The former are simpler, the latter more efficient.

—For *per-RB writes*, as soon as an RB is ready to be written, it is flushed to disk. Doing so causes a copy of its contents to flow past the parity-

calculation logic, which XORs it with its previous contents—the parity for this stripe. Once the data have been written, the parity can also be written. The prior contents of the parity block are stored in nonvolatile memory during this process to protect against power failure. With this scheme, each data-RB write causes two disk writes: one for the data and one for the parity RB. This scheme has the advantage of simplicity, at the cost of slightly worse performance.

—For *batched writes*, the parity is written only after all the data RBs in a stripe have been written, or at the end of a batch. If, at the beginning of a batched write, there are already valid data in the PEG being written, the prior contents of the parity block are copied to nonvolatile memory along with the index of the highest-numbered RB in the PEG that contains valid data. (The parity was calculated by XORing only RBs with indices less than or equal to this value.) RBs are then written to the data portion of the stripe until the end of the stripe is reached or until the batch completes; at that point the parity is written. The new parity is computed on-the-fly by the parity-calculation logic as each data RB is being written. If the batched write fails to complete for any reason, the system is returned to its prebatch state by restoring the old parity and RB index, and the write is retried using the per-RB method.

Batched writes require a bit more coordination than per-RB writes, but require only one additional parity write for each full stripe of data that is written. Most RAID 5 writes are batched writes.

In addition to these logging write methods, the method typically used in nonlogging RAID 5 implementations (*read-modify-write*) is also used in some cases. This method, which reads old data and parity, modifies them, and rewrites them to disk, is used to allow forward progress in rare cases when no PEG is available for use by the logging write processes. It is also used when it is better to update data (or holes—see Section 2.4.1) in place in RAID 5 than to migrate an RB into mirrored storage, such as in background migrations when the array is idle.

2.4 Background Operations

In addition to the foreground activities described above, the HP AutoRAID array controller executes many background activities such as garbage collection and layout balancing. These background algorithms attempt to provide "slack" in the resources needed by foreground operations so that the foreground never has to trigger a synchronous version of these background tasks, which can dramatically reduce performance.

The background operations are triggered when the array has been "idle" for a period of time. "Idleness" is defined by an algorithm that looks at current and past device activity—the array does not have to be completely devoid of activity. When an idle period is detected, the array performs one set of background operations. Each subsequent idle period, or continuation of the current one, triggers another set of operations.

After a long period of array activity, the current algorithm may need a moderate amount of time to detect that the array is idle. We hope to apply some of the results from Golding et al. [1995] to improve idle-period detection and prediction accuracy, which will in turn allow us to be more aggressive about executing background algorithms.

2.4.1 *Compaction: Cleaning and Hole-Plugging.* The mirrored storage class acquires *holes*, or empty RB slots, when RBs are demoted to the RAID 5 storage class. (Since updates to mirrored RBs are written in place, they generate no holes.) These holes are added to a free list in the mirrored storage class and may subsequently be used to contain promoted or newly created RBs. If a new PEG is needed for the RAID 5 storage class, and no free PEXes are available, a mirrored PEG may be chosen for *cleaning*: all the data are migrated out to fill holes in other mirrored PEGs, after which the PEG can be reclaimed and reallocated to the RAID 5 storage class.

Similarly, the RAID 5 storage class acquires holes when RBs are promoted to the mirrored storage class, usually because the RBs have been updated. Because the normal RAID 5 write process uses logging, the holes cannot be reused directly; we call them *garbage*, and the array needs to perform a periodic *garbage collection* to eliminate them.

If the RAID 5 PEG containing the holes is almost full, the array performs *hole-plugging* garbage collection. RBs are copied from a PEG with a small number of RBs and used to fill in the holes of an almost full PEG. This minimizes data movement if there is a spread of fullness across the PEGs, which is often the case.

If the PEG containing the holes is almost empty, and there are no other holes to be plugged, the array does PEG *cleaning*: that is, it appends the remaining valid RBs to the current end of the RAID 5 write log and reclaims the complete PEG as a unit.

2.4.2 *Migration: Moving RBs Between Levels.* A background migration policy is run to move RBs from mirrored storage to RAID 5. This is done primarily to provide enough empty RB slots in the mirrored storage class to handle a future write burst. As Ruemmler and Wilkes [1993] showed, such bursts are quite common.

RBs are selected for migration by an approximate Least Recently Written algorithm. Migrations are performed in the background until the number of free RB slots in the mirrored storage class or free PEGs exceeds a high-water mark that is chosen to allow the system to handle a burst of incoming data. This threshold can be set to provide better burst-handling at the cost of slightly lower out-of-burst performance. The current AutoRAID firmware uses a fixed value, but the value could also be determined dynamically.

2.4.3 *Balancing: Adjusting Data Layout Across Drives.* When new drives are added to an array, they contain no data and therefore do not contribute to the system's performance. *Balancing* is the process of migrating PEXes between disks to equalize the amount of data stored on each disk, and thereby also the request load imposed on each disk. Access histories could be

used to balance the disk load more precisely, but this is not currently done. Balancing is a background activity, performed when the system has little else to do.

Another type of imbalance results when a new drive is added to an array: newly created RAID 5 PEGs will use all of the drives in the system to provide maximum performance, but previously created RAID 5 PEGs will continue to use only the original disks. This imbalance is corrected by another low-priority background process that copies the valid data from the old PEGs to new, full-width PEGs.

2.5 Workload Logging

One of the uncertainties we faced while developing the HP AutoRAID design was the lack of a broad range of real system workloads at the disk I/O level that had been measured accurately enough for us to use in evaluating its performance.

To help remedy this in the future, the HP AutoRAID array incorporates an I/O workload logging tool. When the system is presented with a specially formatted disk, the tool records the start and stop times of every externally issued I/O request. Other events can also be recorded, if desired. The overhead of doing this is very small: the event logs are first buffered in the controller's RAM and then written out in large blocks. The result is a faithful record of everything the particular unit was asked to do, which can be used to drive simulation design studies of the kind we describe later in this article.

2.6 Management Tool

The HP AutoRAID controller maintains a set of internal statistics, such as cache utilization, I/O times, and disk utilizations. These statistics are relatively cheap to acquire and store, and yet can provide significant insight into the operation of the system.

The product team developed an off-line, inference-based management tool that uses these statistics to suggest possible configuration choices. For example, the tool is able to determine that for a particular period of high load, performance could have been improved by adding cache memory because the array controller was short of read cache. Such information allows administrators to maximize the array's performance in their environment.

3. HP AUTORAID PERFORMANCE RESULTS

A combination of prototyping and event-driven simulation was used in the development of HP AutoRAID. Most of the novel technology for HP AutoRAID is embedded in the algorithms and policies used to manage the storage hierarchy. As a result, hardware and firmware prototypes were developed concurrently with event-driven simulations that studied design choices for algorithms, policies, and parameters to those algorithms.

The primary development team was based at the product division that designed, built, and tested the prototype hardware and firmware. They were supported by a group at HP Laboratories that built a detailed simulator of

the hardware and firmware and used it to model alternative algorithm and policy choices in some depth. This organization allowed the two teams to incorporate new technology into products in the least possible time while still fully investigating alternative design choices.

In this section we present measured results from a laboratory prototype of a disk array product that embodies the HP AutoRAID technology. In Section 4 we present a set of comparative performance analyses of different algorithm and policy choices that were used to help guide the implementation of the real thing.

3.1 Experimental Setup

The baseline HP AutoRAID configuration on which we report was a 12-disk system with one controller and 24MB of controller data cache. It was connected via two fast-wide, differential SCSI adapters to an HP 9000/K400 system with one processor and 512MB of main memory running release 10.0 of the HP-UX operating system [Clegg et al. 1986]. All the drives used were 2.0GB 7200RPM Seagate ST32550 Barracudas with immediate write reporting turned off.

To calibrate the HP AutoRAID results against external systems, we also took measurements on two other disk subsystems. These measurements were taken on the same host hardware, on the same days, with the same host configurations, number of disks, and type of disks:

—*A Data General CLARiiON® Series 2000 Disk-Array Storage System Deskside Model 2300 with 64MB front-end cache.* (We refer to this system as "RAID array.") This array was chosen because it is the recommended third-party RAID array solution for one of the primary customers of the HP AutoRAID product.

  Because the CLARiiON supports only one connection to its host, only one of the K400's fast-wide, differential SCSI channels was used. The single channel was not, however, the bottleneck of the system. The array was configured to use RAID 5. (Results for RAID 3 were never better than for RAID 5.)

—*A set of directly connected individual disk drives.* This solution provides no data redundancy at all. The HP-UX Logical Volume Manager (LVM) was used to stripe data across these disks in 4MB chunks. Unlike HP AutoRAID and the RAID array, the disks had no central controller and therefore no controller-level cache. We refer to this configuration as "JBOD-LVM" (Just a Bunch Of Disks).

3.2 Performance Results

We begin by presenting some database macrobenchmarks in order to demonstrate that HP AutoRAID provides excellent performance for real-world workloads. Such workloads often exhibit behaviors such as burstiness that are not present in simple I/O rate tests; relying only on the latter can provide a misleading impression of how a system will behave in real use.

3.2.1 *Macrobenchmarks.* An OLTP database workload made up of medium-weight transactions was run against the HP AutoRAID array, the regular RAID array, and JBOD-LVM. The database used in this test was 6.7GB, which allowed it to fit entirely in mirrored storage in the HP AutoRAID; working-set sizes larger than available mirrored space are discussed below. For this benchmark, (1) the RAID array's 12 disks were spread evenly across its 5 SCSI channels, (2) the 64MB cache was enabled, (3) the cache page size was set to 2KB (the optimal value for this workload), and (4) the default 64KB stripe-unit size was used. Figure 6(a) shows the result: HP AutoRAID significantly outperforms the RAID array and has performance about three-fourths that of JBOD-LVM. These results suggest that the HP AutoRAID is performing much as expected: keeping the data in mirrored storage means that writes are faster than the RAID array, but not as fast as JBOD-LVM. Presumably, reads are being handled about equally well by all the cases.

Figure 6(b) shows HP AutoRAID's performance when data must be migrated between mirrored storage and RAID 5 because the working set is too large to be contained entirely in the mirrored storage class. The same type of OLTP database workload as described above was used, but the database size was set to 8.1GB. This would not fit in a 5-drive HP AutoRAID system, so we started with a 6-drive system as the baseline. Mirrored storage was able to accommodate one-third of the database in this case, two-thirds in the 7-drive system, almost all in the 8-drive system, and all of it in larger systems.

The differences in performance between the 6-, 7-, and 8-drive systems were due primarily to differences in the number of migrations performed, while the differences in the larger systems result from having more spindles across which to spread the same amount of mirrored data. The 12-drive configuration was limited by the host K400's CPU speed and performed about the same as the 11-drive system. From these data we see that even for this database workload, which has a fairly random access pattern across a large data set, HP AutoRAID performs within a factor of two of its optimum when only one-third of the data is held in mirrored storage and at about three-fourths of its optimum when two-thirds of the data are mirrored.

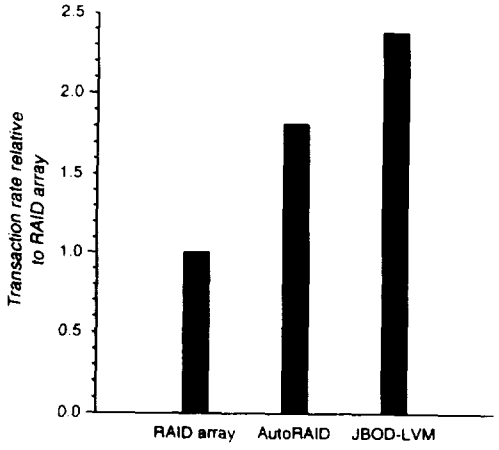
3.2.2 *Microbenchmarks.* In addition to the database macrobenchmark, we also ran some microbenchmarks that used a synthetic workload generation program known as DB to drive the arrays to saturation; the working-set size for the random tests was 2GB. These measurements were taken under slightly different conditions from the ones reported in Section 3.1:

—The HP AutoRAID contained 16MB of controller data cache.

—An HP 9000/897 was the host for all the tests.

—A single fast-wide, differential SCSI channel was used for the HP AutoRAID and RAID array tests.

—The JBOD case did not use LVM, so it did not do any striping. (Given the nature of the workload, this was probably immaterial.) In addition, 11 JBOD disks were used rather than 12 to match the amount of space available for data in the other configurations. Finally, the JBOD test used



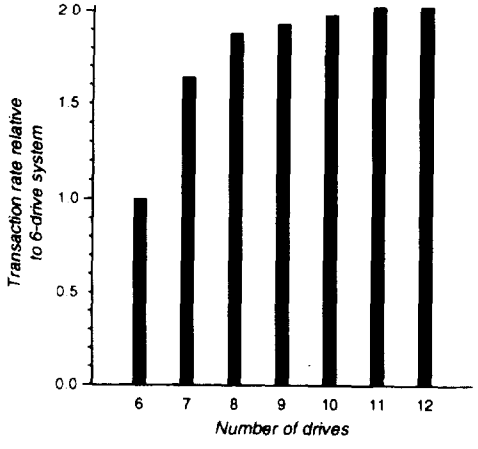


Fig. 6. OLTP macrobenchmark results; (a) comparison of HP AutoRAID and non-RAID drives with a regular RAID array. Each system used 12 drives, and the entire 6.7GB database fit in mirrored storage in HP AutoRAID; (b) performance of HP AutoRAID when different numbers of drives are used. The fraction of the 8.1GB database held in mirrored storage was: 1/3 in the 6-drive system, 2/3 in the 7-drive system, nearly all in the 8-drive system, and all in the larger systems.

a fast-wide, single-ended SCSI card that required more host CPU cycles per I/O. We believe that this did not affect the microbenchmarks because they were not CPU limited.

—The RAID array used 8KB cache pages and cache on or off as noted.

Data from the microbenchmarks are provided in Figure 7. This shows the relative performance of the two arrays and JBOD for random and sequential reads and writes.

The random 8KB read-throughput test is primarily a measure of controller overheads. HP AutoRAID performance is roughly midway between the RAID array with its cache disabled and JBOD. It would seem that the cache-searching algorithm of the RAID array is significantly limiting its performance, given that the cache hit rate would have been close to zero in these tests.

The random 8KB write-throughput test is primarily a test of the low-level storage system used, since the systems are being driven into a disk-limited

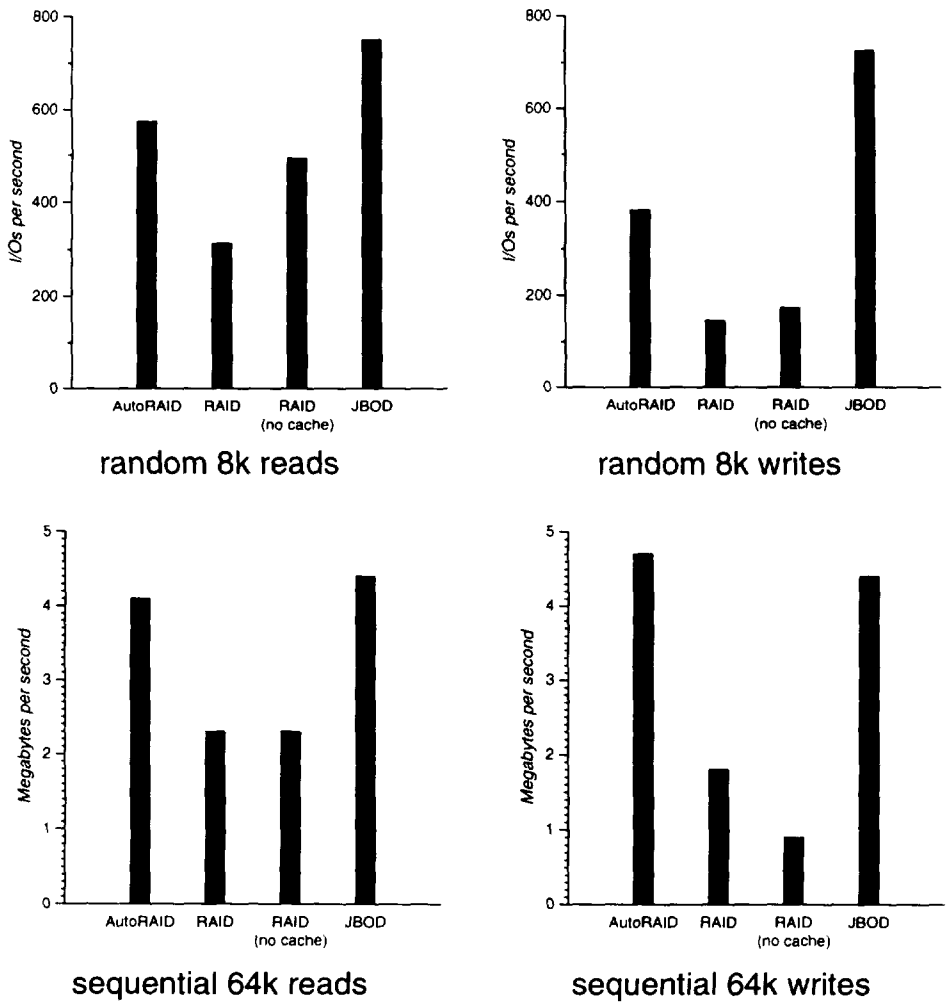


Fig. 7. Microbenchmark comparisons of HP AutoRAID, a regular RAID array, and non-RAID drives.

behavior by the benchmark. As expected, there is about a 1:2:4 ratio in I/Os per second for RAID 5 (4 I/Os for a small update): HP AutoRAID (2 I/Os to mirrored storage): JBOD (1 write in place).

The sequential 64KB read-bandwidth test shows that the use of mirrored storage in HP AutoRAID can largely compensate for controller overhead and deliver performance comparable to that of JBOD.

Finally, the sequential 64KB write-bandwidth test illustrates HP AutoRAID's ability to stream data to disk through its NVRAM cache: its performance is better than the pure JBOD solution.

We do not have a good explanation for the relatively poor performance of the RAID array in the last two cases; the results shown are the best obtained

from a number of different array configurations. Indeed, the results demonstrated the difficulties involved in properly configuring a RAID array: many parameters were adjusted (caching on or off, cache granularity, stripe depth, and data layout), and no single combination performed well across the range of workloads examined.

3.2.3 *Thrashing.* As we noted in Section 1.1, the performance of HP AutoRAID depends on the working-set size of the applied workload. With the working set within the size of the mirrored space, performance is very good, as shown by Figure 6(a) and Figure 7. And as Figure 6(b) shows, good performance can also be obtained when the entire working set does not fit in mirrored storage.

If the active write working set exceeds the size of mirrored storage for long periods of time, however, it is possible to drive the HP AutoRAID array into a *thrashing* mode in which each update causes the target RB to be promoted up to the mirrored storage class and a second one demoted to RAID 5. An HP AutoRAID array can usually be configured to avoid this by adding enough disks to keep all the write-active data in mirrored storage. If *all* the data were write active, the cost-performance advantages of the technology would, of course, be reduced. Fortunately, it is fairly easy to predict or detect the environments that have a large write working set and to avoid them if necessary. If thrashing does occur, HP AutoRAID detects it and reverts to a mode in which it writes directly to RAID 5—that is, it automatically adjusts its behavior so that performance is no worse than that of RAID 5.

4. SIMULATION STUDIES

In this section, we will illustrate several design choices that were made inside the HP AutoRAID implementation using a trace-driven simulation study.

Our simulator is built on the Pantheon [Cao et al. 1994; Golding et al. 1994] simulation framework,[1] which is a detailed, trace-driven simulation environment written in C++. Individual simulations are configured from the set of available C++ simulation objects using scripts written in the Tcl language [Ousterhout 1994] and configuration techniques described in Golding et al. [1994]. The disk models used in the simulation are improved versions of the detailed, calibrated models described in Ruemmler and Wilkes [1994].

The traces used to drive the simulations are from a variety of systems, including: cello, a time-sharing HP 9000 Series 800 HP-UX system; snake, an HP 9000 Series 700 HP-UX cluster file server; OLTP, an HP 9000 Series 800 HP-UX system running a database benchmark made up of medium-weight transactions (not the system described in Section 3.1); hplajw, a personal workstation; and a Netware server. We also used subsets of these traces, such as the /usr disk from cello, a subset of the database disks from OLTP, and the OLTP log disk. Some of them were for long time periods (up to three months),

[1]The simulator was formerly called TickerTAIP, but we have changed its name to avoid confusion with the parallel RAID array project of the same name [Cao et al. 1994].

although most of our simulation runs used two-day subsets of the traces. All but the Netware trace contained detailed timing information to 1μs resolution. Several of them are described in considerable detail in Ruemmler and Wilkes [1993].

We modeled the hardware of HP AutoRAID using Pantheon components (caches, buses, disks, etc.) and wrote detailed models of the basic firmware and of several alternative algorithms or policies for each of about 40 design experiments. The Pantheon simulation core comprises about 46k lines of C++ and 8k lines of Tcl, and the HP-AutoRAID-specific portions of the simulator added another 16k lines of C++ and 3k lines of Tcl.

Because of the complexity of the model and the number of parameters, algorithms, and policies that we were examining, it was impossible to explore all combinations of the experimental variables in a reasonable amount of time. We chose instead to organize our experiments into baseline runs and runs with one or a few related changes to the baseline. This allowed us to observe the performance effects of individual or closely related changes and to perform a wide range of experiments reasonably quickly. (We used a cluster of 12 high-performance workstations to run the simulations; even so, executing all of our experiments took about a week of elapsed time.)

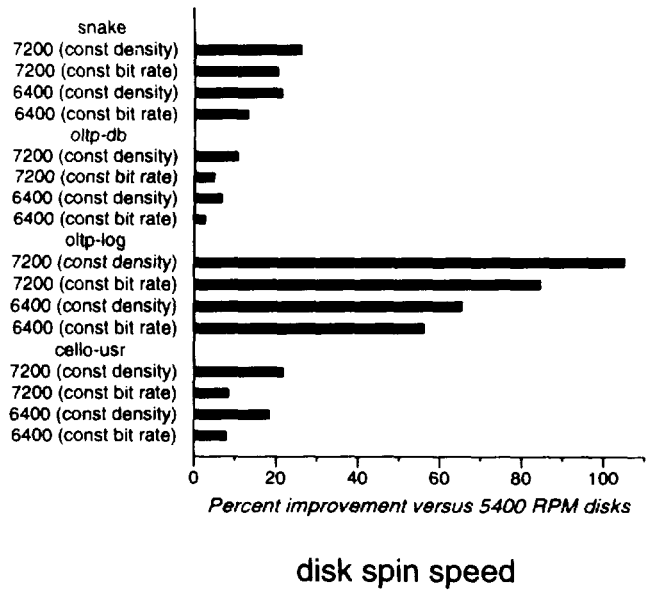
We performed additional experiments to combine individual changes that we suspected might strongly interact (either positively or negatively) and to test the aggregate effect of a set of algorithms that we were proposing to the product development team.

No hardware implementation of HP AutoRAID was available early in the simulation study, so we were initially unable to calibrate our simulator (except for the disk models). Because of the high level of detail of the simulation, however, we were confident that *relative* performance differences predicted by the simulator would be valid even if *absolute* performance numbers were not yet calibrated. We therefore used the relative performance differences we observed in simulation experiments to suggest improvements to the team implementing the product firmware, and these are what we present here. In turn, we updated our baseline model to correspond to the changes they made to their implementation.

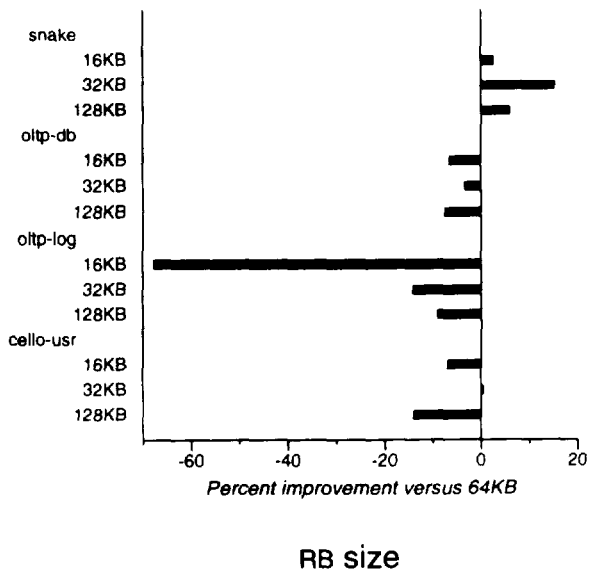
Since there are far too many individual results to report here, we have chosen to describe a few that highlight some of the particular behaviors of the HP AutoRAID system.

4.1 Disk Speed

Several experiments measured the sensitivity of the design to the size or performance of various components. For example, we wanted to understand whether faster disks would be cost effective. The baseline disks held 2GB and spun at 5400 RPM. We evaluated four variations of this disk: spinning at 6400 RPM and 7200 RPM, keeping either the data density (bits per inch) or transfer rate (bits per second) constant. As expected, increasing the back-end disk performance generally improves overall performance, as shown in Figure 8(a). The results suggest that improving transfer rate is more important than improving rotational latency.



(a)



(b)

Fig. 8. Effects of (a) disk spin speed and (b) RB size on performance.

4.2 RB Size

The standard AutoRAID system uses 64KB RBs as the basic storage unit. We looked at the effect of using smaller and larger sizes. For most of the workloads (see Figure 8(b)), the 64KB size was the best of the ones we tried: the balance between seek and rotational overheads versus data movement costs is about right. (This is perhaps not too surprising: the disks we are using have track sizes of around 64KB, and transfer sizes in this range will tend to get much of the benefit from fewer mechanical delays.)

4.3 Data Layout

Since the system allows blocks to be remapped, blocks that the host system has tried to lay out sequentially will often be physically discontiguous. To see how bad this problem could get, we compared the performance of the system when host LUN address spaces were initially laid out completely linearly on disk (as a best case) and completely randomly (as a worst case). Figure 9(a) shows the difference between the two layouts: there is a modest improvement in performance in the linear case compared with the random one. This suggests that the RB size is large enough to limit the impact of seek delays for sequential accesses.

4.4 Mirrored Storage Class Read Selection Algorithm

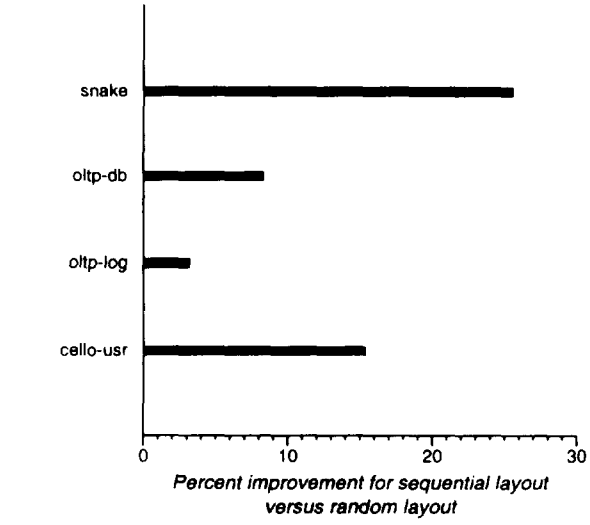
When the front-end read cache misses on an RB that is stored in the mirrored storage class, the array can choose to read either of the stored copies. The baseline system selects the copy at random in an attempt to avoid making one disk a bottleneck. However, there are several other possibilities:

—strictly alternating between disks (alternate);
—attempting to keep the heads on some disks near the outer edge while keeping others near the inside (inner/outer);
—using the disk with the shortest queue (shortest queue);
—using the disk that can reach the block first, as determined by a shortest-positioning-time algorithm [Jacobson and Wilkes 1991; Seltzer et al. 1990] (shortest seek).

Further, the policies can be "stacked," using first the most aggressive policy but falling back to another to break a tie. In our experiments, random was always the final fallback policy.

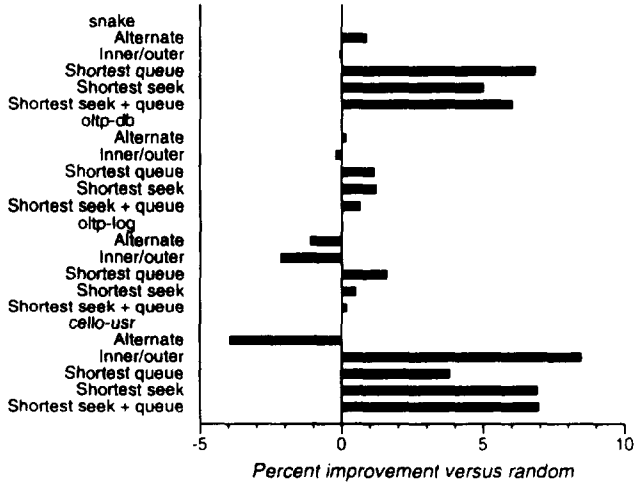
Figure 9(b) shows the results of our investigations into the possibilities. By using shortest queue as a simple load-balancing heuristic, performance is improved by an average of 3.3% over random for these workloads. Shortest seek performed 3.4% better than random on the average, but it is much more complex to implement because it requires detailed knowledge of disk head position and seek timing.

Static algorithms such as alternate and inner/outer sometimes perform better than random, but sometimes interact unfavorably with patterns in the workload and decrease system performance.



data layout

(a)



read disk selection policy

(b)

Fig. 9. Effects of (a) data layout and (b) mirrored storage class read disk selection policy on performance.

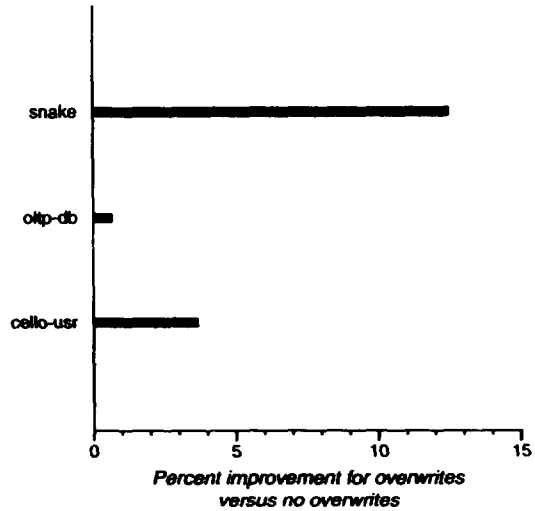


Fig. 10. Effect of allowing write cache overwrites on performance.

We note in passing that these differences do not show up under microbenchmarks (of the type reported in Figure 7) because the disks are typically always driven to saturation and do not allow such effects to show through.

4.5 Write Cache Overwrites

We investigated several policy choices for managing the NVRAM write cache. The baseline system, for instance, did not allow one write operation to overwrite dirty data already in cache; instead, the second write would block until the previous dirty data in the cache had been flushed to disk. As Figure 10 shows, allowing overwrites had a noticeable impact on most of the workloads. It had a huge impact on the OLTP-log workload, improving its performance by a factor of 5.3. We omitted this workload from the graph for scaling reasons.

4.6 Hole-Plugging During RB Demotion

RBs are typically written to RAID 5 for one of two reasons: demotion from mirrored storage or for garbage collection. During normal operation, the system creates holes in RAID 5 by promoting RBs to the mirrored storage class. In order to keep space consumption constant, the system later demotes (other) RBs to RAID 5. In the default configuration, HP AutoRAID uses logging writes to demote RBs to RAID 5 quickly, even if the demotion is done during idle time; these demotions do not fill the holes left by the promoted RBs. To reduce the work done by the RAID 5 cleaner, we allowed RBs demoted during idle periods to be written to RAID 5 using hole-plugging. This optimization reduced the number of RBs moved by the RAID 5 cleaner by

93% for the cello-usr workload and by 96% for snake, and improved mean I/O time for user I/Os by 8.4% and 3.2%.

5. SUMMARY

The HP AutoRAID technology works extremely well, providing performance close to that of a nonredundant disk array across many workloads. At the same time, it provides full data redundancy and can tolerate failures of any single array component.

It is very easy to use: one of the authors of this article was delivered a system without manuals a day before a demonstration and had it running a trial benchmark five minutes after getting it connected to his completely unmodified workstation. The product team has had several such experiences in demonstrating the system to potential customers.

The HP AutoRAID technology is not a panacea for all storage problems: there are workloads that do not suit its algorithms well and environments where the variability in response time is unacceptable. Nonetheless, it is able to adapt to a great many of the environments that are encountered in real life, and it provides an outstanding general-purpose storage solution where availability matters.

The first product based on the technology, the HP XLR1200 Advanced Disk Array, is now available.

ACKNOWLEDGMENTS

We would like to thank our colleagues in HP's Storage Systems Division. They developed the HP AutoRAID system architecture and the product version of the controller and were the customers for our performance and algorithm studies. Many more people put enormous amounts of effort into making this program a success than we can possibly acknowledge directly by name; we thank them all.

Chris Ruemmler wrote the DB benchmark used for the results in Section 3.2.

This article is dedicated to the memory of our late colleague Al Kondoff, who helped establish the collaboration that produced this body of work.

REFERENCES

AKYÜREK, S. AND SALEM, K. 1993. Adaptive block rearrangement. Tech. Rep. CS-TR-2854.1, Dept. of Computer Science, Univ. of Maryland, College Park, Md.

BAKER, M., ASAMI, S., DEPRIT, E., OUSTERHOUT, J., AND SELTZER, M. 1992. Non-volatile memory for fast, reliable file systems. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems. Comput. Arch. News 20*, (Oct.), 10–22.

BLACKWELL, T., HARRIS, J., AND SELTZER, M. 1995. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems*. USENIX Assoc., Berkeley, Calif., 277–288.

BURKES, T., DIAMOND, B., AND VOIGT, D. 1995. Adaptive hierarchical RAID: A solution to the RAID 5 write problem. Part No. 5963-9151, Hewlett-Packard Storage Systems Division, Boise, Idaho.

BURROWS, M., JERIAN, C., LAMPSON, B., AND MANN, T. 1992. On-line data compression in a log-structured file system. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems. Comput. Arch News 20*, (Oct.), 2–9.

CAO, P., LIM, S. B., VENKATARAMAN, S., AND WILKES, J. 1994. The TickerTAIP parallel RAID architecture. *ACM Trans. Comput. Syst. 12*, 3 (Aug.), 236–269.

CARSON, S. AND SETIA, S. 1992. Optimal write batch size in log-structured file systems. In *USENIX Workshop on File Systems*. USENIX Assoc., Berkeley, Calif., 79–91.

CATE, V. 1990. Two levels of file system hierarchy on one disk. Tech. Rep. CMU-CS-90-129, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

CHAO, C., ENGLISH, R., JACOBSON, D., STEPANOV, A., AND WILKES, J. 1992. Mime: A high performance storage device with strong recovery guarantees. Tech. Rep. HPL-92-44, Hewlett-Packard Laboratories, Palo Alto, Calif.

CHEN, P. 1973. Optimal file allocation in multi-level storage hierarchies. In *Proceedings of National Computer Conference and Exposition*. AFIPS Conference Proceedings, vol. 42. AFIPS Press, Montvale, N.J., 277–282.

CHEN, P. M. AND LEE, E. K. 1993. Striping in a RAID level-5 disk array. Tech. Rep. CSE-TR-181-93, The Univ. of Michigan, Ann Arbor, Mich.

CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. 1994. RAID: High-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (June), 145–185.

CLEGG, F. W., HO, G. S.-F., KUSMER, S. R., AND SONTAG, J. R. 1986. The HP-UX operating system on HP Precision Architecture computers. *Hewlett-Packard J. 37*, 12 (Dec.), 4–22.

COHEN, E. I., KING, G. M., AND BRADY, J. T. 1989. Storage hierarchies. *IBM Syst. J. 28*, 1, 62–76.

DEC. 1993. *POLYCENTER Storage Management for OpenVMS VAX Systems*. Digital Equipment Corp., Maynard, Mass.

DE JONGE, W., KAASHOEK, M. F., AND HSIEH, W. C. 1993. The Logical Disk: A new approach to improving file systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, New York, 15–28.

DESHPANDE, M. B. AND BUNT, R. B. 1988. Dynamic file management techniques. In *Proceedings of the 7th IEEE Phoenix Conference on Computers and Communication*. IEEE, New York, 86–92.

DUNPHY, R. H., JR., WALSH, R., BOWERS, J. H., AND RUDESEAL, G. A. 1991. Disk drive memory. U.S. Patent 5,077,736, U.S. Patent Office, Washington, D.C.

ENGLISH, R. M. AND STEPANOV, A. A. 1992. Loge: A self-organizing storage device. In *Proceedings of USENIX Winter '92 Technical Conference*. USENIX Assoc., Berkeley, Calif., 237–251.

EPOCH SYSTEMS. 1988. Mass storage: Server puts optical discs on line for workstations. *Electronics* (Nov.).

EWING, J. 1993. RAID: An overview. Part No. W 17004-A 09/93, Storage Technology Corp., Louisville, Colo. Available as http://www.stortek.com:80/StorageTek/raid.html.

FLOYD, R. A. AND SCHLATTER ELLIS, C. 1989. Directory reference patterns in hierarchical file systems. *IEEE Trans. Knowl. Data Eng. 1*, 2 (June), 238–247.

GEIST, R., REYNOLDS, R., AND SUGGS, D. 1994. Minimizing mean seek distance in mirrored disk systems by cylinder remapping. *Perf. Eval. 20*, 1–3 (May), 97–114.

GELB, J. P. 1989. System managed storage. *IBM Syst. J. 28*, 1, 77–103.

GOLDING, R., STAELIN, C., SULLIVAN, T., AND WILKES, J. 1994. "Tcl cures 98.3% of all known simulation configuration problems" claims astonished researcher! In *Proceedings of Tcl/Tk Workshop*. Available as Tech. Rep. HPL-CCD-94-11, Concurrent Computing Dept., Hewlett-Packard Laboratories, Palo Alto, Calif.

GOLDING, R., BOSCH, P., STAELIN, C., SULLIVAN, T., AND WILKES, J. 1995. Idleness is not sloth. In *Proceedings of USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems*. USENIX Assoc., Berkeley, Calif., 201–212.

GRAY, J. 1990. A census of Tandem system availability between 1985 and 1990. Tech. Rep. 90.1, Tandem Computers Inc., Cupertino, Calif.

HENDERSON, R. L. AND POSTON, A. 1989. MSS-II and RASH: A mainframe Unix based mass storage system with a rapid access storage hierarchy file management system. In *Proceedings of USENIX Winter 1989 Conference*. USENIX Assoc., Berkeley, Calif., 65–84.

HOLLAND, M. AND GIBSON, G. A. 1992. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems. Comput. Arch. News 20*, (Oct.), 23–35.

JACOBSON, D. M. AND WILKES, J. 1991. Disk scheduling algorithms based on rotational position. Tech. Rep. HPL-CSP-91-7, Hewlett-Packard Laboratories, Palo Alto, Calif.

KATZ, R. H., ANDERSON, T. E., OUSTERHOUT, J. K., AND PATTERSON, D. A. 1991. Robo-line storage: Low-latency, high capacity storage systems over geographically distributed networks. UCB/CSD 91/651, Computer Science Div., Dept. of Electrical Engineering and Computer Science, Univ. of California at Berkeley, Berkeley, Calif.

KOHL, J. T., STAELIN, C., AND STONEBRAKER, M. 1993. Highlight: Using a log-structured file system for tertiary storage management. In *Proceedings of Winter 1993 USENIX*. USENIX Assoc., Berkeley, Calif., 435–447.

LAWLOR, F. D. 1981. Efficient mass storage parity recovery mechanism. *IBM Tech. Discl. Bull. 24*, 2 (July), 986–987.

MAJUMDAR, S. 1984. Locality and file referencing behaviour: Principles and applications. M.Sc. thesis, Tech. Rep. 84-14, Dept. of Computer Science, Univ. of Saskatchewan, Saskatoon, Saskatchewan, Canada.

MCDONALD, M. S. AND BUNT, R. B. 1989. Improving file system performance by dynamically restructuring disk space. In *Proceedings of Phoenix Conference on Computers and Communication*. IEEE, New York, 264–269

MCNUTT, B. 1994. Background data movement in a log-structured disk subsystem. *IBM J. Res. Devel. 38*, 1, 47–58.

MENON, J. AND COURTNEY, J. 1993. The architecture of a fault-tolerant cached RAID controller. In *Proceedings of 20th International Symposium on Computer Architecture*. ACM, New York, 76–86.

MENON, J. AND KASSON, J. 1989. Methods for improved update performance of disk arrays. Tech. Rep. RJ 6928 (66034), IBM Almaden Research Center, San Jose, Calif. Declassified Nov. 21, 1990.

MENON, J. AND KASSON, J. 1992. Methods for improved update performance of disk arrays. In *Proceedings of 25th International Conference on System Sciences*. Vol. 1. IEEE, New York, 74–83.

MENON, J. AND MATTSON, D. 1992. Comparison of sparing alternatives for disk arrays. In *Proceedings of 19th International Symposium on Computer Architecture*. ACM, New York, 318–329.

MILLER, E. L. 1991. File migration on the Cray Y-MP at the National Center for Atmospheric Research. UCB/CSD 91/638, Computer Science Div., Dept. of Electrical Engineering and Computer Science, Univ. of California at Berkeley, Berkeley, Calif.

MISRA, P. N. 1981. Capacity analysis of the mass storage system. *IBM Syst. J. 20*, 3, 346–361.

MOGI, K. AND KITSUREGAWA, M. 1994. Dynamic parity stripe reorganizations for RAID5 disk arrays. In *Proceedings of Parallel and Distributed Information Systems International Conference*. IEEE, New York, 17–26.

OUSTERHOUT, J. AND DOUGLIS, F. 1989. Beating the I/O bottleneck: A case for log-structured file systems. *Oper. Syst. Rev. 23*, 1 (Jan.), 11–27.

OUSTERHOUT, J. K. 1994. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Mass.

PARK, A. AND BALASUBRAMANIAN, K. 1986. Providing fault tolerance in parallel secondary storage systems. Tech. Rep. CS-TR-057-86, Dept. of Computer Science, Princeton Univ., Princeton, N.J.

PATTERSON, D. A., CHEN, P., GIBSON, G., AND KATZ, R. H. 1989. Introduction to redundant arrays of inexpensive disks (RAID). In *Spring COMPCON '89*. IEEE, New York, 112–117.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of 1988 ACM SIGMOD International Conference on Management of Data*. ACM, New York.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb.), 26–52.

RUEMMLER, C. AND WILKES, J. 1991. Disk shuffling. Tech. Rep. HPL-91-156, Hewlett-Packard Laboratories, Palo Alto, Calif.

RUEMMLER, C. AND WILKES, J. 1993. UNIX disk access patterns. In *Proceedings of the Winter 1993 USENIX Conference*. USENIX Assoc., Berkeley, Calif., 405–420.

RUEMMLER, C. AND WILKES, J. 1994. An introduction to disk drive modeling. *IEEE Comput. 27*, 3 (Mar.), 17–28.

SCSI. 1991. Draft proposed American National Standard for information systems—Small Computer System Interface-2 (SCSI-2). Draft ANSI standard X3T9.2/86-109, (revision 10d). Secretariat, Computer and Business Equipment Manufacturers Association.

SELTZER, M., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. 1993. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter 1993 USENIX Conference*. USENIX Assoc., Berkeley, Calif., 307–326.

SELTZER, M., CHEN, P., AND OUSTERHOUT, J. 1990. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Conference*. USENIX Assoc., Berkeley, Calif., 313–323.

SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. 1995. File system logging versus clustering: A performance comparison. In *Conference Proceedings of USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems*. USENIX Assoc., Berkeley, Calif., 249–264.

SIENKNECHT, T. F., FRIEDRICH, R. J., MARTINKA, J. J., AND FRIEDENBACH, P. M. 1994. The implications of distributed data in a commercial environment on the design of hierarchical storage management. *Perf. Eval. 20*, 1–3 (May), 3–25.

SMITH, A. J. 1981. Optimization of I/O systems by cache disks and file migration: A summary. *Perf. Eval. 1*, 249–262.

STK. 1995. Iceberg 9200 disk array subsystem. Storage Technology Corp., Louisville, Colo. Available as http://www.stortek.com:80/StorageTek/iceberg.html.

TAUNTON, M. 1991. Compressed executables: An exercise in thinking small. In *Proceedings of Summer USENIX*. USENIX Assoc., Berkeley, Calif., 385–403.

Received September 1995; revised October 1995; accepted October 1995

ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging

C. MOHAN
IBM Almaden Research Center
and
DON HADERLE
IBM Santa Teresa Laboratory
and
BRUCE LINDSAY, HAMID PIRAHESH and PETER SCHWARZ
IBM Almaden Research Center

In this paper we present a simple and efficient method, called ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*), which supports partial rollbacks of transactions, fine-granularity (e.g., record) locking and recovery using write-ahead logging (WAL). We introduce the paradigm of *repeating history* to redo all missing updates *before* performing the rollbacks of the loser transactions during restart after a system failure. ARIES uses a log sequence number in each page to correlate the state of a page with respect to logged updates of that page. All updates of a transaction are logged, including those performed during rollbacks. By appropriate chaining of the log records written during rollbacks to those written during forward progress, a bounded amount of logging is ensured during rollbacks even in the face of repeated failures during restart or of nested rollbacks We deal with a variety of features that are very important in building and operating an *industrial-strength* transaction processing system ARIES supports fuzzy checkpoints, selective and deferred restart, fuzzy image copies, media recovery, and high concurrency lock modes (e.g., increment/decrement) which exploit the semantics of the operations and require the ability to perform operation logging. ARIES is flexible with respect to the kinds of buffer management policies that can be implemented. It supports objects of varying length efficiently. By enabling parallelism during restart, page-oriented redo, and logical undo, it enhances concurrency and performance. We show why some of the System R paradigms for logging and recovery, which were based on the shadow page technique, need to be changed in the context of WAL. We compare ARIES to the WAL-based recovery methods of

Authors' addresses: C Mohan, Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120; D. Haderle, Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, CA 95150; B. Lindsay, H. Pirahesh, and P. Schwarz, IBM Almaden Research Center, San Jose, CA 95120.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1992 0362-5915/92/0300-0094 $1.50

DB2™, IMS, and Tandem™ systems. ARIES is applicable not only to database management systems but also to persistent object-oriented languages, recoverable file systems and transaction-based operating systems. ARIES has been implemented, to varying degrees, in IBM's OS/2™ Extended Edition Database Manager, DB2, Workstation Data Save Facility/VM, Starburst and QuickSilver, and in the University of Wisconsin's EXODUS and Gamma database machine.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability—*backup procedures, checkpoint/restart, fault tolerance*; E.5. [**Data**]: Files—*backup/recovery*; H.2.2 [**Database Management**]: Physical Design—*recovery and restart*; H.2.4 [**Database Management**]: Systems—*concurrency, transaction processing*; H.2.7 [**Database Management**]: Database Administration—*logging and recovery*

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Buffer management, latching, locking, space management, write-ahead logging

1. INTRODUCTION

In this section, first we introduce some basic concepts relating to recovery, concurrency control, and buffer management, and then we outline the organization of the rest of the paper.

1.1 Logging, Failures, and Recovery Methods

The transaction concept, which is well understood by now, has been around for a long time. It encapsulates the *ACID* (Atomicity, Consistency, Isolation and Durability) properties [36]. The application of the transaction concept is not limited to the database area [6, 17, 22, 23, 30, 39, 40, 51, 74, 88, 90, 101]. Guaranteeing the atomicity and durability of transactions, in the face of concurrent execution of multiple transactions and various failures, is a very important problem in transaction processing. While many methods have been developed in the past to deal with this problem, the assumptions, performance characteristics, and the complexity and ad hoc nature of such methods have not always been acceptable. Solutions to this problem may be judged using several metrics: degree of concurrency supported within a page and across pages, complexity of the resulting logic, space overhead on nonvolatile storage and in memory for data and the log, overhead in terms of the number of synchronous and asynchronous I/Os required during restart recovery and normal processing, kinds of functionality supported (partial transaction rollbacks, etc.), amount of processing performed during restart recovery, degree of concurrent processing supported during restart recovery, extent of system-induced transaction rollbacks caused by deadlocks, restrictions placed

™ AS/400, DB2, IBM, and OS/2 are trademarks of the International Business Machines Corp. Encompass, NonStop SQL and Tandem are trademarks of Tandem Computers, Inc. DEC, VAX DBMS, VAX and Rdb/VMS are trademarks of Digital Equipment Corp. Informix is a registered trademark of Informix Software, Inc.

on stored data (e.g., requiring unique keys for all records, restricting maximum size of objects to the page size, etc.), ability to support novel lock modes which allow the concurrent execution, based on commutativity and other properties [2, 26, 38, 45, 88, 89], of operations like increment/decrement on the same data by different transactions, and so on.

In this paper we introduce a new recovery method, called *ARIES*[1] (*Algorithm for Recovery and Isolation Exploiting Semantics*), which fares very well with respect to all these metrics. It also provides a great deal of flexibility to take advantage of some special characteristics of a class of applications for better performance (e.g., the kinds of applications that IMS Fast Path [28, 42] supports efficiently).

To meet transaction and data recovery guarantees, ARIES records in a *log* the progress of a transaction, and its actions which cause changes to recoverable data objects. The log becomes the source for ensuring either that the transaction's committed actions are reflected in the database despite various types of failures, or that its uncommitted actions are undone (i.e., rolled back). When the logged actions reflect data object content, then those log records also become the source for reconstruction of damaged or lost data (i.e., media recovery). *Conceptually*, the log can be thought of as an ever growing *sequential* file. In the actual implementation, multiple physical files may be used in a serial fashion to ease the job of archiving log records [15]. Every log record is assigned a unique *log sequence number* (*LSN*) when that record is appended to the log. The LSNs are assigned in ascending sequence. Typically, they are the *logical* addresses of the corresponding log records. At times, version numbers or timestamps are also used as LSNs [67]. If more than one log is used for storing the log records relating to *different* pieces of data, then a form of two-phase commit protocol (e.g., the current industry-standard Presumed Abort protocol [63, 64]) must be used.

The nonvolatile version of the log is stored on what is generally called *stable storage*. Stable storage means nonvolatile storage which remains intact and available across system failures. Disk is an example of nonvolatile storage and its stability is generally improved by maintaining synchronously two identical copies of the log on different devices. We would expect the online log records stored on direct access storage devices to be archived to a cheaper and slower medium like tape at regular intervals. The archived log records may be discarded once the appropriate image copies (archive dumps) of the database have been produced and those log records are no longer needed for media recovery.

Whenever log records are written, they are placed first only in the *volatile* storage (i.e., virtual storage) buffers of the log file. Only at certain times (e.g., at commit time) are the log records up to a certain point (LSN) written, in log page sequence, to stable storage. This is called *forcing* the log up to that LSN. Besides forces caused by transaction and buffer manager activi-

[1] The choice of the name ARIES, besides its use as an acronym that describes certain features of our recovery method, is also supposed to convey the relationship of our work to the Starburst project at IBM, since Aries is the name of a constellation.

ties, a system process may, in the background, periodically force the log buffers as they fill up.

For ease of exposition, we assume that each log record describes the update performed to only a single page. This is not a requirement of ARIES. In fact, in the Starburst [87] implementation of ARIES, sometimes a single log record might be written to describe updates to two pages. The *undo* (respectively, *redo*) portion of a log record provides information on how to undo (respectively, redo) changes performed by the transaction. A log record which contains both the undo and the redo information is called an *undo-redo log record*. Sometimes, a log record may be written to contain only the redo information or only the undo information. Such a record is called a *redo-only log record* or an *undo-only log record*, respectively. Depending on the action that is performed, the undo-redo information may be recorded *physically* (e.g., before the update and after the update images or values of specific fields within the object) or *operationally* (e.g., add 5 to field 3 of record 15, subtract 3 from field 4 of record 10). Operation logging permits the use of high concurrency lock modes, which exploit the semantics of the operations performed on the data. For example, with certain operations, the same field of a record could have uncommitted updates of many transactions. These permit more concurrency than what is permitted by the *strict executions* property of the model of [3], which essentially says that modified objects must be locked exclusively (X mode) for commit duration.

ARIES uses the widely accepted write ahead logging (WAL) protocol. Some of the commercial and prototype systems based on WAL are IBM's AS/400™ [9, 21], CMU's Camelot [23, 90], IBM's DB2™ [1, 10, 11, 12, 13, 14, 15, 19, 35, 96], Unisys's DMS/1100 [27], Tandem's Encompass™ [4, 37], IBM's IMS [42, 43, 53, 76, 80, 94], Informix's Informix-Turbo™ [16], Honeywell's MRDS [91], Tandem's NonStop SQL™ [95], MCC's ORION [29], IBM's OS/2 Extended Edition™ Database Manager [7], IBM's QuickSilver [40], IBM's Starburst [87], SYNAPSE [78], IBM's System/38 [99], and DEC's VAX DBMS™ and VAX Rdb/VMS™ [81]. In WAL-based systems, an updated page is written back to the same nonvolatile storage location from where it was read. That is, *in-place updating* is performed on nonvolatile storage. Contrast this with what happens in the shadow page technique which is used in systems such as System R [31] and SQL/DS [5] and which is illustrated in Figure 1. There the updated version of the page is written to a different location on nonvolatile storage and the previous version of the page is used for performing database recovery if the system were to fail before the next checkpoint.

The *WAL protocol* asserts that the log records representing changes to some data must already be on stable storage before the changed data is allowed to replace the previous version of that data on nonvolatile storage. That is, the system is not allowed to write an updated page to the nonvolatile storage version of the database until at least the undo portions of the log records which describe the updates to the page have been written to stable storage. To enable the enforcement of this protocol, systems using the WAL method of recovery store in every page the LSN of the log record that describes the most recent update performed on that page. The reader is

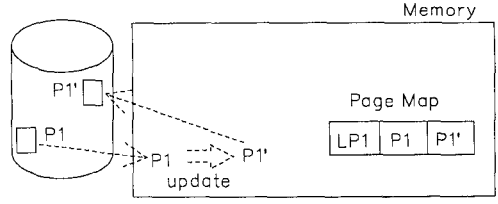


Fig. 1. Shadow page technique.

Logical page LP1 is read from physical page P1 and after modification is written to physical page P1′ P1′ is the *current* version and P1 is the *shadow* version During a checkpoint, the shadow version is discarded and the current version becomes the shadow version also On a failure, data base recovery is performed using the log and the shadow version of the data base

referred to [31, 97] for discussions about why the WAL technique is considered to be better than the shadow page technique. [16, 78] discuss methods in which shadowing is performed using a separate log. While these avoid some of the problems of the original shadow page approach, they still retain some of the important drawbacks and they introduce some new ones. Similar comments apply to the methods suggested in [82, 88]. Later, in Section 10, we show why some of the recovery paradigms of System R, which were based on the shadow page technique, are inappropriate in the WAL context, when we need support for high levels of concurrency and various other features that are described in Section 2.

Transaction status is also stored in the log and no transaction can be considered complete until its committed status and all its log data are safely recorded on stable storage by forcing the log up to the transaction's commit log record's LSN. This allows a restart recovery procedure to recover any transactions that completed successfully but whose updated pages were not physically written to nonvolatile storage before the failure of the system. This means that a transaction is not permitted to complete its *commit* processing (see [63, 64]) until the redo portions of all log records of that transaction have been written to stable storage.

We deal with three types of failures: transaction or process, system, and media or device. When a transaction or process failure occurs, typically the transaction would be in such a state that its updates would have to be undone. It is possible that the transaction had corrupted some pages in the buffer pool if it was in the middle of performing some updates when the process disappeared. When a system failure occurs, typically the virtual storage contents would be lost and the transaction system would have to be restarted and recovery performed using the nonvolatile storage versions of the database and the log. When a media or device failure occurs, typically the contents of that media would be lost and the lost data would have to be recovered using an image copy (archive dump) version of the lost data and the log.

Forward processing refers to the updates performed when the system is in normal (i.e., not restart recovery) processing and the transaction is updating

the database because of the data manipulation (e.g., SQL) calls issued by the user or the application program. That is, the transaction is not rolling back and using the log to generate the (undo) update calls. *Partial rollback* refers to the ability to set up *savepoints* during the execution of a transaction and later in the transaction request the rolling back of the changes performed by the transaction since the establishment of a previous savepoint [1, 31]. This is to be contrasted with *total rollback* in which all updates of the transaction are undone and the transaction is terminated. Whether or not the savepoint concept is exposed at the application level is immaterial to us since this paper deals only with database recovery. A *nested rollback* is said to have taken place if a partial rollback were to be later followed by a total rollback or another partial rollback whose point of termination is an earlier point in the transaction than the point of termination of the first rollback. *Normal undo* refers to total or partial transaction rollback when the system is in normal operation. A normal undo may be caused by a transaction request to rollback or it may be system initiated because of deadlocks or errors (e.g., integrity constraint violations). *Restart undo* refers to transaction rollback during restart recovery after a system failure. To make partial or total rollback efficient and also to make debugging easier, all the log records written by a transaction are linked via the *PrevLSN* field of the log records in reverse chronological order. That is, the most recently written log record of the transaction would point to the previous most recent log record written by that transaction, if there is such a log record.[2] In many WAL-based systems, the updates performed during a rollback are logged using what are called *compensation log records* (*CLRs*) [15]. Whether a CLR's update is undone, should that CLR be encountered during a rollback, depends on the particular system. As we will see later, in ARIES, a CLR's update is never undone and hence CLRs are viewed as redo-only log records.

Page-oriented redo is said to occur if the log record whose update is being redone describes which page of the database was originally modified during normal processing and if the same page is modified during the redo processing. No internal descriptors of tables or indexes need to be accessed to redo the update. That is, no other page of the database needs to be examined. This is to be contrasted with *logical redo* which is required in System R, SQL/DS and AS/400 for indexes [21, 62]. In those systems, since index changes are not logged separately but are redone using the log records for the data pages, performing a redo requires accessing several descriptors and pages of the database. The index tree would have to be retraversed to determine the page(s) to be modified and, sometimes, the index page(s) modified because of this redo operation may be different from the index page(s) originally modified during normal processing. Being able to perform page-oriented redo allows the system to provide *recovery independence amongst objects*. That is, the recovery of one page's contents does not require accesses to any other

[2] The AS/400, Encompass and NonStop SQL do not explicitly link all the log records written by a transaction. This makes undo inefficient since a *sequential* backward scan of the log must be performed to retrieve all the desired log records of a transaction.

(data or catalog) pages of the database. As we will describe later, this makes media recovery very simple.

In a similar fashion, we can define *page-oriented undo* and *logical undo*. Being able to perform logical undos allows the system to provide higher levels of concurrency than what would be possible if the system were to be restricted only to page-oriented undos. This is because the former, with appropriate concurrency control protocols, would permit uncommitted updates of one transaction to be moved to a different page by another transaction. If one were restricted to only page-oriented undos, then the latter transaction would have had to wait for the former to commit. Page-oriented redo and page-oriented undo permit faster recovery since pages of the database other than the pages mentioned in the log records are not accessed. In the interest of efficiency, ARIES supports page-oriented redo and its supports, in the interest of high concurrency, logical undos. In [62], we introduce the ARIES/IM method for concurrency control and recovery in B^+-tree indexes and show the advantages of being able to perform logical undos by comparing ARIES/IM with other index methods.

1.2 Latches and Locks

Normally latches and locks are used to control access to shared information. Locking has been discussed to a great extent in the literature. Latches, on the other hand, have not been discussed that much. *Latches* are like semaphores. Usually, latches are used to guarantee physical consistency of data, while *locks* are used to assure logical consistency of data. We need to worry about physical consistency since we need to support a multiprocessor environment. Latches are usually held for a much shorter period than are locks. Also, the deadlock detector is not informed about latch waits. Latches are requested in such a manner so as to avoid deadlocks involving latches alone, or involving latches and locks.

Acquiring and releasing a latch is much cheaper than acquiring and releasing a lock. In the no-conflict case, the overhead amounts to 10s of instructions for the former versus 100s of instructions for the latter. Latches are cheaper because the *latch control information* is always in virtual memory in a fixed place, and direct addressability to the latch information is possible given the latch name. As the protocols presented later in this paper and those in [57, 62] show, each transaction holds at most two or three latches simultaneously. As a result, the *latch request blocks* can be permanently allocated to each transaction and initialized with transaction ID, etc. right at the start of that transaction. On the other hand, typically, storage for individual locks has to be acquired, formatted and released dynamically, causing more instructions to be executed to acquire and release locks. This is advisable because, in most systems, the number of lockable objects is many orders of magnitude greater than the number of latchable objects. Typically, all information relating to locks currently held or requested by all the transactions is stored in a single, central hash table. Addressability to a particular lock's information is gained by first hashing the lock name to get the address of the hash anchor and then, possibly, following a chain of pointers. Usually, in the process of trying to locate the *lock control block*,

because multiple transactions may be simultaneously reading and modifying the contents of the lock table, one or more latches will be acquired and released—one latch on the hash anchor and, possibly, one on the specific lock's chain of holders and waiters.

Locks may be obtained in different *modes* such as S (Shared), X (eXclusive), IX (Intention eXclusive), IS (Intention Shared) and SIX (Shared Intention eXclusive), and at different *granularities* such as record (tuple), table (relation), and file (tablespace) [32]. The S and X locks are the most common ones. S provides the read privilege and X provides the read and write privileges. Locks on a given object can be held simultaneously by different transactions only if those locks' modes are *compatible*. The compatibility relationships amongst the above modes of locking are shown in Figure 2. A check mark ('√') indicates that the corresponding modes are compatible. With *hierarchical locking*, the intention locks (IX, IS, and SIX) are generally obtained on the higher levels of the hierarchy (e.g., table), and the S and X locks are obtained on the lower levels (e.g., record). The nonintention mode locks (S and X), when obtained on an object at a certain level of the hierarchy, *implicitly* grant locks of the corresponding mode on the lower level objects of that higher level object. The intention mode locks, on the other hand, only give the privilege of requesting the corresponding intention or nonintention mode locks on the lower level objects. For example, SIX on a table implicitly grants S on all the records of that table, and it allows X to be requested *explicitly* on the records. Additional, semantically rich lock modes have been defined in the literature [2, 38, 45, 55] and ARIES can accommodate them.

Lock requests may be made with the conditional or the unconditional option. A *conditional* request means that the requestor is not willing to wait if, when the request is processed, the lock is not grantable immediately. An *unconditional* request means that the requestor is willing to wait until the lock becomes grantable. Locks may be held for different durations. An unconditional request for an *instant duration* lock means that the lock is not to be actually granted, but the lock manager has to delay returning the lock call with the success status until the lock becomes grantable. *Manual duration* locks are released some time after they are acquired and, typically, long before transaction termination. *Commit duration* locks are released only when the transaction terminates, i.e., after commit or rollback is completed. The above discussions concerning conditional requests, different modes, and durations, except for commit duration, apply to latches also.

1.3 Fine-Granularity Locking

Fine-granularity (e.g., record) locking has been supported by nonrelational database systems (e.g., IMS [53, 76, 80]) for a long time. Surprisingly, only a few of the commercially available relational systems provide fine-granularity locking, even though IBM's System R [32], S/38 [99] and SQL/DS [5], and Tandem's Encompass [37] supported record and/or key locking from the beginning.[3] Although many interesting problems relating to providing

[3] Encompass and S/38 had only X locks for records and no locks were acquired *automatically* by these systems for reads.

Fig. 2. Lock mode compatability matrix

	S	X	IS	IX	SIX
S	√		√		
X					
IS	√		√	√	√
IX			√	√	
SIX			√		

fine-granularity locking in the context of WAL remain to be solved, the research community has not been paying enough attention to this area [3, 75, 88]. Some of the System R solutions worked only because of the use of the shadow page recovery technique in combination with locking (see Section 10). Supporting fine-granularity locking and variable length records in a flexible fashion requires addressing some interesting storage management issues which have never really been discussed in the database literature. Unfortunately, some of the interesting techniques that were developed for System R and which are now part of SQL/DS did not get documented in the literature. At the expense of making this paper long, we will be discussing here some of those problems and their solutions.

As supporting high concurrency gains importance (see [79] for the description of an application requiring very high concurrency) and as object-oriented systems gain in popularity, it becomes necessary to invent concurrency control and recovery methods that take advantage of the semantics of the operations on the data [2, 26, 38, 88, 89], and that support fine-granularity locking efficiently. Object-oriented systems may tend to encourage users to define a large number of small objects and users may expect object instances to be the appropriate granularity of locking. In the object-oriented logical view of the database, the concept of a page, with its physical orientation as the container of objects, becomes unnatural to think about as the unit of locking during object accesses and modifications. Also, object-oriented system users may tend to have many terminal interactions during the course of a transaction, thereby increasing the lock hold times. If the unit of locking were to be a page, lock wait times and deadlock possibilities will be aggravated. Other discussions concerning transaction management in an object-oriented environment can be found in [22, 29].

As more and more customers adopt relational systems for production applications, it becomes ever more important to handle *hot-spots* [28, 34, 68, 77, 79, 83] and storage management without requiring too much tuning by the system users or administrators. Since relational systems have been welcomed to a great extent because of their ease of use, it is important that we pay greater attention to this area than what has been done in the context of the nonrelational systems. Apart from the need for high concurrency for user data, the ease with which online data definition operations can be performed in relational systems by even ordinary users requires the support for high concurrency of access to, at least, the catalog data. Since a leaf page in an index typically describes data in hundreds of data pages, page-level locking of index data is just not acceptable. A flexible recovery method that

allows the support of high levels of concurrency during index accesses is needed.

The above facts argue for supporting semantically rich modes of locking such as increment/decrement which allow multiple transactions to concurrently modify even the same piece of data. In funds-transfer applications, increment and decrement operations are frequently performed on the branch and teller balances by numerous transactions. If those transactions are forced to use only X locks, then they will be serialized, even though their operations commute.

1.4 Buffer Management

The buffer manager (BM) is the component of the transaction system that manages the buffer pool and does I/Os to read/write pages from/to the nonvolatile storage version of the database. The *fix* primitive of the BM may be used to request the buffer address of a logical page in the database. If the requested page is not in the buffer pool, BM allocates a buffer slot and reads the page in. There may be instances (e.g., during a B^+-tree page split, when the new page is allocated) where the current contents of a page on nonvolatile storage are not of interest. In such a case, the *fix_new* primitive may be used to make the BM allocate a *free* slot and return the address of that slot, if BM does not find the page in the buffer pool. The fix_new invoker will then format the page as desired. Once a page is fixed in the buffer pool, the corresponding buffer slot is not available for page replacement until the *unfix* primitive is issued by the data manipulative component. Actually, for each page, BM keeps a fix count which is incremented by one during every fix operation and which is decremented by one during every unfix operation. A page in the buffer pool is said to be *dirty* if the buffer version of the page has some updates which are not yet reflected in the nonvolatile storage version of the same page. The fix primitive is also used to communicate the intention to modify the page. Dirty pages can be written back to nonvolatile storage when no fix with the modification intention is held, thus allowing read accesses to the page while it is being written out. [96] discusses the role of BM in writing in the background, on a continuous basis, dirty pages to nonvolatile storage to reduce the amount of redo work that would be needed if a system failure were to occur and also to keep a certain percentage of the buffer pool pages in the nondirty state so that they may be replaced with other pages without synchronous write I/Os having to be performed at the time of replacement. While performing those writes, BM ensures that the WAL protocol is obeyed. As a consequence, BM may have to force the log up to the LSN of the dirty page before writing the page to nonvolatile storage. Given the large buffer pools that are common today, we would expect a force of this nature to be very rare and most log forces to occur because of transactions committing or entering the prepare state.

BM also implements the support for latching pages. To provide direct addressability to page latches and to reduce the storage associated with those latches, the latch on a logical page is actually the latch on the corresponding buffer slot. This means that a logical page can be latched only after it is fixed

in the buffer pool and the latch has to be released before the page is unfixed. These are highly acceptable conditions. The latch control information is stored in the buffer control block (BCB) for the corresponding buffer slot. The BCB also contains the identity of the logical page, what the fix count is, the dirty status of the page, etc.

Buffer management policies differ among the many systems in existence (see Section 11, "Other WAL-Based Methods"). If a page modified by a transaction is allowed to be written to the permanent database on nonvolatile storage before that transaction commits, then the *steal* policy is said to be followed by the buffer manager (see [36] for such terminologies). Otherwise, a *no-steal* policy is said to be in effect. Steal implies that during normal or restart rollback, some undo work might have to be performed on the nonvolatile storage version of the database. If a transaction is not allowed to commit until all pages modified by it are written to the permanent version of the database, then a *force* policy is said to be in effect. Otherwise, a *no-force* policy is said to be in effect. With a force policy, during restart recovery, no redo work will be necessary for committed transactions. *Deferred updating* is said to occur if, even in the virtual storage database buffers, the updates are not performed in-place when the transaction issues the corresponding database calls. The updates are kept in a pending list elsewhere and are performed in-place, using the pending list information, only after it is determined that the transaction is definitely committing. If the transaction needs to be rolled back, then the pending list is discarded or ignored. The deferred updating policy has implications on whether a transaction can "see" its own updates or not, and on whether partial rollbacks are possible or not.

For more discussions concerning buffer management, see [8, 15, 24, 96].

1.5 Organization

The rest of the paper is organized as follows. After stating our goals in Section 2 and giving an overview of the new recovery method ARIES in Section 3, we present, in Section 4, the important data structures used by ARIES during normal and restart recovery processing. Next, in Section 5, the protocols followed during normal processing are presented followed, in Section 6, by the description of the processing performed during restart recovery. The latter section also presents ways to exploit parallelism during recovery and methods for performing recovery selectively or postponing the recovery of some of the data. Then, in Section 7, algorithms are described for taking checkpoints during the different log passes of restart recovery to reduce the impact of failures during recovery. This is followed, in Section 8, by the description of how fuzzy image copying and media recovery are supported. Section 9 introduces the significant notion of *nested top actions* and presents a method for implementing them efficiently. Section 10 describes and critiques some of the existing recovery paradigms which originated in the context of the shadow page technique and System R. We discuss the problems caused by using those paradigms in the WAL context. Section 11 describes in detail the characteristics of many of the WAL-based recovery methods in use in different systems such as IMS, DB2, Encompass and NonStop SQL.

Section 12 outlines the many different properties of ARIES. We conclude by summarizing, in Section 13, the features of ARIES which provide flexibility and efficiency, and by describing the extensions and the current status of the implementations of ARIES.

Besides presenting a new recovery method, by way of motivation for our work, we also describe some previously unpublished aspects of recovery in System R. For comparison purposes, we also do a survey of the recovery methods used by other WAL-based systems and collect information appearing in several publications, many of which are not widely available. One of our aims in this paper is to show the intricate and unobvious interactions resulting from the different choices made for the recovery technique, the granularity of locking and the storage management scheme. One cannot make arbitrarily independent choices for these and still expect the combination to function together correctly and efficiently. This point needs to be emphasized as it is not always dealt with adequately in most papers and books on concurrency control and recovery. In this paper, we have tried to cover, as much as possible, all the interesting recovery-related problems that one encounters in building and operating an *industrial-strength* transaction processing system.

2. GOALS

This section lists the goals of our work and outlines the difficulties involved in designing a recovery method that supports the features that we aimed for. The goals relate to the metrics for comparison of recovery methods that we discussed earlier, in Section 1.1.

Simplicity. Concurrency and recovery are complex subjects to think about and program for, compared with other aspects of data management. The algorithms are bound to be error-prone, if they are complex. Hence, we strived for a simple, yet powerful and flexible, algorithm. Although this paper is long because of its comprehensive discussion of numerous problems that are mostly ignored in the literature, the main algorithm itself is quite simple. Hopefully, the overview presented in Section 3 gives the reader that feeling.

Operation logging. The recovery method had to permit operation logging (and value logging) so that semantically rich lock modes could be supported. This would let one transaction modify the same data that was modified earlier by another transaction which has not yet committed, when the two transactions' actions are semantically compatible (e.g., increment/decrement operations; see [2, 26, 45, 88]). As should be clear, recovery methods which always perform *value* or *state logging* (i.e., logging before-images and after-images of modified data), cannot support operation logging. This includes systems that do very physical—byte-oriented—logging of all changes to a page [6, 76, 81]. The difficulty in supporting operation logging is that we need to track precisely, using a concept like the LSN, the exact state of a page with respect to logged actions relating to that page. An undo or a redo of an update should not be performed without being sure that the original update

is present or is not present, respectively. This also means that, if one or more transactions that had previously modified a page start rolling back, then we need to know precisely how the page has been affected during the rollbacks and how much of each of the rollbacks had been accomplished so far. This requires that updates performed during rollbacks also be logged via the so-called *compensation log records* (*CLRs*). The LSN concept lets us avoid attempting to redo an operation when the operation's effect is already present in the page. It also lets us avoid attempting to undo an operation when the operation's effect is not present in the page. Operation logging lets us perform, if found desirable, *logical logging*, which means that not everything that was changed on a page needs to be logged explicitly, thereby saving log space. For example, changes of control information, like the amount of free space on the page, need not be logged. The redo and the undo operations can be performed logically. For a good discussion of operation and value logging, see [88].

Flexible storage management. Efficient support for the storage and manipulation of varying length data is important. In contrast to systems like IMS, the intent here is to be able to avoid the need for off-line reorganization of the data to garbage collect any space that might have been freed up because of deletions and updates that caused data shrinkage. It is desirable that the recovery method and the concurrency control method be such that the logging and locking is *logical* in nature so that movements of the data within a page for garbage collection reasons do not cause the moved data to be locked or the movements to be logged. For an index, this also means that one transaction must be able to split a leaf page even if that page currently has some uncommitted data inserted by another transaction. This may lead to problems in performing page-oriented undos using the log; *logical undos* may be necessary. Further, we would like to be able to let a transaction that has freed up some space be able to use, if necessary, that space during its later insert activity [50]. System R, for example, does not permit this in data pages.

Partial rollbacks. It was essential that the new recovery method support the concept of savepoints and rollbacks to savepoints (i.e., partial rollbacks). This is crucial for handling, in a user-friendly fashion (i.e., without requiring a total rollback of the transaction), integrity constraint violations (see [1, 31]), and problems arising from using obsolete cached information (see [49]).

Flexible buffer management. The recovery method should make the least number of restrictive assumptions about the buffer management policies (*steal*, *force*, etc.) in effect. At the same time, the method must be able to take advantage of the characteristics of any specific policy that is in effect (e.g., with a force policy there is no need to perform any redos for committed transactions.) This flexibility could result in increased concurrency, decreased I/Os and efficient usage of buffer storage. Depending on the policies, the work that needs to be performed during restart recovery after a system

failure or during media recovery may be more or less complex. Even with large main memories, it must be noted that a steal policy is still very desirable. This is because, with a no-steal policy, a page may never get written to nonvolatile storage if the page always contains *uncommitted* updates due to fine-granularity locking and overlapping transactions' updates to that page. The situation would be further aggravated if there are long-running transactions. Under those conditions, either the system would have to frequently reduce concurrency by quiescing all activities on the page (i.e., by locking all the objects on the page) and then writing the page to nonvolatile storage, or by doing nothing special and then paying a huge restart redo recovery cost if the system were to fail. Also, a no-steal policy incurs additional bookkeeping overhead to track whether a page contains any uncommitted updates. We believe that, given our goal of supporting semantically rich lock modes, partial rollbacks and varying length objects efficiently, in the general case, we need to perform undo logging and in-place updating. Hence, methods like the transaction workspace model of AIM [46] are not general enough for our purposes. Other problems relating to no-steal are discussed in Section 11 with reference to IMS Fast Path.

Recovery independence. It should be possible to image copy (archive dump), and perform media recovery or restart recovery at different granularities, rather than only at the entire database level. The recovery of one object should not force the concurrent or lock-step recovery of another object. Contrast this with what happens in the shadow page technique as implemented in System R, where index and space management information are recovered *lock-step* with user and catalog table (relation) data by starting from an internally consistent state of the *whole* database and redoing changes to all the related objects of the database simultaneously, as in normal processing. Recovery independence means that, during the restart recovery of some object, catalog information in the database cannot be accessed for descriptors of that object and its related objects, since that information itself may be undergoing recovery in parallel with the object being recovered and the two may be out of synchronization [14]. During restart recovery, it should be possible to do selective recovery and defer recovery of some objects to a later point in time to speed up restart and also to accommodate some offline devices. *Page-oriented recovery* means that even if one page in the database is corrupted because of a process failure or a media problem, it should be possible to recover that page alone. To be able to do this efficiently, we need to log every page's change individually, even if the object being updated spans multiple pages and the update affects more than one page. This, in conjunction with the writing of CLRs for updates performed during rollbacks, will make media recovery very simple (see Section 8). This will also permit image copying of different objects to be performed independently and at different frequencies.

Logical undo. This relates to the ability, during undo, to affect a page that is different from the one modified during forward processing, as is

needed in the earlier-mentioned context of the split by one transaction of an index page containing uncommitted data of another transaction. Being able to perform logical undos allows higher levels of concurrency to be supported, especially in search structures [57, 59, 62]. If logging is not performed during rollback processing, logical undos would be very difficult to support, if we also desired recovery independence and page-oriented recovery. System R and SQL/DS support logical undos, but at the expense of recovery independence.

Parallelism and fast recovery. With multiprocessors becoming very common and greater data availability becoming increasingly important, the recovery method has to be able to exploit parallelism during the different stages of restart recovery and during media recovery. It is also important that the recovery method be such that recovery can be very fast, if in fact a *hot-standby* approach is going to be used (a la IBM's IMS/VS XRF [43] and Tandem's NonStop [4, 37]). This means that redo processing and, whenever possible, undo processing should be page-oriented (cf. always logical redos and undos in System R and SQL/DS for indexes and space management). It should also be possible to let the backup system start processing new transactions, even before the undo processing for the interrupted transactions completes. This is necessary because undo processing may take a long time if there were long update transactions.

Minimal overhead. Our goal is to have good performance both during normal and restart recovery processing. The overhead (log data volume, storage consumption, etc.) imposed by the recovery method in virtual and nonvolatile storages for accomplishing the above goals should be minimal. Contrast this with the space overhead caused by the shadow page technique. This goal also implied that we should minimize the number of pages that are modified (dirtied) during restart. The idea is to reduce the number of pages that have to be written back to nonvolatile storage and also to reduce CPU overhead. This rules out methods which, during restart recovery, first undo some committed changes that had already reached the nonvolatile storage before the failure and then redo them (see, e.g., [16, 21, 72, 78, 88]). It also rules out methods in which updates that are not present in a page on nonvolatile storage are undone unnecessarily (see, e.g., [41, 71, 88]). The method should not cause deadlocks involving transactions that are already rolling back. Further, the writing of CLRs should not result in an unbounded number of log records having to be written for a transaction because of the undoing of CLRs, if there were nested rollbacks or repeated system failures during rollbacks. It should also be possible to take checkpoints and image copies without quiescing significant activities in the system. The impact of these operations on other activities should be minimal. To contrast, checkpointing and image copying in System R cause major perturbations in the rest of the system [31].

As the reader will have realized by now, some of these goals are contradictory. Based on our knowledge of different developers' existing systems' features, experiences with IBM's existing transaction systems and contacts

with customers, we made the necessary tradeoffs. We were keen on learning from the past successes and mistakes involving many prototypes and products.

3. OVERVIEW OF ARIES

The aim of this section is to provide a brief overview of the new recovery method ARIES, which satisfies quite reasonably the goals that we set forth in Section 2. Issues like deferred and selective restart, parallelism during restart recovery, and so on will be discussed in the later sections of the paper.

ARIES guarantees the atomicity and durability properties of transactions in the fact of process, transaction, system and media failures. For this purpose, ARIES keeps track of the changes made to the database by using a log and it does write-ahead logging (WAL). Besides logging, on a per-affected-page basis, update activities performed during forward processing of transactions, ARIES also logs, typically using compensation log records (CLRs), updates performed during partial or total rollbacks of transactions during both normal and restart processing. Figure 3 gives an example of a partial rollback in which a transaction, after performing three updates, rolls back two of them and then starts going forward again. Because of the undo of the two updates, two CLRs are written. In ARIES, CLRs have the property that they are redo-only log records. By appropriate chaining of the CLRs to log records written during forward processing, a bounded amount of logging is ensured during rollbacks, even in the face of repeated failures during restart or of nested rollbacks. This is to be contrasted with what happens in IMS, which may undo the same non-CLR multiple times, and in AS/400, DB2 and NonStop SQL, which, besides undoing the same non-CLR multiple times, may also undo CLRs one or more times (see Figure 4). These have caused severe problems in real-life customer situations.

In ARIES, as Figure 5 shows, when the undo of a log record causes a CLR to be written, the CLR, besides containing a description of the compensating action for redo purposes, is made to contain the *UndoNxtLSN* pointer which points to the *predecessor* of the just undone log record. The predecessor information is readily available since every log record, including a CLR, contains the *PrevLSN* pointer which points to the most recent preceding log record written by the same transaction. The UndoNxtLSN pointer allows us to determine precisely how much of the transaction has not been undone so far. In Figure 5, log record 3′, which is the CLR for log record 3, points to log record 2, which is the predecessor of log record 3. Thus, during rollback, the UndoNxtLSN field of the most recently written CLR keeps track of the progress of rollback. It tells the system from where to continue the rollback of the transaction, if a system failure were to interrupt the completion of the rollback or if a nested rollback were to be performed. It lets the system bypass those log records that had already been undone. Since CLRs are available to describe what actions are actually performed during the undo of an original action, the undo action need not be, in terms of which page(s) is affected, the exact inverse of the original action. That is, logical undo which allows very high concurrency to be supported is made possible. For example,

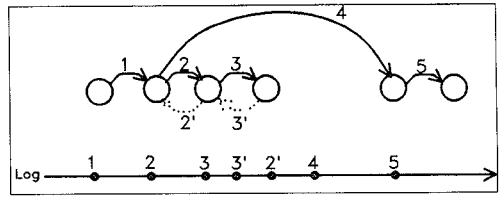
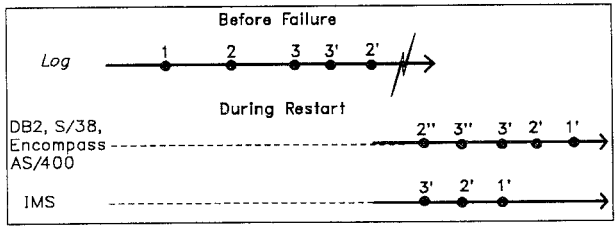


Fig. 3. Partial rollback example.

After performing 3 actions, the transaction performs a partial rollback by undoing actions 3 and 2, writing the compensation log records 3′ and 2′, and then starts going forward again and performs actions 4 and 5



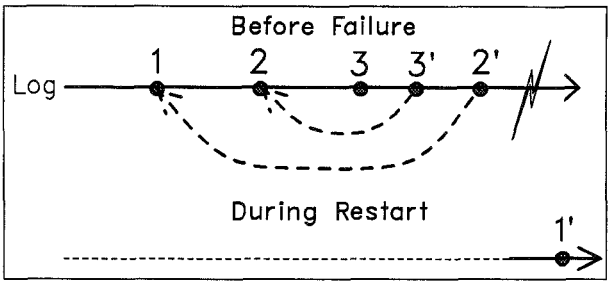
I′ is the CLR for I and I″ is the CLR for I′

Fig. 4 Problem of compensating compensations or duplicate compensations, or both.

a key inserted on page 10 of a B^+-tree by one transaction may be moved to page 20 by another transaction before the key insertion is committed. Later, if the first transaction were to roll back, then the key will be located on page 20 by retraversing the tree and deleted from there. A CLR will be written to describe the key deletion on page 20. This permits page-oriented redo which is very efficient. [59, 62] describe ARIES/LHS and ARIES/IM which exploit this logical undo feature.

ARIES uses a single LSN on each page to track the page's state. Whenever a page is updated and a log record is written, the LSN of the log record is placed in the *page_LSN* field of the updated page. This tagging of the page with the LSN allows ARIES to precisely track, for restart- and media-recovery purposes, the state of the page with respect to logged updates for that page. It allows ARIES to support novel lock modes, using which, before an update performed on a record's field by one transaction is committed, another transaction may be permitted to modify the same data for specified operations.

Periodically during normal processing, ARIES takes checkpoints. The checkpoint log records identify the transactions that are active, their states, and the LSNs of their most recently written log records, and also the modified data (dirty data) that is in the buffer pool. The latter information is needed to determine from where the redo pass of restart recovery should begin its processing.



I′ is the Compensation Log Record for I
I′ points to the predecessor, if any, of I

Fig. 5. ARIES' technique for avoiding compensating compensations and duplicate compensations.

During restart recovery (see Figure 6), ARIES first scans the log, starting from the first record of the last checkpoint, up to the end of the log. During this *analysis pass*, information about dirty pages and transactions that were in progress at the time of the checkpoint is brought up to date as of the end of the log. The analysis pass uses the dirty pages information to determine the starting point (*RedoLSN*) for the log scan of the immediately following redo pass. The analysis pass also determines the list of transactions that are to be rolled back in the undo pass. For each in-progress transaction, the LSN of the most recently written log record will also be determined. Then, during the *redo pass*, ARIES *repeats history*, with respect to those updates logged on stable storage, but whose effects on the database pages did not get reflected on nonvolatile storage before the failure of the system. This is done for the updates of all transactions, including the updates of those transactions that had neither committed nor reached the in-doubt state of two-phase commit by the time of the system failure (i.e., even the missing updates of the so-called *loser* transactions are redone). This essentially reestablishes the state of the database as of the time of the system failure. A log record's update is redone if the affected page's page_LSN is less than the log record's LSN. No logging is performed when updates are redone. The redo pass obtains the locks needed to protect the uncommitted updates of those distributed transactions that will remain in the in-doubt (prepared) state [63, 64] at the end of restart recovery.

The next log pass is the *undo pass* during which all loser transactions' updates are rolled back, in reverse chronological order, in a single sweep of the log. This is done by continually taking the maximum of the LSNs of the next log record to be processed for each of the yet-to-be-completely-undone loser transactions, until no transaction remains to be undone. Unlike during the redo pass, performing undos is not a conditional operation during the undo pass (and during normal undo). That is, ARIES does not compare the page_LSN of the affected page to the LSN of the log record to decide

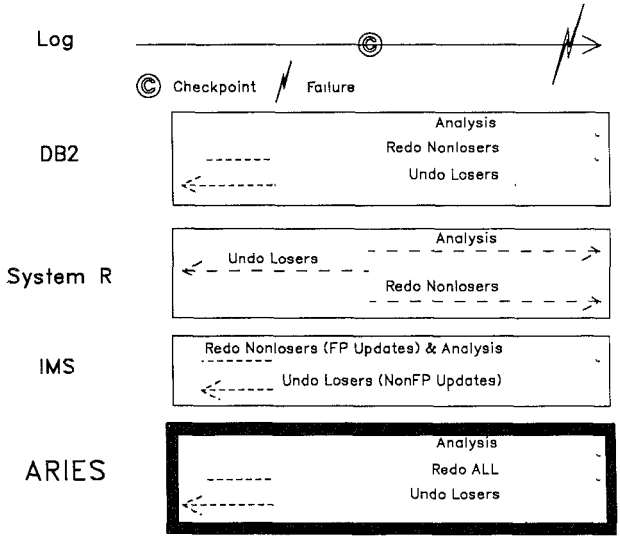


Fig. 6. Restart processing in different methods.

whether or not to undo the update. When a non-CLR is encountered for a transaction during the undo pass, if it is an undo-redo or undo-only log record, then its update is undone. In any case, the next record to process for that transaction is determined by looking at the PrevLSN of that non-CLR. Since CLRs are never undone (i.e., CLRs are not compensated—see Figure 5), when a CLR is encountered during undo, it is used just to determine the next log record to process by looking at the UndoNxtLSN field of the CLR.

For those transactions which were already rolling back at the time of the system failure, ARIES will rollback only those actions that had not already been undone. This is possible since history is repeated for such transactions and since the last CLR written for each transaction points (directly or indirectly) to the next non-CLR record that is to be undone. The net result is that, if only page-oriented undos are involved or logical undos generate only CLRs, then, for rolled back transactions, the number of CLRs written will be exactly equal to the number of undoable) log records written during forward processing of those transactions. This will be the case even if there are repeated failures during restart or if there are nested rollbacks.

4. DATA STRUCTURES

This section describes the major data structures that are used by ARIES.

4.1 Log Records

Below, we describe the important fields that may be present in different types of log records.

LSN. Address of the first byte of the log record in the ever-growing log address space. This is a monotonically increasing value. This is shown here as a field only to make it easier to describe ARIES. The LSN need not actually be stored in the record.

Type. Indicates whether this is a compensation record ('compensation'), a regular update record ('update'), a commit protocol-related record (e.g., 'prepare'), or a nontransaction-related record (e.g., 'OSfile_return').

TransID. Identifier of the transaction, if any, that wrote the log record.

PrevLSN. LSN of the preceding log record written by the same transaction. This field has a value of zero in nontransaction-related records and in the first log record of a transaction, thus avoiding the need for an explicit begin transaction log record.

PageID. Present only in records of type 'update' or 'compensation'. The identifier of the page to which the updates of this record were applied. This PageID will normally consist of two parts: an objectID (e.g., tablespaceID), and a page number within that object. ARIES can deal with a log record that contains updates for multiple pages. For ease of exposition, we assume that only one page is involved.

UndoNxtLSN. Present only in CLRs. It is the LSN of the next log record of this transaction that is to be processed during rollback. That is, UndoNxtLSN is the value of PrevLSN of the log record that the current log record is compensating. If there are no more log records to be undone, then this field contains a zero.

Data. This is the redo and/or undo data that describes the update that was performed. CLRs contain only redo information since they are never undone. Updates can be logged in a logical fashion. Changes to some fields (e.g., amount of free space) of that page need not be logged since they can be easily derived. The undo information and the redo information for the entire object need not be logged. It suffices if the changed fields alone are logged. For increment or decrement types of operations, before and after-images of the field are not needed. Information about the type of operation and the decrement or increment amount is enough. The information here would also be used to determine the appropriate action routine to be used to perform the redo and/or undo of this log record.

4.2 Page Structure

One of the fields in every page of the database is the *page_LSN* field. It contains the LSN of the log record that describes the latest update to the page. This record may be a regular update record or a CLR. ARIES expects the buffer manager to enforce the WAL protocol. Except for this, ARIES does not place any restrictions on the buffer page replacement policy. The steal buffer management policy may be used. In-place updating is performed on nonvolatile storage. Updates are applied immediately and directly to the

buffer version of the page containing the object. That is, no deferred updating as in INGRES [86] is performed. If it is found desirable, deferred updating and, consequently, deferred logging can be implemented. ARIES is flexible enough not to preclude those policies from being implemented.

4.3 Transaction Table

A table called the *transaction table* is used during restart recovery to track the state of active transactions. The table is initialized during the analysis pass from the most recent checkpoint's record(s) and is modified during the analysis of the log records written after the beginning of that checkpoint. During the undo pass, the entries of the table are also modified. If a checkpoint is taken during restart recovery, then the contents of the table will be included in the checkpoint record(s). The same table is also used during normal processing by the transaction manager. A description of the important fields of the transaction table follows:

TransID. Transaction ID.

State. Commit state of the transaction: prepared ('P'—also called in-doubt) or unprepared ('U').

LastLSN. The LSN of the latest log record written by the transaction.

UndoNxtLSN. The LSN of the next record to be processed during rollback. If the most recent log record written or seen for this transaction is an undoable non-CLR log record, then this field's value will be set to LastLSN. If that most recent log record is a CLR, then this field's value is set to the UndoNxtLSN value from that CLR.

4.4 Dirty_Pages Table

A table called the *dirty_pages table* is used to represent information about dirty buffer pages during normal processing. This table is also used during restart recovery. The actual implementation of this table may be done using hashing or via the deferred-writes queue mechanism of [96]. Each entry in the table consists of two fields: PageID and *RecLSN* (recovery LSN). During normal processing, when a nondirty page is being fixed in the buffers with the intention to modify, the buffer manager records in the *buffer pool* (BP) dirty_pages table, as RecLSN, the current end-of-log LSN, which will be the LSN of the next log record to be written. The value of RecLSN indicates from what point in the log there may be updates which are, possibly, not yet in the nonvolatile storage version of the page. Whenever pages are written back to nonvolatile storage, the corresponding entries in the BP dirty_pages table are removed. The contents of this table are included in the checkpoint record(s) that is written during normal processing. The *restart* dirty_pages table is initialized from the latest checkpoint's record(s) and is modified during the analysis of the other records during the analysis pass. The

minimum RecLSN value in the table gives the starting point for the redo pass during restart recovery.

5. NORMAL PROCESSING

This section discusses the actions that are performed as part of normal transaction processing. Section 6 discusses the actions that are performed as part of recovering from a system failure.

5.1 Updates

During normal processing, transactions may be in forward processing, partial rollback or total rollback. The rollbacks may be system- or application-initiated. The causes of rollbacks may be deadlocks, error conditions, integrity constraint violations, unexpected database state, etc.

If the granularity of locking is a record, then, when an update is to be performed on a record in a page, after the record is locked, that page is fixed in the buffer and latched in the X mode, the update is performed, a log record is appended to the log, the LSN of the log record is placed in the page_LSN field of the page and in the transaction table, and the page is unlatched and unfixed. The page latch is held during the call to the logger. This is done to ensure that the order of logging of updates of a page is the same as the order in which those updates are performed on the page. This is very important if some of the redo information is going to be logged physically (e.g., the amount of free space in the page) and repetition of history has to be guaranteed for the physical redo to work correctly. The page latch must be held during read and update operations to ensure physical consistency of the page contents. This is necessary because inserters and updaters of records might move records around within a page to do garbage collection. When such garbage collection is going on, no other transaction should be allowed to look at the page since they might get confused. Readers of pages latch in the S mode and modifiers latch in the X mode.

The data page latch is not held while any necessary index operations are performed. At most two page latches are held simultaneously (also see [57, 62]). This means that two transactions, T1 and T2, that are modifying different pieces of data may modify a particular data page in one order (T1, T2) and a particular index page in another order (T2, T1).[4] This scenario is impossible in System R and SQL/DS since in those systems, locks, instead of latches are used for providing physical consistency. Typically, all the (physical) page locks are released only at the end of the RSS (data manager) call. A single RSS call deals with modifying the data and all relevant indexes. This may involve waiting for many I/Os and locks. This means that deadlocks involving (physical) page locks alone or (physical) page locks and

[4] The situation gets very complicated if operations like increment/decrement are supported with high concurrency lock modes and indexes are allowed to be defined on fields on which such operations are supported. We are currently studying those situations.

(logical) record/key locks are possible. They have been a major problem in System R and SQL/DS.

Figure 7 depicts a situation at the time of a system failure which followed the commit of two transactions. The dotted lines show how up to date the states of pages P1 and P2 are on nonvolatile storage with respect to logged updates of those pages. During restart recovery, it must be realized that the most recent log record written for P1, which was written by a transaction which later committed, needs to be redone, and that there is nothing to be redone for P2. This situation points to the need for having the LSN to relate the state of a page on nonvolatile storage to a particular position in the log and the need for knowing where restart redo pass should begin by noting some information in the checkpoint record (see Section 5.4). For the example scenario, the restart redo log scan should begin at least from the log record representing the most recent update of P1 by T2, since that update needs to be redone.

It is not assumed that a single log record can always accommodate all the information needed to redo or undo the update operation. There may be instances when more than one record needs to be written for this purpose. For example, one record may be written with the undo information and another one with the redo information. In such cases, (1) the undo-only log record should be written before the redo-only log record is written, and (2) it is the LSN of the *redo-only log record* that should be placed in the page_LSN field. The first condition is enforced to make sure that we do not have a situation in which the redo-only record and not the undo-only record gets written to stable storage before a failure, and that during restart recovery, the redo of that redo-only log record is performed (because of the repeating history feature) only to realize later that there isn't an undo-only record to undo the effect of that operation. Given that the undo-only record is written before the redo-only record, the second condition ensures that we do not have a situation in which even though the page in nonvolatile storage already contains the update of the redo-only record, that same update gets redone unnecessarily during restart recovery because the page contained the LSN of the undo-only record instead of that of the redo-only record. This unnecessary redo could cause integrity problems if operation logging is being performed.

There may be some log records written during forward processing that cannot or should not be undone (prepare, free space inventory update, etc. records). These are identified as *redo-only* log records. See Section 10.3 for a discussion of this kind of situation for free space inventory updates.

Sometimes, the identity of the (data) record to be modified or read may not be known before a (data) page is examined. For example, during an insert, the record ID is not determined until the page is examined to find an empty slot. In such cases, the record lock must be obtained after the page is latched. To avoid waiting for a lock while holding a latch, which could lead to an undetected deadlock, the lock is requested *conditionally*, and if it is not granted, then the latch is released and the lock is requested *unconditionally*. Once the unconditionally requested lock is granted, the page is latched again, and any previously verified conditions are rechecked. This rechecking is

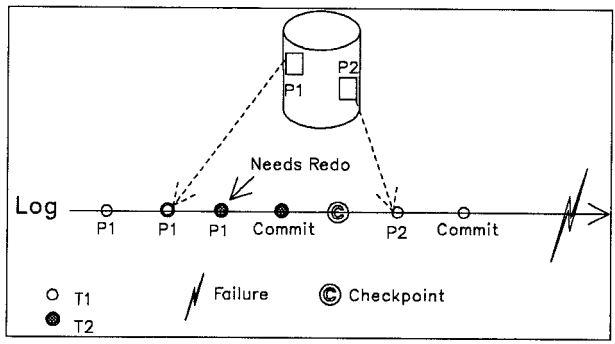


Fig. 7. Database state as a failure.

required because, after the page was unlatched, the conditions could have changed. The page_LSN value at the time of unlatching could be remembered to detect quickly, on relatching, if any changes could have possibly occurred. If the conditions are still found to be satisfied for performing the update, it is performed as described above. Otherwise, corrective actions are taken. If the conditionally requested lock is granted immediately, then the update can proceed as before.

If the granularity of locking is a page or something coarser than a page, then there is no need to latch the page since the lock on the page will be sufficient to isolate the executing transaction. Except for this change, the actions taken are the same as in the record-locking case. But, if the system is to support unlocked or *dirty* reads, then, even with page locking, a transaction that is updating a page should be made to hold the X latch on the page so that readers who are not acquiring locks are assured physical consistency if they hold an S latch while reading the page. Unlocked reads may also be performed by the image copy utility in the interest of causing the least amount of interference to normal transaction processing.

Applicability of ARIES is not restricted to only those systems in which locking is used as the concurrency control mechanism. Even other concurrency control schemes that are similar to locking, like the ones in [2], could be used with ARIES.

5.2 Total or Partial Rollbacks

To provide flexibility in limiting the extent of transaction rollbacks, the notion of a *savepoint* is supported [1, 31]. At any point during the execution of a transaction, a savepoint can be established. Any number of savepoints could be outstanding at a point in time. Typically, in a system like DB2, a savepoint is established before every SQL data manipulation command that might perform updates to the data. This is needed to support SQL statement-level atomicity. After executing for a while, the transaction or the system can request the undoing of all the updates performed after the establishment of a still outstanding savepoint. After such a partial rollback, the transaction can

continue execution and start going forward again (see Figure 3). A particular savepoint is no longer outstanding if a rollback has been performed to that savepoint or to a preceding one. When a savepoint is established, the LSN of the latest log record written by the transaction, called *SaveLSN*, is remembered in virtual storage. If the savepoint is being established at the beginning of the transaction (i.e., when it has not yet written a log record) SaveLSN is set to zero. When the transaction desires to roll back to a savepoint, it supplies the remembered SaveLSN. If the savepoint concept were to be exposed at the user level, then we would expect the system not to expose the SaveLSNs to the user but use some symbolic values or sequence numbers and do the mapping to LSNs internally, as is done in IMS [42] and INGRES [18].

Figure 8 describes the routine *ROLLBACK* which is used for rolling back to a savepoint. The input to the routine is the SaveLSN and the TransID. No locks are acquired during rollback, even though a latch is acquired during undo activity on a page. Since we have always ensured that latches do not get involved in deadlocks, a rolling back transaction cannot get involved in a deadlock, as in System R and R* [31, 64] and in the algorithms of [100]. During the rollback, the log records are undone in reverse chronological order and, for each log record that is undone, a CLR is written. For ease of exposition, assume that all the information about the undo action will fit in a single CLR. It is easy to extend ARIES to the case where multiple CLRs need to be written. It is possible that, when a logical undo is performed, some non-CLRs are sometimes written, as described in [59, 62]. As mentioned before, when a CLR is written, its UndoNxtLSN field is made to contain the PrevLSN value in the log record whose undo caused this CLR to be written. Since CLRs will never be undone, they don't have to contain undo information (e.g., before-images). Redo-only log records are ignored during rollback. When a non-CLR is encountered, after it is processed, the next record to process is determined by looking up its PrevLSN field. When a CLR is encountered during rollback, the UndoNxtLSN field of that record is looked up to determine the next log record to be processed. Thus, the UndoNxtLSN pointer helps us skip over already undone log records. This means that if a nested rollback were to occur, then, because of the UndoNxtLSN in CLRs, during the second rollback none of the log records that were undone during the first rollback would be processed again. Even though Figures 4, 5, and 13 describe partial rollback scenarios in conjunction with restart undos in the various recovery methods, it should be easy to see how nested rollbacks are handled efficiently by ARIES.

Being able to describe, via CLRs, the actions performed during undo gives us the flexibility of not having to force the undo actions to be the exact inverses of the original actions. In particular, the undo action could affect a page which was not involved in the original action. Such logical undo situations are possible in, for example, index management [62] and space management (see Section 10.3).

ARIES' guarantee of a bounded amount of logging during undo allows us to deal safely with small computer systems situations in which a circular online

```
ROLLBACK(SaveLSN,TransID);
UndoNxt := Trans_Table[TransID].UndoNxtLSN;                    /* addr of 1st record to undo   */
WHILE SaveLSN < UndoNxt DO;                          /* loop thru all relevant records        */
  LogRec := Log_Read(UndoNxt);                       /* read record to be processed           */
  SELECT (LogRec.Type)
    WHEN('update') DO;
      IF LogRec is undoable THEN DO;
        Page := fix&latch(LogRec.PageID,'X');
        Undo_Update(Page,LogRec);
        Log_Write('compensation',LogRec.TransID,Trans_Table[TransID].LastLSN,
                LogRec.PageID,LogRec.PrevLSN, ...,LgLSN,Data);                /* write CLR */
        Page.LSN := LgLSN;
        Trans_Table[TransID].LastLSN := LgLSN;
        unfix&unlatch(Page);
      END;
      UndoNxt := LogRec.PrevLSN;
    END; /* WHEN('update') */
    WHEN('compensation')  UndoNxt := LogRec.UndoNxtLSN;          /* a CLR - nothing to undo */
    OTHERWISE  UndoNxt := LogRec.PrevLSN              /* skip record and go to previous one */
  END; /* SELECT */
  Trans_Table[TransID].UndoNxtLSN := UndoNxt;
END;   /* WHILE  */
RETURN;
```

Fig. 8. Pseudocode for rollback.

log might be used and log space is at a premium. Knowing the bound, we can keep in reserve enough log space to be able to roll back all currently running transactions under critical conditions (e.g., log space shortage). The implementation of ARIES in the OS/2 Extended Edition Database Manager takes advantage of this.

When a transaction rolls back, the locks obtained after the establishment of the savepoint which is the target of the rollback may be released after the partial or total rollback is completed. In fact, systems like DB2 do not and cannot release any of the locks after a partial rollback because, after such a lock release, a later rollback may still cause the same updates to be undone again, thereby causing data inconsistencies. System R does release locks after a partial rollback completes. But, because ARIES never undoes CLRs nor ever undoes a particular non-CLR more than once, because of the chaining of the CLRs using the UndoNxtLSN field, during a (partial) rollback, when the transaction's very first update to a particular object is undone and a CLR is written for it, the system can release the lock on that object. This makes it possible to consider resolving deadlocks using partial rollbacks rather than always resorting to total rollbacks.

5.3 Transaction Termination

Assume that some form of two-phase commit protocol (e.g., Presumed Abort or Presumed Commit (see [63, 64])) is used to terminate transactions and that the *prepare* record which is *synchronously* written to the log as part of the protocol includes the list of update-type locks (IX, X, SIX, etc.) held by the transaction. The logging of the locks is done to ensure that if a system failure were to occur after a transaction enters the in-doubt state, then those locks could be reacquired, during restart recovery, to protect the uncommitted updates of the in-doubt transaction.[5] When the prepare record is written, the read locks (e.g., S and IS) could be released, if no new locks would be acquired later as part of getting into the prepare state in some other part of the distributed transaction (at the same site or a different site). To deal with actions (such as the dropping of objects) which may cause files to be erased, for the sake of avoiding the logging of such objects' complete contents, we postpone performing actions like erasing files until we are sure that the transaction is definitely committing [19]. We need to log these *pending actions* in the prepare record.

Once a transaction enters the in-doubt state, it is committed by writing an *end* record and releasing its locks. Once the end record is written, if there are any pending actions, they they must be performed. For each pending action which involves erasing or returning a file to the operating system, we write an *OSfile_return* redo-only log record. For ease of exposition, we assume that this log record is not associated with any particular transaction and that this action does not take place when a checkpoint is in progress.

[5] Another possibility is not to log the locks, but to regenerate the lock names during restart recovery by examining all the log records written by the in-doubt transaction—see Sections 6.1 and 6 4, and item 18 (Section 12) for further ramifications of this approach

A transaction in the *in-doubt* state is rolled back by writing a *rollback* record, rolling back the transaction to its beginning, discarding the pending actions list, releasing its locks, and then writing the end record. Whether or not the rollback and end records are *synchronously* written to stable storage will depend on the type of two-phase commit protocol used. Also, the writing of the prepare record may be avoided if the transaction is not a distributed one or is read-only.

5.4 Checkpoints

Periodically, checkpoints are taken to reduce the amount of work that needs to be performed during restart recovery. The work may relate to the extent of the log that needs to be examined, the number of data pages that have to be read from nonvolatile storage, etc. Checkpoints can be taken asynchronously (i.e., while transaction processing, including updates, is going on). Such a *fuzzy checkpoint* is initiated by writing a *begin_chkpt* record. Then the *end_chkpt* record is constructed by including in it the contents of the normal transaction table, the BP dirty_pages table, and any file mapping information for the objects (like tablespace, indexspace, etc.) that are "open" (i.e., for which BP dirty_pages table has entries). Only for simplicity of exposition, we assume that all the information can be accommodated in a single end_chkpt record. It is easy to deal with the case where multiple records are needed to log this information. Once the end_chkpt record is constructed, it is written to the log. Once that record reaches stable storage, the LSN of the begin_chkpt record is stored in the *master record* which is in a well-known place on stable storage. If a failure were to occur before the end_chkpt record migrates to stable storage, but after the begin_chkpt record migrates to stable storage, then that checkpoint is considered an *incomplete checkpoint*. Between the *begin_chkpt* and *end_chkpt* log records, transactions might have written other log records. If one or more transactions are likely to remain in the *in-doubt* state for a long time because of prolonged loss of contact with the commit coordinator, then it is a good idea to include in the end_chkpt record information about the update-type locks (e.g., X, IX and SIX) held by those transactions. This way, if a failure were to occur, then, during restart recovery, those locks could be reacquired without having to access the prepare records of those transactions.

Since latches may need to be acquired to read the dirty_pages table correctly while gathering the needed information, it is a good idea to gather the information a little at a time to reduce contention on the tables. For example, if the dirty_pages table has 1000 rows, during each latch acquisition 100 entries can be examined. If the already examined entries change before the end of the checkpoint, the recovery algorithms remain correct (see Figure 10). This is because, in computing the restart redo point, besides taking into account the minimum of the RecLSNs of the dirty pages included in the end_chkpt record, ARIES also takes into account the log records that were written by transactions since the beginning of the checkpoint. This is important because the effect of some of the updates that were performed since

the initiation of the checkpoint might not be reflected in the dirty page list that is recorded as part of the checkpoint.

ARIES does not require that any dirty pages be forced to nonvolatile storage during a checkpoint. The assumption is that the buffer manager is, on a continuous basis, writing out dirty pages in the background using system processes. The buffer manager can batch the writes and write multiple pages in one I/O operation. [96] gives details about how DB2 manages its buffer pools in this fashion. Even if there are some hot-spot pages which are frequently modified, the buffer manager has to ensure that those pages are written to nonvolatile storage reasonably often to reduce restart redo work, just in case a system failure were to occur. To avoid the prevention of updates to such hot-spot pages during an I/O operation, the buffer manager could make a copy of each of those pages and perform the I/O from the copy. This minimizes the data unavailability time for writes.

6. RESTART PROCESSING

When the transaction system restarts after a failure, recovery needs to be performed to bring the data to a consistent state and ensure the atomicity and durability properties of transactions. Figure 9 describes the *RESTART* routine that gets invoked at the beginning of the restart of a failed system. The input to this routine is the LSN of the master record which contains the pointer to the begin_chkpt record of the last complete checkpoint taken before site failure or shutdown. This routine invokes the routines for the analysis pass, the redo pass and the undo pass, in that order. The buffer pool dirty_pages table is updated appropriately. At the end of restart recovery, a checkpoint is taken.

For high availability, the duration of restart processing must be as short as possible. One way of accomplishing this is by exploiting parallelism during the redo and undo passes. Only if parallelism is going to be employed is it necessary to latch pages before they are modified during restart recovery. Ideas for improving data availability by allowing new transaction processing during recovery are explored in [60].

6.1 Analysis Pass

The first pass of the log that is made during restart recovery is the *analysis pass*. Figure 10 describes the *RESTART_ANALYSIS* routine that implements the analysis pass actions. The input to this routine is the LSN of the *master* record. The outputs of this routine are the transaction table, which contains the list of transactions which were in the *in-doubt* or unprepared state at the time of system failure or shutdown; the dirty_pages table, which contains the list of pages that were potentially dirty in the buffers when the system failed or was shut down; and the *RedoLSN*, which is the location on the log from which the redo pass must start processing the log. The only log records that may be written by this routine are end records for transactions that had totally rolled back before system failure, but for whom end records are missing.

```
RESTART(Master_Addr);
Restart_Analysis(Master_Addr,Trans_Table, Dirty_Pages, RedoLSN);
Restart_Redo(RedoLSN, Trans_Table, Dirty_Pages);
buffer pool Dirty_Pages table := Dirty_Pages;
remove entries for non-buffer-resident pages from the buffer pool Dirty_Pages table;
Restart_Undo(Trans_Table);
reacquire locks for prepared transactions;
checkpoint();
RETURN;
```

Fig. 9. Pseudocode for restart.

During this pass, if a log record is encountered for a page whose identity does not already appear in the dirty_pages table, then an entry is made in the table with the current log record's LSN as the page's RecLSN. The transaction table is modified to track the state changes of transactions and also to note the LSN of the most recent log record that would need to be undone if it were determined ultimately that the transaction had to be rolled back. If an OSfile_return log record is encountered, then any pages belonging to that file which are in the dirty_pages table are removed from the latter in order to make sure that no page belonging to that version of that file is accessed during the redo pass. The same file may be recreated and updated later, once the original operation causing the file erasure is committed. In that case, some pages of the recreated file will reappear in the dirty_pages table later with RecLSN values greater than the end_of_log LSN when the file was erased. The RedoLSN is the minimum RecLSN from the dirty_pages table at the end of the analysis pass. The redo pass can be skipped if there are no pages in the dirty_pages table.

It is not necessary that there be a separate analysis pass and, in fact, in the ARIES implementation in the OS/2 Extended Edition Database Manager there is no analysis pass. This is especially because, as we mentioned before (see also Section 6.2), in the redo pass, ARIES unconditionally redoes all missing updates. That is, it redoes them irrespective of whether they were logged by loser or nonloser transactions, unlike System R, SQL/DS and DB2. Hence, redo does not need to know the loser or nonloser status of a transaction. That information is, strictly speaking, needed only for the undo pass. This would not be true for a system (like DB2) in which for in-doubt transactions their update locks are reacquired by inferring the lock names from the log records of the in-doubt transactions, as they are encountered during the redo pass. This technique for reacquiring locks forces the RedoLSN computation to consider the Begin_LSNs of in-doubt transactions which in turn requires that we know, before the start of the redo pass, the identities of the in-doubt transactions.

Without the analysis pass, the transaction table could be constructed from the checkpoint record and the log records encountered during the redo pass. The RedoLSN would have to be the minimum(minimum(RecLSN from the dirty_pages table in the end_chkpt record), LSN(begin_chkpt record)). Suppression of the analysis pass would also require that other methods be used to

```
RESTART_ANALYSIS(Master_Addr, Trans_Table, Dirty_Pages, RedoLSN);
initialize the tables Trans_Table and Dirty_Pages to empty;
Master_Rec := Read_Disk(Master_Addr);
Open_Log_Scan(Master_Rec.ChkptLSN);                    /* open log scan at Begin_Chkpt record */
LogRec := Next_Log();                                       /* read in the Begin_Chkpt record */
LogRec := Next_Log();                                 /* read log record following Begin_Chkpt */
WHILE NOT(End_of_Log) DO;
  IF trans related record & LogRec.TransID NOT in Trans_Table THEN /* not chkpt/OSfile_return*/
    insert (LogRec.TransID,'U',LogRec.LSN,LogRec.PrevLSN) into Trans_Table;    /* log record */
  SELECT(LogRec.Type)
    WHEN('update' | 'compensation') DO;
      Trans_Table[LogRec.TransID].LastLSN := LogRec.LSN;
      IF LogRec.Type = 'update' THEN
        IF LogRec is undoable THEN Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.LSN;
      ELSE Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.UndoNxtLSN;
                                  /* next record to undo is the one pointed to by this CLR */
      IF LogRec is redoable & LogRec.PageID NOT IN Dirty_Pages THEN
        insert (LogRec.PageID, LogRec.LSN) into Dirty_Pages;
    END; /* WHEN('update' | 'compensation') */
    WHEN('Begin_Chkpt') ;  /* found an incomplete checkpoint's Begin_Chkpt record. ignore it */
    WHEN('End_Chkpt') DO;
      FOR each entry in LogRec.Tran_Table DO;
        IF TransID NOT IN Trans_Table THEN DO;
          insert entry(TransID,State,LastLSN,UndoNxtLSN) in Trans_Table;
        END;
      END; /* FOR */
      FOR each entry in LogRec.Dirty_PageLst DO;
        IF PageID NOT IN Dirty_Pages THEN insert entry(PageID,RecLSN) in Dirty_Pages;
        ELSE set RecLSN of Dirty_Pages entry to RecLSN in Dirty_PageLst;
      END; /* FOR */
    END; /* WHEN('End_Chkpt') */
    WHEN('prepare' | 'rollback') DO;
      IF LogRec.Type = 'prepare' THEN Trans_Table[LogRec.TransID].State := 'P';
      ELSE Trans_Table[LogRec.TransID].State := 'U';
      Trans_Table[LogRec.TransID].LastLSN := LogRec.LSN;
    END; /* WHEN('prepare' | 'rollback') */
    WHEN('end') delete Trans_Table entry for which TransID = LogRec.TransID;
    WHEN('OSfile_return') delete from Dirty_Pages all pages of returned file;
  END; /* SELECT */
  LogRec := Next_Log();
END; /* WHILE */
FOR EACH Trans_Table entry with (State = 'U') & (UndoNxtLSN = 0) DO; /* rolled back trans    */
  write end record and remove entry from Trans_Table;            /* with missing end record */
END; /* FOR */
RedoLSN := minimum(Dirty_Pages.RecLSN);                    /* return start position for redo */
RETURN;
```

Fig. 10. Pseudocode for restart analysis.

avoid processing updates to files which have been returned to the operating system. Another consequence is that the dirty_pages table used during the redo pass cannot be used to filter update log records which occur after the begin_chkpt record.

6.2 Redo Pass

The second pass of the log that is made during restart recovery is the *redo pass*. Figure 11 describes the *RESTART_REDO* routine that implements

```
RESTART_REDO(RedoLSN, Dirty_Pages);
Open_Log_Scan(RedoLSN);                          /* open log scan and position at restart pt */
LogRec := Next_Log();                             /* read log record at restart redo point   */
WHILE NOT(End_of_Log) DO;                            /* look at all records till end of log */
  IF LogRec.Type = ('update'||'compensation') & LogRec is redoable &
     LogRec.PageID IN Dirty_Pages & LogRec.LSN >= Dirty_Pages[LogRec.PageID].RecLSN
    THEN DO;              /* a redoable page update. updated page might not have made it to */
                          /* disk before sys failure. need to access page and check its LSN */
      Page := fix&latch(LogRec.PageID,'X');
      IF Page.LSN < LogRec.LSN THEN DO                /* update not on page. need to redo it */
        Redo_Update(Page,LogRec);                                            /* redo update */
        Page.LSN := LogRec.LSN;
      END;                                                                  /* redid update */
      ELSE Dirty_Pages[LogRec.PageID].RecLSN := Page.LSN+1;       /* update already on page */
                     /* update dirty page list with correct info. this will happen if this */
                     /* page was written to disk after the checkpt but before sys failure  */
      unfix&unlatch(Page);
    END;                                                   /* LSN on page has to be checked */
  LogRec := Next_Log();                                              /* read next log record */
END;                                                                /* reading till end of log */
RETURN;
```

Fig. 11. Pseudocode for restart redo.

the redo pass actions. The inputs to this routine are the RedoLSN and the dirty_pages table supplied by the restart_analysis routine. No log records are written by this routine. The redo pass starts scanning the log records from the RedoLSN point. When a redoable log record is encountered, a check is made to see if the referenced page appears in the dirty_pages table. If it does and if the log record's LSN is greater than or equal to the RecLSN for the page in the table, then it is suspected that the page state might be such that the log record's update might have to be redone. To resolve this suspicion, the page is accessed. If the page's LSN is found to be less than the log record's LSN, then the update is redone. Thus, the RecLSN information serves to limit the number of pages which have to be examined. This routine reestablishes the database state as of the time of system failure. Even updates performed by loser transactions are redone. The rationale behind this repeating of history is explained in Section 10.1. It turns out that some of that redo of loser transactions' log records may be unnecessary. In [69] we have explored further the idea of restricting the repeating of history to possibly reduce the number of pages which get dirtied during this pass.

Since redo is page-oriented, only the pages with entries in the dirty_pages table may get modified during the redo pass. Only the pages listed in the dirty_pages table will be read and examined during this pass. Not all the pages that are read may require redo. This is because some of the pages that were dirty at the time of the last checkpoint or which became dirty later might have been written to nonvolatile storage before the system failure. Because of reasons like reducing log volume and saving some CPU overhead, we do not expect systems to write log records that identify the dirty pages that were written to nonvolatile storage, although that option is available and such log records can be used to eliminate the corresponding pages from

the dirty_pages table when those log records are encountered during the analysis pass. Even if such records were always to be written after I/Os complete, a system failure in a narrow window could prevent them from being written. The corresponding pages will not get modified during this pass.

For brevity, we do not discuss here as to how, if a failure were to occur after the logging of the end record of a transaction, but before the execution of all the pending actions of that transaction, the remaining pending actions are redone during the redo pass.

For exploiting parallelism, the availability of the information in the dirty_pages table gives us the possibility of initiating asynchronous I/Os in parallel to read all these pages so that they may be available in the buffers possibly before the corresponding log records are encountered in the redo pass. Since updates performed during the redo pass are not logged, we can also perform sophisticated things like building in-memory queues of log records which potentially need to be reapplied (as dictated by the information in the dirty_pages table) on a per page or group of pages basis and, as the asynchronously initiated I/Os complete and pages come into the buffer pool, processing the corresponding log record queues using multiple processes. This requires that each queue be dealt with by only one process. Updates to *different* pages may get applied in different orders from the order represented in the log. This does not violate any correctness properties since for a given page all its missing updates are reapplied in the same order as before. These parallelism ideas are also applicable to the context of supporting disaster recovery via remote backups [73].

6.3 Undo Pass

The third pass of the log that is made during restart recovery is the *undo pass*. Figure 12 describes the *RESTART_UNDO* routine that implements the undo pass actions. The input to this routine is the restart transaction table. The dirty_pages table is not consulted during this undo pass. Also, since history is repeated before the undo pass is initiated, the LSN on the page is not consulted to determine whether an undo operation should be performed or not. Contrast this with what we describe in Section 10.1 for systems like DB2 that do not repeat history but perform selective redo.

The restart_undo routine rolls back losers transactions, in reverse chronological order, in a single sweep of the log. This is done by continually taking the maximum of the LSNs of the next log record to be processed for each of the yet-to-be-completely-undone loser transactions, until no loser transaction remains to be undone. The next record to process for each transaction to be rolled back is determined by an entry in the transaction table for each of those transactions. The processing of the encountered log records is exactly as we described before in Section 5.2. In the process of rolling back the transactions, this routine writes CLRs. The buffer manager follows the usual WAL protocol while writing dirty pages to nonvolatile storage during the undo pass.

```
RESTART_UNDO(Trans_Table);
WHILE EXISTS(Trans with State = 'U' in Trans_Table) DO;
  UndoLSN := maximum(UndoNxtLSN) from Trans_Table entries with State = 'U';
                          /* pick up UndoNxtLSN of unprepared trans with maximum UndoNxtLSN */
  LogRec := Log_Read(UndoLSN);                        /* read log record to be undone or a CLR */
  SELECT(LogRec.Type)
    WHEN('update') DO;
      IF LogRec is undoable THEN DO;          /* record needs undoing (not redo-only record) */
        Page := fix&latch(LogRec.PageID,'X');
        Undo_Update(Page,LogRec);
        Log_Write('compensation',LogRec.TransID,Trans_Table[LogRec.TransID].LastLSN,
              LogRec.PageID,LogRec.PrevLSN, ...,LgLSN,Data);                    /* write CLR */
        Page.LSN := LgLSN;                                          /* store LSN of CLR in page */
        Trans_Table[LogRec.TransID].LastLSN := LgLSN;              /* store LSN of CLR in table */
        unfix&unlatch(Page);
      END;                                                              /* undoable record case */
      ELSE;                                                /* record cannot be undone - ignore it */
      Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.PrevLSN; /* next record to process is */
                                  /* the one preceding this record in its backward chain */
      IF LogRec.PrevLSN = 0 THEN DO;                   /* have undone completely - write end  */
        Log_Write('end',LogRec.TransID,Trans_Table[LogRec.TransID].LastLSN,...);
        delete Trans_Table entry where TransID = LogRec.TransID;  /* delete trans from table */
      END;                                                              /* trans fully undone */
    END; /* WHEN('update') */
    WHEN('compensation') Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.UndoNxtLSN;
                                                       /* pick up addr of next record to examine */
    WHEN('rollback'|'prepare') Trans_Table[LogRec.TransID].UndoNxtLSN := LogRec.PrevLSN;
                                                       /* pick up addr of next record to examine */
  END; /* SELECT */
END;   /* WHILE */
RETURN;
```

Fig. 12. Pseudocode for estart undo.

To exploit parallelism, the undo pass can also be performed using multiple processes. It is important that each transaction be dealt with completely by a single process because of the UndoNxtLSN chaining in the CLRs. This still leaves open the possibility of writing the CLRs first, without applying the undos to the pages (see Section 6.4 for problems in accomplishing this for objects that may require logical undos), and then redoing the CLRs in parallel, as explained in Section 6.2. In this fashion, the undo work of actually applying the changes to the pages can be performed in parallel, even for a single transaction.

Figure 13 depicts an example restart recovery scenario using ARIES. Here, all the log records describe updates to the same page. Before the failure, the page was written to disk after the second update. After that disk write, a partial rollback was performed (undo of log records 4 and 3) and then the transaction went forward (updates 5 and 6). During restart recovery, the missing updates (3, 4, 4′, 3′, 5 and 6) are first redone and then the undos (of 6, 5, 2 and 1) are performed. Each update log record will be matched with at most one CLR, regardless of how many times restart recovery is performed.

With ARIES, we have the option of allowing the continuation of loser transactions after restart recovery is completed. Since ARIES repeats history and supports the savepoint concept, we could, in the undo pass, roll back each

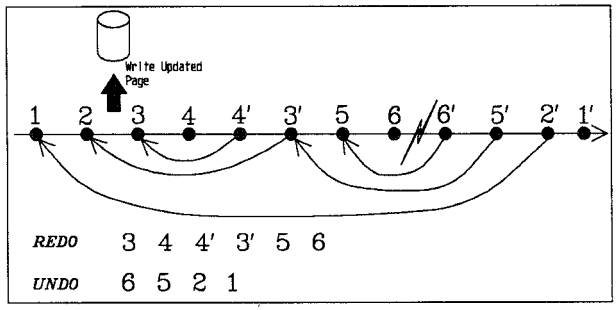


Fig. 13. Restart recovery example with ARIES.

loser only to its latest savepoint, instead of totally rolling back the loser transactions. Later, we could resume the transaction by invoking its application at a special entry point and passing enough information about the savepoint from which execution is to be resumed. Doing this correctly would require (1) the ability to generate lock names from the transaction's log records for its uncommitted, not undone updates, (2) reacquiring those locks before completing restart recovery, and (3) logging enough information whenever savepoints are established so that the system can restore cursor positions, application program state, and so on.

6.4 Selective or Deferred Restart

Sometimes, after a system failure, we may wish to restart the processing of new transactions as soon as possible. Hence, we may wish to defer doing some recovery work to a later point in time. This is usually done to reduce the amount of time during which some critical data is unavailable. It is accomplished by recovering such data first and then opening the system for the processing of new transactions. In DB2, for example, it is possible to perform restart recovery even when some of the objects for which redo and/or undo work needs to be performed are offline when the system is brought up. If some undo work needs to be performed for some loser transactions on those offline objects, then DB2 is able to write the CLRs alone and finish handling the transactions. This is possible because the CLRs can be generated based solely on the information in the non-CLR records written during the forward processing of the transactions [15]. Because page (or minipage, for indexes) is the smallest granularity of locking, the undo actions will be exact inverses of the original actions. That is, there are no logical undos in DB2. DB2 remembers, in an exceptions table (called the database allocation (DBA) table) that is maintained in the log and in virtual storage, the fact that those offline objects need to be recovered when they are brought online, before they are made accessible to other transactions [14]. The LSN ranges of log records to be applied are also remembered. Unless there are some in-doubt transactions with uncommitted updates to those objects, no locks need to be acquired to protect those objects since accesses to those objects will not be permitted until recovery is completed. When those objects are brought online, then

recovery is performed efficiently by rolling forward using the log records in the remembered ranges. Even during normal rollbacks, CLRs may be written for offline objects.

In ARIES also, we can take similar actions, provided none of the loser transactions has modified one or more of the offline objects that may require logical undos. This is because logical undos are based on the current state of the object. Redos are not at all a problem, since they are always page-oriented. For logical undos involving space management (see Section 10.3), generally we can take a conservative approach and generate the appropriate CLRs. For example, during the undo of an insert record operation, we can write a CLR for the space-related update stating that the page is 0% full. But for the high concurrency, index management methods of [62] this is not possible, since the effect of the logical undo (e.g., retraversing the index tree to do a key deletion), in terms of which page may be affected, is unpredictable; in fact, we cannot even predict when page-oriented undo will not work and hence logical undo is necessary.

It is not possible to handle the undos of some of the records of a transaction during restart recovery and handle the undos (possibly, logical) of the rest of the records at a later point in time, if the two sets of records are interspersed. Remember that in all the recovery methods, undo of a transaction is done in reverse chronological order. Hence, it is enough to remember, for each transaction, the next record to be processed during the undo; from that record, the PrevLSN and/or the UndoNxtLSN chain leads us to all the other records to be processed.

Even under the circumstances where one or more of the loser transactions have to perform, potentially logical, undos on some offline objects, if deferred restart needs to be supported, then we suggest the following algorithm:

1. Perform the repeating of history for the *online* objects, as usual; postpone it for the *offline* objects and remember the log ranges.
2. Proceed with the undo pass as usual, but stop undoing a loser transaction when one of its log records is encountered for which a CLR cannot be generated for the above reasons. Call such a transaction a *stopped transaction*. But continue undoing the other, unstopped transactions.
3. For the stopped transactions, acquire locks to protect their updates which have not yet been undone. This could be done as part of the undo pass by continuing to follow the pointers, as usual, even for the stopped transactions and acquiring locks based on the encountered non-CLRs that were written by the stopped transactions.
4. When restart recovery is completed and later the previously offline objects are made online, first repeat history based on the remembered log ranges and then continue with the undoing of the stopped transactions. After each of the stopped transactions is totally rolled back, release its still held locks.
5. Whenever an offline object becomes online, when the repeating of history is completed for that object, new transactions can be allowed to access that object in parallel with the further undoing of all of the stopped transactions that can make progress.

The above requires the ability to generate lock names based on the information in the update (non-CLR) log records. DB2 is doing that already for in-doubt transactions.

Even if none of the objects to be recovered is offline, but it is desired that the processing of new transactions start before the rollbacks of the loser transactions are completed, then we can accommodate it by doing the following: (1) first repeat history and reacquire, based on their log records, the locks for the uncommitted updates of the loser and in-doubt transactions, and (2) then start processing new transactions even as the rollbacks of the loser transactions are performed in parallel. The locks acquired in step (1) are released as each loser transaction's rollback completes. Performing step (1) requires that the restart RedoLSN be adjusted appropriately to ensure that all the log records of the loser transactions are encountered during the redo pass. If a loser transaction was already rolling back at the time of the system failure, then, with the information obtained during the analysis pass for such a transaction, it will be known as to which log records remain to be undone. These are the log records whose LSNs are less than or equal to the UndoNxtLSN of the transaction's last CLR. Locks need to be obtained during the redo pass only for those updates that have not yet been undone.

If a long transaction is being rolled back and we would like to release some of its locks as soon as possible, then we can mark specially those log records which represent the first update by that transaction on the corresponding object (e.g., record, if record locking is in effect) and then release that object's lock as soon as the corresponding log record is undone. This works only because we do not undo CLRs and because we do not undo the same non-CLR more than once; hence, it will not work in systems that undo CLRs (e.g., Encompass, AS/400, DB2) or that undo a non-CLR more than once (e.g., IMS). This early release of locks can be performed in ARIES during normal transaction undo to possibly permit resolution of deadlocks using partial rollbacks.

7. CHECKPOINTS DURING RESTART

In this section, we describe how the impact of failures on CPU processing and I/O can be reduced by, optionally, taking checkpoints during different stages of restart recovery processing.

Analysis pass. By taking a checkpoint at the end of the analysis pass, we can save some work if a failure were to occur during recovery. The entries of the transaction table of this checkpoint will be the same as the entries of the transaction table at the end of the analysis pass. The entries of the dirty_pages list of this checkpoint will be the same as the entries that the *restart* dirty_pages table contains at the end of the analysis pass. This is different from what happens during a normal checkpoint. For the latter, the dirty_pages list is obtained from the buffer pool (BP) dirty_pages table.

Redo pass. At the beginning of the redo pass, the buffer manager (BM) is notified so that, whenever it writes out a modified page to nonvolatile storage during the redo pass, it will change the restart dirty_pages table entry for that page by making the RecLSN be equal to the LSN of that log record such

that all log records up to that log record had been processed. It is enough if BM manipulates the restart dirty_pages table in this fashion. BM does not have to maintain its own dirty_pages table as it does during normal processing. Of course, it should still be keeping track of what pages are currently in the buffers. The above allow checkpoints to be taken any time during the redo pass to reduce the amount of the log that would need to be redone if a failure were to occur before the end of the redo pass. The entries of the dirty_pages list of this checkpoint will be the same as the entries of the *restart* dirty_pages table at the time of the checkpoint. The entries of the transaction table of this checkpoint will be the same as the entries of the transaction table at the end of the analysis pass. This checkpointing is not affected by whether or not parallelism is employed in the redo pass.

Undo pass. At the beginning of the undo pass, the restart dirty_pages table becomes the BP dirty_pages table. At this point, the table is cleaned up by removing those entries for which the corresponding pages are no longer in the buffers. From then onward, the BP manager manipulates this table as it does during normal processing—removing entries when pages are written to nonvolatile storage, adding entries when pages are about to become dirty, etc. During the undo pass, the entries of the transaction table are modified as during normal undo. If a checkpoint is taken any time during the undo pass, then the entries of the dirty_pages list of that checkpoint are the same as the entries of the BP dirty_pages table at the time of the checkpoint. The entries of the transaction table of this checkpoint will be the same as the entries of the transaction table at that time.

In System R, during restart recovery, sometimes it may be required that a checkpoint be taken to free up some physical pages (the shadow pages) for more undo or redo work to be performed. This is another consequence of the fact that history cannot be repeated in System R. This complicates the restart logic since the view depicted in Figure 17 would no longer be true after a restart checkpoint completes. The restart checkpoint logic and its effect on a restart following a system failure during an earlier restart were considered too complex to be describable in [31]. ARIES is able to easily accommodate checkpoints during restart. While these checkpoints are optional in our case, they may be forced to take place in System R.

8. MEDIA RECOVERY

We will assume that media recovery will be required at the level of a file or some such (like DBspace, tablespace, etc.) entity. A *fuzzy image copy* (also called a *fuzzy archive dump*) operation involving such an entity can be performed concurrently with modifications to the entity by other transactions. With such a high concurrency image copy method, the image copy might contain some uncommitted updates, in contrast to the method of [52]. Of course, if desired, we could also easily produce an image copy with no uncommitted updates. Let us assume that the image copying is performed directly from the nonvolatile storage version of the entity. This means that

more recent versions of some of the copied pages may be present in the transaction system's buffers. Copying directly from the nonvolatile storage version of the object would usually be much more efficient since the device geometry can be exploited during such a copy operation and since the buffer manager overheads will be eliminated. Since the transaction system does not have to be up for the direct copying, it may also be more convenient than copying via the transaction system's buffers. If the latter is found desirable (e.g., to support incremental image copying, as described in [13]), then it is easy to modify the presented method to accommodate it. Of course, in that case, some minimal amount of synchronization will be needed. For example, latching at the page level, but no locking will be needed.

When the fuzzy image copy operation is initiated, the location of the begin_chkpt record of the most recent complete checkpoint is noted and remembered along with the image copy data. Let us call this checkpoint the *image copy checkpoint*. The assertion that can be made based on this checkpoint information is that all updates that had been logged in log records with LSNs less than minimum(minimum(RecLSNs of dirty pages of the image-copied entity in the image copy checkpoint's end_chkpt record), LSN(begin_chkpt record of the image copy checkpoint)) would have been externalized to nonvolatile storage by the time the fuzzy image copy operation began. Hence, the image-copied version of the entity would be at least as up to date as of that point in the log. We call that point the *media recovery redo point*. The reason for taking into account the LSN of the begin_chkpt record in computing the media recovery redo point is the same as the one given in Section 5.4 while discussing the computation of the restart redo point.

When media recovery is required, the image-copied version of the entity is reloaded and then a redo scan is initiated starting from the media recovery redo point. During the redo scan, all the log records relating to the entity being recovered are processed and the corresponding updates are applied, unless the information in the image copy checkpoint record's dirty_pages list or the LSN on the page makes it unnecessary. Unlike during *restart* redo, if a log record refers to a page that is not in the dirty_pages list and the log record's LSN is greater than the LSN of the begin_chkpt log record of the image copy checkpoint, then that page must be accessed and its LSN compared to the log record's LSN to check if the update must be redone. Once the end of the log is reached, if there are any in-progress transactions, then those transactions that had made changes to the entity are undone, as in the undo pass of restart recovery. The information about the identities, etc. of such transactions may be kept separately somewhere (e.g., in an exceptions table such as the DBA table in DB2—see Section 6.4) or may be obtained by performing an analysis pass from the last complete checkpoint in the log until the end of the log.

Page-oriented logging provides recovery independence amongst objects. Since, in ARIES, every database page's update is logged separately, even if an arbitrary database page is damaged in the nonvolatile storage and the page needs recovery, the recovery can be accomplished easily by extracting

an earlier copy of that page from an image copy and rolling forward that version of the page using the log as described above. This is to be contrasted with systems like System R in which, since for some pages' updates (e.g., index and space management pages') log records are not written, recovery from damage to such a page may require the expensive operation of reconstructing the entire object (e.g., rebuilding the complete index even when only one page of an index is damaged). Also, even for pages for which logging is performed explicitly (e.g., data pages in System R), if CLRs are not written when undo is performed, then bringing a page's state up to date by starting from the image copy state would require paying attention to the log records representing the transaction state (commit, partial or total rollback) to determine what actions, if any, should be undone. If any transactions had rolled back partially or totally, then backward scans of such transactions would be required to see if they made any changes to the page being recovered so that they are undone. These backward scans may result in useless work being performed, if it turns out that some rolled back transaction had not made any changes to the page being recovered. An alternative would be to preprocess the log and place forward pointers to skip over rolled back log records, as it is done in System R during the analysis pass of restart recovery (see Section 10.2 and Figure 18).

Individual pages of the database may be corrupted not only because of media problems but also because of an abnormal process termination while the process is actively making changes to a page in the buffer pool and before the process gets a chance to write a log record describing the changes. If the database code is executed by the application process itself, which is what performance-conscious systems like DB2 implement, such abnormal terminations may occur because of the user's interruption (e.g., by hitting the *attention* key) or due to the operating system's action on noting that the process had exhausted its CPU time limit. It is generally an expensive operation to put the process in an uninterruptable state before every page update. Given all these circumstances, an efficient way to recover the corrupted page is to read the uncorrupted version of the page from the nonvolatile storage and bring it up to date by rolling forward the page state using all relevant log records for that page. The roll-forward redo scan of the log is started from the RecLSN remembered for the buffer by the buffer manager. DB2 does this kind of internal recovery operation automatically [15]. The corruption of a page is detected by using a bit in the page header. The bit is set to '1' after the page is fixed and X-latched. Once the update operation is complete (i.e., page updated, update logged and page LSN modified), the bit is reset to '0'. Given this, whenever a page is latched, for read or write, first this bit is tested to see if its value is equal to '1', in which case automatic page recovery is initiated. From an availability viewpoint, it is unacceptable to bring down the entire transaction system to recover from such a *broken page* situation by letting restart recovery redo all those logged updates that were in the corrupted page but were missing in the uncorrupted version of the page on nonvolatile storage. A related problem is to make sure that for those pages that were left in the fixed state by the abnormally

terminating process, unfix calls are issued by the transaction system. By leaving enough *footprints* around before performing operations like fix, unfix and latch, the user process aids system processes in performing the necessary clean-ups.

For the variety of reasons mentioned in this section and elsewhere, writing CLRs is a very good idea even if the system is supporting only page locking. This is to be contrasted with the no-CLRs approach, suggested in [52], which supports only page locking.

9. NESTED TOP ACTIONS

There are times when we would like some updates of a transaction to be committed, irrespective of whether the transaction ultimately commits or not. We do need the atomicity property for these updates themselves. This is illustrated in the context of file extension. After a transaction extends a file which causes updates to some system data in the database, other transactions may be allowed to use the extended area prior to the commit of the extending transaction. If the extending transaction were to roll back, then it would not be acceptable to undo the effects of the extension. Such an undo might very well lead to a loss of updates performed by the other committed transactions. On the other hand, if the extension-related updates to the system data in the database were themselves interrupted by a failure before their completion, it is necessary to undo them. These kinds of actions have been traditionally performed by starting independent transactions, called *top actions* [51]. A transaction initiating such an independent transaction waits until that independent transaction commits before proceeding. The independent transaction mechanism is, of course, vulnerable to lock conflicts between the initiating transaction and the independent transaction, which would be unacceptable.

In ARIES, using the concept of a *nested top action*, we are able to support the above requirement very efficiently, without having to initiate independent transactions to perform the actions. A nested top action, for our purposes, is taken to mean any subsequence of actions of a transaction which should not be undone once the sequence is complete and some later action which is dependent on the nested top action is logged to stable storage, irrespective of the outcome of the enclosing transaction.

A transaction execution performing a sequence of actions which define a nested top action consists of the following steps:

(1) ascertaining the position of the current transaction's last log record;
(2) logging the redo and undo information associated with the actions of the nested top action; and
(3) on completion of the nested top action, writing a *dummy CLR* whose UndoNxtLSN points to the log record whose position was remembered in step (1).

We assume that the effects of any actions like creating a file and their associated updates to system data normally resident outside the database are externalized, before the dummy CLR is written. When we discuss redo, we are referring to only the system data that is resident in the database itself.

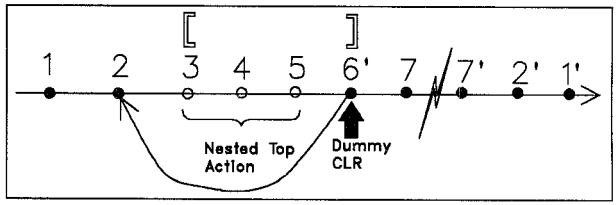


Fig. 14. Nested top action example.

Using this nested top action approach, if the enclosing transaction were to roll back after the completion of the nested top action, then the dummy CLR will ensure that the updates performed as part of the nested top action are not undone. If a system failure were to occur before the dummy CLR is written, then the incomplete nested top action will be undone since the nested top action's log records are written as undo-redo (as opposed to redo-only) log records. This provides the desired atomicity property for the nested top action. Unlike for the normal CLRs, there is nothing to redo when a dummy CLR is encountered during the redo pass. The dummy CLR in a sense can be thought of as the commit record for the nested top action. The advantage of our approach is that the enclosing transaction need not wait for this record to be forced to stable storage before proceeding with its subsequent actions.[6] Also, we do not pay the price of starting a new transaction. Nor do we run into lock conflict problems. Contrast this approach with the costly independent-transaction approach.

Figure 14 gives an example of a nested top action consisting of the actions 3, 4 and 5. Log record 6′ acts as the dummy CLR. Even though the enclosing transaction's activity is interrupted by a failure and hence it needs to be rolled back, 6′ ensures that the nested top action is not undone.

It should be emphasized that the nested top action implementation relies on repeating history. If the nested top action consists of only a single update, then we can log that update using a single *redo-only* log record and avoid writing the dummy CLR. Applications of the nested top action concept in the context of a hash-based storage method and index management can be found in [59, 62].

10. RECOVERY PARADIGMS

This section describes some of the problems associated with providing fine-granularity (e.g., record) locking and handling transaction rollbacks. Some additional discussion can be found in [97]. Our aim is to show how certain features of the existing recovery methods caused us difficulties in accomplishing our goals and to motivate the need for certain features which we had to include in ARIES. In particular, we show why some of the recovery paradigms of System R, which were developed in the context of the shadow page

[6] The dummy CLR may have to be forced if some *unlogged* updates may be performed later by other transactions which depended on the nested top action having completed.

technique, are inappropriate when WAL is to be used and there is a need for high levels of concurrency. In the past, one or more of those System R paradigms have been adopted in the context of WAL, leading to the design of algorithms with limitations and/or errors [3, 15, 16, 52, 71, 72, 78, 82, 88]. The System R paradigms that are of interest are:

—selective redo during restart recovery.
—undo work preceding redo work during restart recovery.
—no logging of updates performed during transaction rollback (i.e., no CLRs).
—no logging of index and space management information changes.
—no tracking of page state on page itself to relate it to logged updates (i.e., no LSNs on pages).

10.1 Selective Redo

The goal of this subsection is to introduce the concept of selective redo that has been implemented in many systems and to show the problems that it introduces in supporting fine-granularity locking with WAL-based recovery. The aim is to motivate why ARIES repeats history.

When transaction systems restart after failures, they generally perform database recovery updates in 2 passes of the log: a redo pass and an undo pass (see Figure 6). System R first performs the undo pass and then the redo pass. As we will show later, the System R paradigm of *undo preceding redo is incorrect with WAL and fine-granularity locking*. The WAL-based DB2, on the other hand, does just the opposite. During the redo pass, System R redoes only the actions of committed and prepared (i.e., in-doubt) transactions [31]. We call this *selective redo*. While the selective redo paradigm of System R intuitively seems to be the efficient approach to take, it has many pitfalls, as we discuss below.

Some WAL-based systems, such as DB2, support only page locking and perform selective redo [15]. This approach will lead to data inconsistencies in such systems, if record locking were to be implemented. Let us consider a WAL technique in which each page contains an LSN as described before. During the redo pass, the page LSN is compared to the LSN of a log record describing an update to the page to determine whether the log record's update needs to be reapplied to the page. If the page LSN is less than the log record's LSN, then the update is redone and the page's LSN is set to the log record's LSN (see Figure 15). During the undo pass, if the page LSN is less than the LSN of the log record to be undone, then no undo action is performed on the page. Otherwise, undo is performed on the page. Whether or not undo needs to be actually performed on the page, a CLR describing the updates that would have been performed as part of the undo operation is always written, when the transaction's actions are being rolled back. The CLR is written, even when the page does not contain the update, just to make media recovery simpler and not force it to handle rolled back updates in a special way. Writing the CLR when an undo is not actually performed on the page turns out to be necessary also when handling a failure of the system

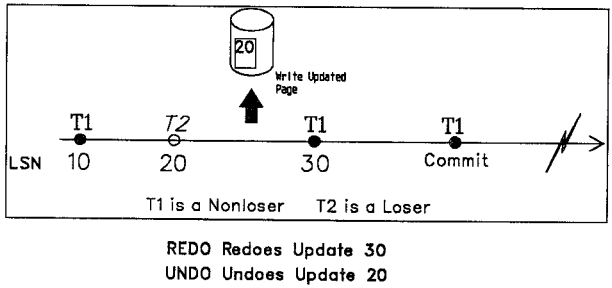


Fig. 15. Selective redo with WAL—problem-free scenario.

during restart recovery. This will happen, if there was an update U2 for page P1 which did not have to be undone, but there was an earlier update U1 for P1 which had to be undone, resulting in U1′ (CLR for U1) being written and P1's LSN being changed to the LSN of U1′ (> LSN of U2). After that, if P1 were to be written to nonvolatile storage before a system failure interrupts the completion of this restart, then, during the next restart, it would appear as if P1 contains the update U2 and an attempt would be made to undo it. On the other hand, if U2′ had been written, then there would not be any problem. It should be emphasized that this problem arises even when only page locking is used, as is the case with DB2 [15].

Given these properties of the selective redo WAL-based method under discussion, we would lose track of the state of a page with respect to a losing (in-progress or in-rollback) transaction in the situation where the page modified first by the losing transaction (say, update with LSN 20 by T2) was subsequently modified by a nonloser transaction's update (say, update with LSN 30 by T1) which had to be redone. The latter would have pushed the LSN of the page beyond the value established by the loser. So, when the time comes to undo the loser, we would not know if its update needs to be undone or not. Figures 15 and 16 illustrate this problem with selective redo and fine-granularity locking. In the latter scenario, not redoing the update with LSN 20 since it belongs to a loser transaction, but redoing the update with LSN 30 since it belongs to a nonloser transaction, causes the undo pass to perform the undo of the former update even though it is not present in the page. This is because the undo logic relies on the page_LSN value to determine whether or not an update should be undone (undo if page_LSN is greater than or equal to log record's LSN). By not repeating history, the page_LSN is no longer a true indicator of the current state of the page.

Undoing an action even when its effect is not present in a page will be harmless only under certain conditions; for example, with physical/byte-oriented locking and logging, as they are implemented in IMS [76], VAX DBMS and VAX Rdb/VMS [81], and other systems [6], there is no automatic reuse of freed space, and unique keys for all records. With operation logging, data inconsistencies will be caused by undoing an original operation whose effect is not present in the page.

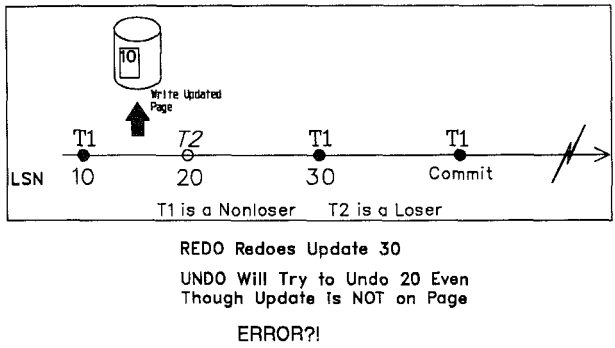


Fig. 16. Selective redo with WAL—problem scenario.

Reversing the order of the selective redo and the undo passes will not solve the problem either. This *incorrect* approach is suggested in [3]. If the undo pass were to precede the redo pass, then we might lose track of which actions need to be redone. In Figure 15, the undo of 20 would make the page LSN become greater than 30, because of the writing of a CLR and the assignment of that CLR's LSN to the page. Since, during the redo pass, a log record's update is redone only if the page_LSN is less than the log record's LSN, we would not redo 30 even though that update is not present on the page. Not redoing that update would violate the durability and atomicity properties of transactions.

The use of the shadow page technique by System R makes it unnecessary to have the concept of page_LSN in that system to determine what needs to be undone and what needs to be redone. With the shadow page technique, during a checkpoint, an action consistent version of the database, called the *shadow version*, is saved on nonvolatile storage. Updates between two checkpoints create a new version of the updated page, thus constituting the *current version* of the database (see Figure 1). During restart, recovery is performed from the *shadow* version, and shadowing is done even during restart recovery. As a result, there is no ambiguity about which updates are in the database and which are not. All updates logged after the last checkpoint are not in the database, and all updates logged before the checkpoint are in the database.[7] This is one reason the System R recovery method functions correctly even with selective redo. The other reason is that index and space management changes are not logged, but are redone or undone logically.[8]

[7] This simple view, as it is depicted in Figure 17, is not completely accurate—see Section 10.2.
[8] In fact, if index changes had been logged, then selective redo would not have worked. The problem would have come from structure modifications (like page split) which were performed after the last checkpoint by *loser* transactions which were taken advantage of later by transactions which ultimately committed. Even if logical undo were performed (if necessary), if redo was page oriented, selective redo would have caused problems. To make it work, the structure modifications could have been performed using separate transactions. Of course, this would have been very expensive. For an alternate, efficient solution, see [62].

As was described before, ARIES does not perform selective redo, but *repeats history*. Apart from allowing us to support fine-granularity locking, repeating history has another beneficial side effect. It gives us the ability to commit some actions of a transaction irrespective of whether the transaction ultimately commits or not, as was described in Section 9.

10.2 Rollback State

The goal of this subsection is to discuss the difficulties introduced by rollbacks in tracking their progress and how writing CLRs that describe updates performed during rollbacks solves some of the problems. While the concept of writing CLRs has been implemented in many systems and has been around for a long time, there has not really been, in the literature, a significant discussion of CLRs, problems relating to them and the advantages of writing them. Their utility and the fundamental role that they play in recovery have not been well recognized by the research community. In fact, whether undone actions could be undone and what additional problems these would present were left as open questions in [56]. In this section and elsewhere in this paper, in the appropriate contexts, we try to note all the known advantages of writing CLRs. We summarize these advantages in Section 13.

A transaction may totally or partially roll back its actions for any number of reasons. For example, a unique key violation will cause only the rollback of the update statement causing the violation and not of the entire transaction. Figure 3 illustrates a partial roll back. Supporting partial rollback [1, 31], at least internally, if not also at the application level, is a very important requirement for present-day transaction systems. Since a transaction may be rolling back when a failure occurs and since some of the effects of the updates performed during the rollback might have been written to nonvolatile storage, we need a way to keep track of the state of progress of transaction rollback. It is relatively easy to do this in System R. The only time we care about the transaction state in System R is at the time a checkpoint is taken. So, the checkpoint record in System R keeps track of the next record to be undone for each of the active transactions, some of which may already be rolling back. The rollback state of a transaction at the time of a system failure is unimportant since the database changes performed after the last checkpoint are not *visible* in the database during restart. That is, restart recovery starts from the state of the database as of the last checkpoint before the system failure—this is the shadow version of the database at the time of system failure. Despite this, since CLRs are never written, System R needs to do some special processing to handle those committed or in-doubt transactions which initiated and completed partial rollbacks after the last checkpoint. The special handling is to avoid the need for multiple passes over the log during the redo pass. The designers wanted to avoid redoing some actions only to have to undo them a little later with a backward scan, when the information about a partial rollback having occurred is encountered.

Figure 18 depicts an example of a restart recovery scenario for System R. All log records are written by the same transaction, say T1. In the checkpoint

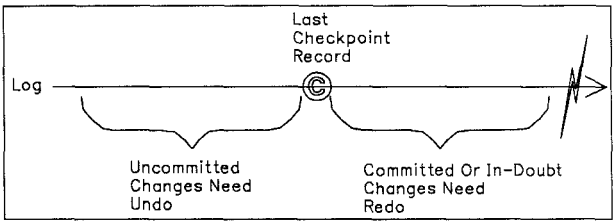


Fig. 17. Simple view of recovery processing in System R

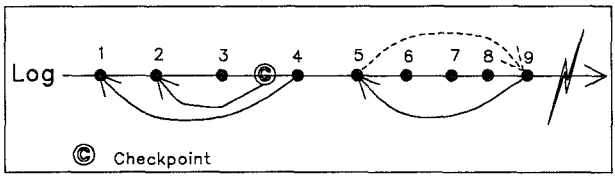


Fig. 18. Partial rollback handling in System R.

record, the information for T1 points to log record 2 since by the time the checkpoint was taken log record 3 had already been undone because of a partial rollback. System R not only does not write CLRs, but it also does not write a separate log record to say that a partial rollback took place. Such information must be inferred from the breakage in the chaining of the log records of a transaction. Ordinarily, a log record written by a transaction points to the record that was most recently written by that transaction via the PrevLSN pointer. But the first forward processing log record written after the completion of a partial rollback does not follow this protocol. When we examine, as part of the analysis pass, log record 4 and notice that its Prev_LSN pointer is pointing to 1, instead of the immediately preceding log record 3, we conclude that the partial rollback that started with the undo of 3 ended with the undo of 2. Since, during restart, the database state from which recovery needs to be performed is the state of the database as of the last checkpoint, log record 2 definitely needs to be undone. Whether 1 needs to be undone or not will depend on whether T1 is a losing transaction or not.

During the analysis pass it is determined that log record 9 points to log record 5 and hence it is concluded that a partial rollback had caused the undo of log records 6, 7, and 8. To ensure that the rolled back records are not redone during the redo pass, the log is patched by putting a forward pointer during the analysis pass in log record 5 to make it point to log record 9.

If log record 9 is a commit record then, during the undo pass, log record 2 will be undone and during the redo pass log records 4 and 5 will be redone. Here, the same transaction is involved both in the undo pass and in the redo pass. To see why the undo pass has to precede the redo pass in System R,[9]

[9] In the other systems, because of the fact that CLRs are written and that, sometimes, page LSNs are compared with log record's LSNs to determine whether redo needs to be performed or not, the redo pass *precedes* the undo pass—see the Section "10.1. Selective Redo" and Figure 6.

consider the following scenario: Since a transaction that deleted a record is allowed to reuse that record's ID for a record inserted later by the same transaction, in the above case, a record might have been deleted because of the partial rollback, which had to be dealt with in the undo pass, and that record's ID might have been reused in the portion of the transaction that is dealt with in the redo pass. To repeat history with respect to the original sequence of actions be fore the failure, the undo must be performed before the redo is performed.

If 9 is neither a commit record nor a prepare record, then the transaction will be determined to be a loser and during the undo pass log records 2 and 1 will be undone. In the redo pass, none of the records will be redone.

Since CLRs are not written in System R and hence the exact way in which one transaction's undo operations were interspersed with other transactions' forward processing or undo actions is not known, the processing, for a given page as well as across different pages during restart may be quite different from what happened during normal processing (i.e., *repeating history* is impossible to guarantee). Not logging index changes in System R also further contributes to this (see footnote 8). These could potentially cause some space management problems such as a split that did not occur during normal processing being required during the restart redo or undo processing (see also Section 5.4). Not writing CLRs also prevents logging of redo information from being done physically (i.e., the operation performed on an object has to be logged—not the after-image created by the operation). Let us consider an example: A piece of data has value 0 after the last checkpoint. Then, transaction T1 adds 1, T2 adds 2, T1 rolls back, and T2 commits. If T1 and T2 had logged the after-image for redo and the operation for undo, then there will be a data integrity problem because after recovery the data will have the value 3 instead of 2. In this case, in System R, undo for T1 is being accomplished by not redoing its update. Of course, System R did not support the fancy lock mode which would be needed to support 2 concurrent updates by different transactions to the same object. Allowing the logging of redo information physically will let redo recovery be performed very efficiently using *dumb* logic. This does not necessarily mean byte-oriented logging; that will depend on whether or not flexible storage management is used (see Section 10.3). Allowing the logging of undo information logically will permit high concurrency to be supported (see [59, 62] for examples). ARIES supports these.

WAL-based systems handle this problem by logging actions performed during rollbacks using CLRs. So, as far as recovery is concerned, the state of the data is always "marching" forward, even if some original actions are being rolled back. Contrast this with the approach, suggested in [52], in which the state of the data, as denoted by the LSN, is "pushed" back during rollbacks. That method works only with page level (or coarser granularity) locking. The immediate consequence of writing CLRs is that, if a transaction were to be rolled back, then some of its original actions are undone more than once and, worse still, the compensating actions are also undone, possibly more than once. This is illustrated in Figure 4, in which a transaction had started rolling back even before the failure of the system. Then, during

recovery, the previously written CLRs are undone and already undone non-CLRs are undone again. ARIES avoids such a situation, while still retaining the idea of writing CLRs. Not undoing CLRs has benefits relating to deadlock management and early release of locks on undone objects also (see item 22, Section 12, and Section 6.4). Additional benefits of CLRs are discussed in the next section and in [69]. Some were already discussed in the Section 8.

Unfortunately, recovery methods like the one suggested in [92] do not support partial rollbacks. We feel that this is an important drawback of such methods.

10.3 Space Management

The goal of this subsection is to point out the problems involved in space management when finer than page level granularity of locking and varying length records are to be supported efficiently.

A problem to be dealt with in doing record locking with flexible storage management is to make sure that the space released by a transaction during record deletion or update on a data page is not consumed by another transaction until the space-releasing transaction is committed. This problem is discussed briefly in [76]. We do not deal with solutions to this space reservation problem here. The interested reader is referred to [50]. For index updates, in the interest of increasing concurrency, we do not want to prevent the space released by one transaction from being consumed by another before the commit of the first transaction. The way undo is dealt with under such circumstances using a logical undo approach is described in [62].

Since flexible storage management was a goal, it was not desirable to do physical (i.e., byte-oriented) locking and logging of data within a page, as some systems do (see [6, 76, 81]). That is, we did not want to use the address of the first byte of a record as the lock name for the record. We did not want to identify the specific bytes that were changed on the page. The logging and locking have to be logical within a page. The record's lock name looks something like (page#, slot#) where the slot# identifies a location on the page which then points to the actual location of the record. The log record describes how the contents of the data record got changed. The consequence is that garbage collection that collects unused space on a page does not have to lock or log the records that are moved around within the page. This gives us the flexibility of being able to move records around within a page to store and modify variable length records efficiently. In systems like IMS, utilities have to be run quite frequently to deal with storage fragmentation. These reduce the availability of data to users.

Figure 19 shows a scenario in which not keeping track of the actual page state (by, e.g., storing the LSN in the nonvolatile storage version of the page) and attempting to perform redo from an earlier point in the log leads to problems when flexible storage management is used. Assuming that all updates in Figure 19 involve the same page and the same transaction, an insert requiring 200 bytes is attempted on a page which has only 100 bytes of free space left in it. This shows the need for exact tracking of page state

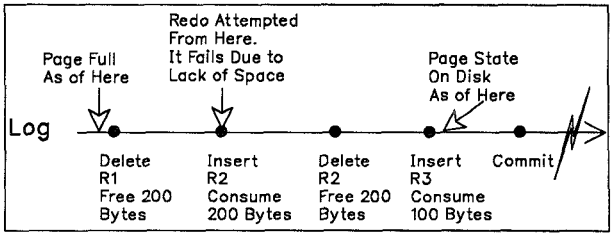


Fig. 19. Wrong redo point-causing problem with space for insert.

using an LSN to avoid attempting to redo operations which are already applied to the page.

Typically, each file containing records of one or more relations has a few pages called free space inventory pages (FSIPs). They are called space map pages (SMPs) in DB2. Each FSIP describes the space information relating to many data or index pages. During a record insert operation, possibly based on information obtained from a clustering index about the location of other records with the same key (or closely related keys) as that of the new record, one or more FSIPs are consulted to identify a data page with enough free space in it for inserting the new record. The FSIP keeps only approximate information (e.g., information such as that at least 25% of the page is full, at least 50% is full, etc.) to make sure that not every space-releasing or -consuming operation to a data page requires an update to the space information in the corresponding FSIP. To avoid special handling of the recovery of the FSIPs during redo and undo, and also to provide recovery independence, updates to the FSIPs must also be logged.

Transaction T1 might cause the space on the page to change from 23% full to 27% full, thereby requiring an update to the FSIP to change it from 0% full to 25% full. Later, T2 might cause the space to go to 35% full, which does not require an update to the FSIP. Now, if T1 were to roll back, then the space would change to 31% full and this should not cause an update to the FSIP. If T1 had written its FSIP change log record as a redo/undo record, then T1's rollback would cause the FSIP entry to say 0% full, which would be wrong, given the *current* state of the data page. This scenario points to the need for logging the changes to the FSIP as *redo-only* changes and for the need to do logical undos with respect to the free space inventory updates. That is, while undoing a data page update, the system has to determine whether that operation causes the free space information to change and if it does cause a change, then update the FSIP and write a CLR which describes the change to the FSIP. We can easily construct an example in which a transaction does not perform an update to the FSIP during forward processing, but needs to perform an update to the FSIP during rollback. We can also construct an example in which the update performed during forward processing is not the exact inverse of the update during the rollback.

10.4 Multiple LSNs

Noticing the problems caused by having one LSN per page when trying to support record locking, it may be tempting to suggest that we track each object's state precisely by assigning a separate LSN to each object. Next we explain why it is not a good idea.

DB2 already supports a granularity of locking that is less than a page. This happens in the case of indexes where the user has the option of requiring DB2 to physically divide up each *leaf* page of the index into 2 to 16 minipages and do locking at the granularity of a minipage [10, 12]. The way DB2 does recovery properly on such pages, despite not redoing actions of loser transactions during the redo pass, is as follows. DB2 tracks each minipage's state separately by associating an LSN with each minipage, besides having an LSN for the leaf page as a whole. Whenever a minipage is updated, the corresponding log record's LSN is stored in the minipage LSN field. The page LSN is set equal to the *maximum* of the minipage LSNs. During undo, it is the minipage LSN and not the page LSN that is compared to the log record's LSN to determine if that log record's update needs to be actually undone on the minipage. This technique, besides incurring too much space overhead for storing the LSNs, tends to fragment (and therefore waste) space available for storing keys. Further, it does not carry over conveniently to the case of record and key locking, especially when varying length objects have to be supported efficiently. Maintaining LSNs for deleted objects is cumbersome at best. We desired to have a *single* state variable (LSN) for each page, even when minipage locking is being done, to make recovery, especially media recovery, very efficient. The simple technique of repeating history during restart recovery before performing the rollback of loser transactions turns out to be sufficient, as we have seen in ARIES. Since DB2 *physically* divides up a page into a fixed number of minipages, no special technique is needed to handle the space reservation problem. Methods like the one proposed in [61] for fine-granularity locking do not support varying length objects (*atoms* in the terminology of that paper).

11. OTHER WAL-BASED METHODS

In the following, we summarize the properties of some other significant recovery methods which also use the WAL protocol. Recovery methods based on the shadow page technique (like that of System R) are not considered here because of their well-known disadvantages, e.g., very costly checkpoints, extra nonvolatile storage space overhead for the shadow copies of data, disturbing the physical clustering of data, and extra I/Os involving page map blocks (see the previous sections of this paper and [31] for additional discussions). First, we briefly introduce the different systems and recovery methods which we will be examining in this section. Next, we compare the different methods along various dimensions. We have been informed that the DB-cache recovery method of [25] has been implemented with significant modifications by Siemens. But, because of lack of information about the implementation, we are unable to include it here.

IBM's IMS/VS [41, 42, 43, 48, 53, 76, 80, 94], which is a hierarchical database system, consists of two parts: IMS Full Function (FF), which is relatively flexible, and IMS Fast Path [28, 42, 93], which is more efficient but has many restrictions (e.g., no support for secondary indexes). A single IMS transaction can access both FF and Fast Path (FP) data. The recovery and buffering methods used by the two parts have many differences. In FF, depending on the database types and the operations, the granularities of the locked objects vary. FP supports two kinds of databases: main storage databases (MSDBs) and data entry databases (DEDBs). MSDBs support only fixed length records, but FP provides the mechanisms (i.e., *field calls*) to make the lock hold times be the minimum possible for MSDB records. Only page locking is supported for DEDBs. But, DEDBs have many high-availability and parallelism features and large database support. IMS, with XRF, provides *hot-standby* support [43]. IMS, via global locking, also supports data sharing across two different systems, each with its own buffer pools [80, 94].

DB2 is IBM's relational database system for the MVS operating system. Limited distributed data access functions are available in DB2. The DB2 recovery algorithm has been presented in [1, 13, 14, 15, 19]. It supports different locking granularities (tablespace, table and page for data, and minipage and page for indexes) and consistency levels (*cursor stability*, *repeatable read*) [10, 11, 12]. DB2 allows logging to be turned off temporarily for tables and indexes only during utility operations like loading and reorganizing data. A single transaction can access both DB2 and IMS data with atomicity. The Encompass recovery algorithm [4, 37] with some changes has been incorporated in Tandem's NonStop SQL [95]. With NonStop, Tandem provides hot-standby support for its products. Both Encompass and NonStop SQL support distributed data access. They allow multisite updates within a single transaction using the Presumed Abort two-phase commit protocol of [63, 64]. NonStop SQL supports different locking granularities (file, key prefix and record) and consistency levels (cursor stability, repeatable read, and unlocked or dirty read). Logging can be turned off temporarily or permanently even for nonutility operations on files.

Schwarz [88] presents two different recovery methods based on value logging (a la IMS) and operation logging. The two methods have several differences, as will be outlined below. The value logging method (VLM), which is much less complex than the operation logging method (OLM), has been implemented in CMU's Camelot [23, 90].

Buffer management. Encompass, NonStop SQL, OLM, VLM and DB2 have adopted the steal and no-force policies. During normal processing, VLM and OLM write a *fetch* record whenever a page is read from nonvolatile storage and an *end-write* record every time a dirty page is successfully written back to nonvolatile storage. These are written during restart processing also in OLM alone. These records help in identifying the super set of dirty pages that might have been in the buffer pool at the time of system failure. DB2 has a sophisticated buffer manager [10, 96], and writes a log

record whenever a tablespace or an indexspace is opened, and another record whenever such a space is closed. The close operation is performed only after all the dirty pages of the space have been written back to nonvolatile storage. DB2's analysis pass uses these log records to bring the dirty objects information up to date as of the failure.

For MSDBs, IMS FP does deferred updating. This means that a transaction does not see its own MSDB updates. For DEDBs, a no-steal policy is used. FP writes, at commit time, all the log records for a given transaction in a single call to the log manager. After placing the log records in the *log buffers* (not on stable storage), the MSDB updates are applied and the MSDB record locks are released. The MSDB locks are released even before the commit log record is placed on stable storage. This is how FP minimizes the amount of time locks are held on the MSDB records. The DEDB locks are transferred to system processes. The log manager is given time to let it force the log records to stable storage ultimately (i.e., group commit logic is used—see [28]). After the logging is completed (i.e., *after* the transaction has been committed), all the pages of the DEDBs that were modified by the transaction are forced to nonvolatile storage using system processes which, on completion of the I/Os, release the DEDB locks. This does not result in any uncommitted updates being forced to nonvolatile storage since page locking with a no-steal policy is used for DEDBs. The use of separate processes for writing the DEDB pages to nonvolatile storage is intended to let the user process go ahead with the next transaction's processing as soon as possible and also to gain parallelism for the I/Os. IMS FF follows the steal and force policies. *Before* committing a transaction, IMS FF forces to nonvolatile storage all the pages that were modified by that transaction. Since finer than page locking is supported by FF, this may result in some uncommitted data being placed on nonvolatile storage. Of course, all the recovery algorithms considered in this section force the log during commit processing.

Normal checkpointing. Normal checkpoints are the ones that are taken when the system is not in the restart recovery mode. OLM and VLM quiesce all activity in the system and take, similar to System R, an operation consistent (not necessarily transaction consistent) checkpoint. The contents of the checkpoint record are similar to those of ARIES. DB2, IMS, NonStop SQL, and Encompass do take (fuzzy) checkpoints even when update and logging activities are going on concurrently. DB2's checkpoint actions are similar to what we described for ARIES. The major difference is that, instead of writing the dirty_pages table, it writes the dirty objects (tablespaces, indexspaces, etc.) list with a RecLSN for each object [96]. For MSDBs alone, IMS writes their complete contents alternately on one of two files on nonvolatile storage during a checkpoint. Since deferred updating is performed for MSDBs, no uncommitted changes will be present in their checkpointed version. Also, it is ensured that no partial committed changes of a transaction are present. Care is needed since the updates are applied after the commit record is written. For DEDBs, any *committed* updated pages which have not yet been written to nonvolatile storage are included in the check-

point records. These together avoid the need for examining, during restart recovery, any log records written before the checkpoint for FP data recovery. Encompass and NonStop SQL might force some dirty pages to nonvolatile storage during a checkpoint. They enforce the policy that requires that a page once dirtied must be written to nonvolatile storage before the completion of the second checkpoint following the dirtying of the page. Because of this policy, the completion of a checkpoint may be delayed waiting for the completion of the writing of the old dirty pages.

Partial rollbacks. Encompass, NonStop SQL, OLM and VLM do not support partial transaction rollback. From Version 2 Release 1, IMS supports partial rollbacks. In fact, the savepoint concept is exposed at the application program level. This support is available only to those applications that do not access FP data. The reason FP data is excluded is because FP does not write undo data in its log records and because deferred updating is performed for MSDBs. DB2 supports partial rollbacks for internal use by the system to provide statement-level atomicity [1].

Compensation log records. Encompass, NonStop SQL, DB2, VLM, OLM and IMS FF write CLRs during normal rollbacks. During a normal rollback, IMS FP does not write CLRs since it would not have written any log records for changes to such data until the decision to rollback is made. This is because FP is always the coordinator in two-phase commit and hence it never needs to get into the prepared state. Since deferred updating is performed for MSDBs, the updates kept in pending (to-do) lists are discarded at rollback time. Since a no-steal policy is followed and page locking is done for DEDBs, the modified pages of DEDBs are simply purged from the buffer pool at rollback time. Encompass, NonStop SQL, DB2 and IMS (FF and FP) write CLRs during restart rollbacks also. During restart recovery, IMS FP might find some log records written by (at the most) one in-progress transaction. This transaction must have been in commit processing—i.e., about to commit, with some of its log records already having been written to nonvolatile storage—when the system went down. Even though, because of the no-steal policy, none of the corresponding FP updates would have been written to nonvolatile storage and hence there would be nothing to be undone, IMS FP writes CLRs for such records to simplify media recovery [93]. Since the FP log records contain only redo information, just to write these CLRs, for which the undo information is needed, the corresponding unmodified data on nonvolatile storage is accessed during restart recovery. This should illustrate to the reader that even with a no-steal policy and without supporting partial rollbacks, there are still some problems to be dealt with at restart for FP. Too often, people assume that no-steal eliminates many problems. Actually, it has many shortcomings.

VLM does not write CLRs during restart rollbacks. As a result, a bounded amount of logging will occur for a rolled back transaction, even in the face of repeated failures during restart. In fact, CLRs are written only for normal rollbacks. Of course, this has some negative implications with respect to media recovery. OLM writes CLRs for undos *and redos* performed during

restart (called *undomodify* and *redomodify* records, respectively). This is done to deal with failures during restart. OLM might write multiple undomodify and redomodify records for a given update record if failures interrupt restart processing. No CLRs are generated for CLRs themselves. During restart recovery, Encompass and DB2 undo changes of CLRs, thus causing the writing of CLRs for CLRs and the writing of multiple, identical CLRs for a given record written during forward or restart processing. In the worst case, the number of log records written during repeated restart failures grows exponentially. Figure 5 shows how ARIES avoids this problem. IMS ignores CLRs during the undo pass and hence does not write CLRs for them. The net result is that, because of multiple failures, like the others, IMS might wind up writing multiple times the same CLR for a given record written during forward processing. In the worst case, the number of log records written by IMS and OLM grows linearly. Because of its force policy, IMS will need to redo the CLR's updates only during media recovery.

Log record contents. IMS FP writes only redo information (i.e., after-image of records) because of its no-steal policy. As mentioned before, IMS does value (or state) logging and physical (i.e., byte-range) locking (see [76]). IMS FF logs both the undo information and the redo information. Since IMS does not undo CLRs' updates, CLRs need to have only the redo information. For providing the XRF hot-standby support, IMS includes enough information in its log records for the backup system to track the lock names of updated objects. IMS FP also logs the address of the buffer occupied by a modified page. This information is used during a backup's takeover or restart recovery to reduce the amount of redo work of DEDBs' updates. Encompass and VLM also log complete undo and redo information of updated records. DB2 and NonStop SQL log only the before- and after-images of the updated fields. OLM logs the description of the update operation. The CLRs of Encompass and DB2 need to contain both the redo and the undo information since their CLRs might be undone. OLM periodically logs an operation consistent *snapshot* of each object. OLM's undomodify and redomodify records contain no redo or undo information but only the LSNs of the corresponding modify records. But OLM's modify, redomodify and undomodify records also contain a page map which specifies the set of pages where parts of the modified object reside.

Page overhead. Encompass and NonStop SQL use one LSN on each page to keep track of the state of the page. VLM uses no LSNs, but OLM uses one LSN. DB2 uses one LSN and IMS FF no LSN. Not having the LSN in IMS FF and VLM to know the exact state of a page does not cause any problems because of IMS' and VLM's value logging and physical locking attributes. It is acceptable to redo an already present update or undo an absent update. IMS FP uses a field in the pages of DEDBs as a version number to correctly handle redos after all the data sharing systems have failed [67]. When DB2 divides an index leaf page into minipages then it uses one LSN for each minipage, besides one LSN for the page as a whole.

Log passes during restart recovery. Encompass and NonStop SQL make two passes (redo and then undo), and DB2 makes three passes (analysis, redo, and then undo—see Figure 6). Encompass and NonStop SQL start their redo passes from the beginning of the penultimate successful checkpoint. This is sufficient because of the buffer management policy of writing to disk a dirty page within two checkpoints after the page became dirty. They also seem to repeat history before performing the undo pass. They do not seem to repeat history if a backup system takes over when a primary system fails [4]. In the case of a takeover by a hot-standby, locks are first reacquired for the losers' updates and then the rollbacks of the losers are performed in parallel with the processing of new transactions. Each loser transaction is rolled back using a separate process to gain parallelism. DB2 starts its redo scan from that point, which is determined using information recorded in the last successful checkpoint, as modified by the analysis pass. As mentioned before, DB2 does selective redo (see Section 10.1).

VLM makes one backward pass and OLM makes three passes (analysis, undo, and then redo). Many lists are maintained during OLM's and VLM's passes. The undomodify and redomodify log records of OLM are used only to modify these lists, unlike in the case of the CLRs written in the other systems. In VLM, the one backward pass is used to undo uncommitted changes on nonvolatile storage and also to redo missing committed changes. No log records are written during these operations. In OLM, during the undo pass, for each object to be recovered, if an operation consistent version of the object does not exist on nonvolatile storage, then it restores a snapshot of the object from the snapshot log record so that, starting from a consistent version of the object, (1) in the remainder of the undo pass any to-be-undone updates that precede the snapshot log record can be undone logically, and (2) in the redo pass any committed or in-doubt updates (modify records only) that follow the snapshot record can be redone logically. This is similar to the shadowing performed in [16, 78] using a separate log—the difference is that the database-wide checkpointing is replaced by object-level checkpointing and the use of a single log instead of two logs.

IMS first reloads MSDBs from the file that received their contents during the latest successful checkpoint before the failure. The dirty DEDB buffers that were included in the checkpoint records are also reloaded into the same buffers as before. This means that, during the restart after a failure, the number of buffers cannot be altered. Then, it makes just one forward pass over the log (see Figure 6). During that pass, it accumulates log records in memory on a per-transaction basis and redoes, if necessary, completed transactions' FP updates. Multiple processes are used in parallel to redo the DEDB updates. As far as FP is concerned, only the updates starting from the last checkpoint before the failure are of interest. At the end of that one pass, in-progress transactions' FF updates are undone (using the log records in memory), in parallel, using one process per transaction. If the space allocated in memory for a transaction's log records is not enough, then a backward scan of the log will be performed to fetch the needed records during that transaction's rollback. In the XRF context, when a hot-standby IMS

takes over, the handling of the loser transactions is similar to the way Tandem does it. That is, rollbacks are performed in parallel with new transaction processing.

Page forces during restart. OLM, VLM and DB2 force all dirty pages at the end of restart. Information on Encompass and NonStop SQL is not available.

Restart checkpoints. IMS, DB2, OLM and VLM take a checkpoint only at the end of restart recovery. Information on Encompass and NonStop SQL is not available.

Restrictions on data. Encompass and NonStop SQL require that every record have a unique key. This unique key is used to guarantee that if an attempt is made to undo a logged action which was never applied to the nonvolatile storage version of the data, then the latter is realized and the undo fails. In other words, idempotence of operations is achieved using the unique key. IMS in effect does byte-range locking and logging and hence does not allow records to be moved around freely within a page. This results in the fragmentation and the less efficient usage of free space. IMS imposes some additional constraints with respect to FP data. VLM requires that an object's representation be divided into fixed length (less than one page sized), unrelocatable quanta. The consequences of these restrictions are similar to those for IMS.

[2, 26, 56] do not discuss recovery from system failures, while the theory of [33] does not include semantically rich modes of locking (i.e., operation logging). In other sections of this paper, we have pointed out the problems with some of the other approaches that have been proposed in the literature.

12. ATTRIBUTES OF ARIES

ARIES makes few assumptions about the data or its model and has several advantages over other recovery methods. While ARIES is simple, it possesses several interesting and useful properties. Each of most of these properties has been demonstrated in one or more existing or proposed systems, as summarized in the last section. However, we know of no single system, proposed or real, which has all of these properties. Some of these properties of ARIES are:

(1) *Support for finer than page-level concurrency control and multiple granularities of locking.* ARIES supports page-level and record-level locking in a uniform fashion. Recovery is not affected by what the granularity of locking is. Depending on the expected contention for the data, the appropriate level of locking can be chosen. It also allows multiple granularities of locking (e.g., record, table, and tablespace-level) for the same object (e.g., tablespace). Concurrency control schemes other than locking (e.g., the schemes of [2]) can also be used.

(2) *Flexible buffer management during restart and normal processing.* As long as the write-ahead logging protocol is followed, the buffer manager is

free to use any page replacement policy. In particular, dirty pages of incomplete transactions can be written to nonvolatile storage before those transactions commit (*steal* policy). Also, it is not required that all pages dirtied by a transaction be written back to nonvolatile storage before the transaction is allowed to commit (i.e., *no-force* policy). These properties lead to reduced demands for buffer storage and fewer I/Os involving frequently updated (*hot-spot*) pages. ARIES does not preclude the possibilities of using deferred-updating and force-at-commit policies and benefiting from them. ARIES is quite flexible in these respects.

(3) *Minimal space overhead—only one LSN per page.* The permanent (excluding log) space overhead of this scheme is limited to the storage required on each page to store the LSN of the last logged action performed on the page. The LSN of a page is a monotonically increasing value.

(4) *No constraints on data to guarantee idempotence of redo or undo of logged actions.* There are no restrictions on the data with respect to unique keys, etc. Records can be of variable length. Data can be moved around within a page for garbage collection. Idempotence of operations is ensured since the LSN on each page is used to determine whether an operation should be redone or not.

(5) *Actions taken during the undo of an update need not necessarily be the exact inverses of the actions taken during the original update.* Since CLRs are being written during undos, any differences between the inverses of the original actions and what actually had to be done during undo can be recorded in the former. An example of when the inverse might not be correct is the one that relates to the free space information (like at least 10% free, 20% free) about data pages that are maintained in space map pages. Because of finer than page-level granularity locking, while no free space information change takes place during the initial update of a page by a transaction, a free space information change might occur during the undo (from 20% free to 10% free) of that original change because of intervening update activities of other transactions (see Section 10.3).

Other benefits of this attribute in the context of hash-based storage methods and index management can be found in [59, 62].

(6) *Support for operation logging and novel lock modes.* The changes made to a page can be logged in a logical fashion. The undo information and the redo information for the entire object need not be logged. It suffices if the changed fields alone are logged. Since history is repeated, for increment or decrement kinds of operations before- and after-images of the field are not needed. Information about the type of operation and the decrement or increment amount is enough. Garbage collection actions and changes to some fields (e.g., amount of free space) of that page need not be logged. Novel lock modes based on commutativity and other properties of operations can be supported [2, 26, 88].

(7) *Even redo-only and undo-only records are accommodated.* While it may be efficient (single call to the log component) sometimes to include the undo and redo information about an update in the same log record, at other

times it may be efficient (from the original data, the undo record can be constructed and, after the update is performed *in-place* in the data record, from the updated data, the redo record can be constructed) and/or necessary (because of log record size restrictions) to log the information in two different records. ARIES can handle both situations. Under these conditions, the undo record must be logged before the redo record.

(8) *Support for partial and total transaction rollback.* Besides allowing transactions to be rolled back totally, ARIES allows the establishment of savepoints and the partial rollback of transactions to such savepoints. Without the support for partial rollbacks, even logically recoverable errors (e.g., unique key violation, out-of-date cached catalog information in a distributed database system) will require total rollbacks and result in wasted work.

(9) *Support for objects spanning multiple pages.* Objects can span multiple pages (e.g., an IMS "record" which consists of multiple segments may be scattered over many pages). When an object is modified, if log records are written for every page affected by that update, ARIES works fine. ARIES itself does not treat multipage objects in any special way.

(10) *Allows files to be acquired or returned, any time, from or to the operating system.* ARIES provides the flexibility of being able to return files dynamically and permanently to the operating system (see [19] for the detailed description of a technique to accomplish this). Such an action is considered to be one that cannot be undone. It does not prevent the same file from being reallocated to the database system. Mappings between objects (tablespaces, etc.) and files are not required to be defined statically as in System R.

(11) *Some actions of a transaction may be committed even if the transaction as a whole is rolled back.* This refers to the technique of using the concept of a dummy CLR to implement nested top actions. File extension has been given as an example situation which could benefit from this. Other applications of this technique, in the context of hash-based storage methods and index management, can be found in [59, 62].

(12) *Efficient checkpoints (including during restart recovery).* By supporting fuzzy checkpointing, ARIES makes taking a checkpoint an efficient operation. Checkpoints can be taken even when update activities and logging are going on concurrently. Permitting checkpoints even during restart processing will help reduce the impact of failures during restart recovery. The dirty_pages information written during checkpointing helps reduce the number of pages which are read from nonvolatile storage during the redo pass.

(13) *Simultaneous processing of multiple transactions in forward processing and/or in rollback accessing same page.* Since many transactions could simultaneously be going forward or rolling back on a given page, the level of concurrent access supported could be quite high. Except for the short duration latching which has to be performed any time a page is being

physically modified or examined, be it during forward processing or during rollback, rolling back transactions do not affect one another in any unusual fashion.

(14) *No locking or deadlocks during transaction rollback.* Since no locking is required during transaction rollback, no deadlocks will involve transactions that are rolling back. Avoiding locking during rollbacks simplifies not only the rollback logic, but also the deadlock detector logic. The deadlock detector need not worry about making the mistake of choosing a rolling back transaction as a victim in the event of a deadlock (cf. System R and R* [31, 49, 64]).

(15) *Bounded logging during restart in spite of repeated failures or of nested rollbacks.* Even if repeated failures occur during restart, the number of CLRs written is unaffected. This is also true if partial rollbacks are nested. The number of log records written will be the same as that written at the time of transaction rollback during normal processing. The latter again is a fixed number and is, usually, equal to the number of undoable records written during the forward processing of the transaction. No log records are written during the redo pass of restart.

(16) *Permits exploitation of parallelism and selective/deferred processing for faster restart.* Restart can be made faster by not doing all the needed I/Os synchronously one at a time while processing the corresponding log record. ARIES permits the early identification of the pages needing recovery and the initiation of asynchronous parallel I/Os for the reading in of those pages. The pages can be processed concurrently as they are brought into memory during the redo pass. Undo parallelism requires complete handling of a given transaction by a single process. Some of the restart processing can be postponed to speed up restart or to accommodate offline devices. If desired, undo of loser transactions can be performed in parallel with new transaction processing.

(17) *Fuzzy image copying (archive dumping) for media recovery.* Media recovery and image copying of the data are supported very efficiently. To take advantage of device geometry, the actual act of copying can even be performed outside the transaction system (i.e., without going through the buffer pool). This can happen even while the latter is accessing and modifying the information being copied. During media recovery only one forward traversal of the log is made.

(18) *Continuation of loser transactions after a system restart.* Since ARIES repeats history and supports the savepoint concept, we could, in the undo pass, instead of totally rolling back the loser transactions, roll back each loser only to its latest savepoint. Locks must be acquired to protect the transaction's uncommitted, not undone updates. Later, we could resume the transaction by invoking its application at a special entry point and passing enough information about the savepoint from which execution is to be resumed.

(19) *Only one backward traversal of log during restart or media recovery.*

Both during media recovery and restart recovery one backward traversal of the log is sufficient. This is especially important if any portion of the log is likely to be stored in a slow medium like tape.

(20) *Need only redo information in compensation log records.* Since compensation records are never undone they need to contain only redo information. So, on the average, the amount of log space consumed during a transaction rollback will be half the space consumed during the forward processing of that transaction.

(21) *Support for distributed transactions.* ARIES accommodates distributed transactions. Whether a given site is a coordinator or a subordinate site does not affect ARIES.

(22) *Early release of locks during transaction rollback and deadlock resolution using partial rollbacks.* Because ARIES never undoes CLRs and because it never undoes a particular non-CLR more than once, during a (partial) rollback, when the transaction's very first update to a particular object is undone and a CLR is written for it, the system can release the lock on that object. This makes it possible to consider resolving deadlocks using partial rollbacks.

It should be noted that ARIES does not prevent the shadow page technique from being used for selected portions of the data to avoid logging of only undo information or both undo and redo information. This may be useful for dealing with long fields, as is the case in the OS/2 Extended Edition Database Manager. In such instances, for such data, the modified pages would have to be forced to nonvolatile storage before commit. Whether or not media recovery and partial rollbacks can be supported will depend on what is logged and for which updates shadowing is done.

13. SUMMARY

In this paper, we presented the ARIES recovery method and showed why some of the recovery paradigms of System R are inappropriate in the WAL context. We dealt with a variety of features that are very important in building and operating an *industrial-strength* transaction processing system. Several issues regarding operation logging, fine-granularity locking, space management, and flexible recovery were discussed. In brief, ARIES accomplishes the goals that we set out with by logging all updates on a per-page basis, using an LSN on every page for tracking page state, repeating history during restart recovery before undoing the loser transactions, and chaining the CLRs to the predecessors of the log records that they compensated. Use of ARIES is not restricted to the database area alone. It can also be used for implementing persistent object-oriented languages, recoverable file systems and transaction-based operating systems. In fact, it is being used in the QuickSilver distributed operating system [40] and in a system designed to aid the backing up of workstation data on a host [44].

In this section, we summarize as to which specific features of ARIES lead to which specific attributes that give us flexibility and efficiency.

Repeating history exactly, which in turn implies using LSNs and writing CLRs during undos, permits the following, irrespective of whether CLRs are chained using the UndoNxtLSN field or not:

(1) Record level locking to be supported and records to be moved around within a page to avoid storage fragmentation without the moved records having to be locked and without the movements having to be logged.
(2) Use only one state variable, a log sequence number, per page.
(3) Reuse of storage released by one transaction for the same transaction's later actions or for other transactions' actions once the former commits, thereby leading to the preservation of clustering of records and the efficient usage of storage.
(4) The inverse of an action origianlly performed during forward processing of a transaction to be different from the action(s) performed during the undo of that original action (e.g., class changes in the space map pages). That is, logical undo with recovery independence is made possible.
(5) Multiple transactions may undo on the same page concurrently with transactions going forward.
(6) Recovery of each page independently of other pages or of log records relating to transaction state, especially during media recovery.
(7) If necessary, the continuation of transactions which were in progress at the time of system failure.
(8) Selective or deferred restart, and undo of losers concurrently with new transaction processing to improve data availability.
(9) Partial rollback of transactions.
(10) Operation logging and logical logging of changes within a page. For example, decrement and increment operations may be logged, rather than the before- and after-images of modified data.

Chaining, using the UndoNxtLSN field, CLRs to log records written during forward processing permits the following, provided the protocol of repeating history is also followed:

(1) The avoidance of undoing CLRs' actions, thus avoiding writing CLRs for CLRs. This also makes it unnecessary to store undo information in CLRs.
(2) The avoidance of the undo of the same log record written during forward processing more than once.
(3) As a transaction is being rolled back, the ability to release the lock on an object when all the updates to that object had been undone. This may be important while rolling back a long transaction or while resolving a deadlock by partially rolling back the victim.
(4) Handling partial rollbacks without any special actions like patching the log, as in System R.
(5) Making permanent, if necessary via nested top actions, some of the

changes made by a transaction, irrespective of whether the transaction itself subsequently rolls back or commits.

Performing the analysis pass before repeating history permits the following:

(1) Checkpoints to be taken any time during the redo and undo passes of recovery.
(2) Files to be returned to the operating system dynamically, thereby allowing dynamic binding between database objects and files.
(3) Recovery of file-related information concurrently with the recovery of user data, without requiring special treatment for the former.
(4) Identifying pages possibly requiring redo, so that asynchronous parallel I/Os could be initiated for them even before the redo pass starts.
(5) Exploiting opportunities to avoid redos on some pages by eliminating those pages from the dirty_pages table on noticing, e.g., that some empty pages have been freed.
(6) Exploiting opportunities to avoid reading some pages during redo, e.g., by writing end_write records after dirty pages have been written to nonvolatile storage and by eliminating those pages from the dirty_pages table when the end_write records are encountered.
(7) Identifying the transactions in the in-doubt and in-progress states so that locks could be reacquired for them during the redo pass to support selective or deferred restart, the continuation of loser transactions after restart, and undo of loser transactions in parallel with new transaction processing.

13.1 Implementations and Extensions

ARIES forms the basis of the recovery algorithms used in the IBM Research prototype systems Starburst [87] and QuickSilver [40], in the University of Wisconsin's EXODUS and Gamma database machine [20], and in the IBM program products OS/2 Extended Edition Database Manager [7] and Workstation Data Save Facility/VM [44]. One feature of ARIES, namely *repeating history*, has been implemented in DB2 Version 2 Release 1 to use the concept of nested top action for supporting segmented tablespaces. A simulation study of the performance of ARIES is reported in [98]. The following conclusions from that study are worth noting: "Simulation results indicate the success of the ARIES recovery method in providing fast recovery from failures, caused by long intercheckpoint intervals, efficient use of page LSNs, log LSNs, and RecLSNs avoids redoing updates unnecessarily, and the actual recovery load is reduced skillfully. Besides, the overhead incurred by the concurrency control and recovery algorithms on transactions is very low, as indicated by the negligibly small difference between the mean transaction response time and the average duration of a transaction if it ran alone in a never failing system. This observation also emerges as evidence that the recovery method goes well with concurrency control through fine-granularity locking, an important virtue."

We have extended ARIES to make it work in the context of the nested transaction model (see [70, 85]). Based on ARIES, we have developed new methods, called ARIES/KVL, ARIES/IM and ARIES/LHS, to efficiently provide high concurrency and recovery for B^+-tree indexes [57, 62] and for hash-based storage structures [59]. We have also extended ARIES to restrict the amount of repeating of history that takes place for the loser transactions [69]. We have designed concurrency control and recovery algorithms, based on ARIES, for the N-way data sharing (i.e., shared disks) environment [65, 66, 67, 68]. *Commit_LSN*, a method which takes advantage of the page_LSN that exists in every page to reduce the locking, latching and predicate reevaluation overheads, and also to improve concurrency, has been presented in [54, 58, 60]. Although messages are an important part of transaction processing, we did not discuss message logging and recovery in this paper.

ACKNOWLEDGMENTS

We have benefited immensely from the work that was performed in the System R project and in the DB2 and IMS product groups. We have learned valuable lessons by looking at the experiences with those systems. Access to the source code and internal documents of those systems was very helpful. The Starburst project gave us the opportunity to begin from scratch and design some of the fundamental algorithms of a transaction system, taking into account experiences with the prior systems. We would like to acknowledge the contributions of the designers of the other systems. We would also like to thank our colleagues in the research and product groups that have adopted our research results. Our thanks also go to Klaus Kuespert, Brian Oki, Erhard Rahm, Andreas Reuter, Pat Selinger, Dennis Shasha, and Irv Traiger for their detailed comments on the paper.

REFERENCES

1. Baker, J., Crus, R., and Haderle, D. Method for assuring atomicity of multi-row update operations in a database system. U.S. Patent 4,498,145, IBM, Feb. 1985.
2. Badrinath, B. R., and Ramamritham, K. Semantics-based concurrency control: Beyond commutativity. In *Proceedings 3rd IEEE International Conference on Data Engineering* (Feb. 1987).
3. Bernstein, P., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass., 1987.
4. Borr, A. Robustness to crash in a distributed database: A non-shared-memory multiprocessor approach. In *Proceedings 10th International Conference on Very Large Data Bases* (Singapore, Aug. 1984).
5. Chamberlin, D., Gilbert, A., and Yost, R. A history of System R and SQL/Data System. In *Proceedings 7th International Conference on Very Large Data Bases* (Cannes, Sept. 1981).
6. Chang, A., and Mergen, M. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst., 6,* 1 (Feb. 1988), 28–50.
7. Chang, P. Y., and Myre, W. W. OS/2 EE database manager: Overview and technical highlights. *IBM Syst. J. 27,* 2 (1988).
8. Copeland, G., Khoshafian, S., Smith, M., and Valduriez, P. Buffering schemes for permanent data. In *Proceedings International Conference on Data Engineering* (Los Angeles, Feb. 1986).

9. CLARK, B. E., AND CORRIGAN, M. J. Application System/400 performance characteristics. *IBM Syst. J. 28*, 3 (1989).
10. CHENG, J., LOOSELY, C., SHIBAMIYA, A., AND WORTHINGTON, P. IBM Database 2 performance: Design, implementation, and tuning. *IBM Syst. J. 23*, 2 (1984).
11. CRUS, R , HADERLE, D., AND HERRON, H. Method for managing lock escalation in a multiprocessing, multiprogramming environment. U.S. Patent 4,716,52 8, IBM, Dec. 1987.
12. CRUS, R., MALKEMUS, T., AND PUTZOLU, G. R. Index mini-pages *IBM Tech. Disclosure Bull. 26*, 4 (April 1983), 5460-5463.
13. CRUS, R., PUTZOLU, F., AND MORTENSON, J. A Incremental data base log image copy *IBM Tech. Disclosure Bull. 25*, 7B (Dec. 1982), 3730-373 2.
14. CRUS, R., AND PUTZOLU, F. Data base allocation table. *IBM Tech. Disclosure Bull. 25*, 7B (Dec. 1982), 3722-2724.
15. CRUS, R. Data recovery in IBM Database 2. *IBM Syst. J. 23*, 2 (1984).
16. CURTIS, R. Informix-Turbo, In *Proceedings IEEE Compcon Spring '88* (Feb.-March 1988).
17. DASGUPTA, P., LEBLANC, R., JR., AND APPELBE, W. The Clouds distributed operating system. In *Proceedings 8th International Conference on Distributed Computing Systems* (San Jose, Calif., June 1988).
18. DATE, C. *A Guide to INGRES*. Addison-Wesley, Reading, Mass., 1987.
19. DEY, R., SHAN, M., AND TRAIGER, I. Method for dropping data sets. *IBM Tech. Disclosure Bull. 25*, 11A (April 1983), 5453-5455.
20. DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H.-I., AND RASMUSSEN, R. The Gamma database machine project. *IEEE Trans. Knowledge Data Eng. 2*, 1 (March 1990).
21. DELORME, D., HOLM, M., LEE, W., PASSE, P., RICARD, G., TIMMS, G., JR., AND YOUNGREN, L. Database index journaling for enhanced recovery. U.S. Patent 4 ,819,156, IBM, April 1989
22. DIXON, G. N., PARRINGTON, G. D., SHRIVASTAVA, S., AND WHEATER, S. M. The treatment of persistent objects in Arjuna. *Comput. J. 32*, 4 (1989).
23. DUCHAMP, D. Transaction management. Ph.D. dissertation, Tech. Rep. CMU-CS-88-192, Carnegie-Mellon Univ., Dec. 1988.
24. EFFELSBERG, W., AND HAERDER, T. Principles of database buffer management. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984).
25. ELHARDT, K , AND BAYER, R. A database cache for high performance and fast restart in database systems. *ACM Trans Database Syst. 9*, 4 (Dec. 1984).
26. FEKETE, A., LYNCH, N., MERRITT, M., AND WEIHL, W. Commutativity-based locking for nested transactions. Tech. Rep. MIT/LCS/TM-370.b, MIT, July 1989.
27. FOSSUM, B Data base integrity as provided for by a particular data base management system. In *Data Base Management*, J. W. Klimbie and K. L. Koffeman, Eds., North-Holland, Amsterdam, 1974.
28. GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS Fast Path. *IEEE Database Eng. 8*, 2 (June 1985).
29. GARZA, J., AND KIM, W. Transaction management in an object-oriented database system. In *Proceedings ACM-SIGMOD International Conference on Management of Data* (Chicago, June 1988).
30. GHEITH, A., AND SCHWAN, K. CHAOS^{art}: Support for real-time atomic transactions. In *Proceedings 19th International Symposium on Fault-Tolerant Computing* (Chicago, June 1989).
31. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the System R database manager. *ACM Comput. Surv. 13*, 2 (June 1981).
32. GRAY, J. Notes on data base operating systems. In *Operating Systems—An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, Eds., LNCS Vol. 60, Springer-Verlag, New York, 1978.
33. HADZILACOS, V. A theory of reliability in database systems. *J. ACM 35*, 1 (Jan. 1988), 121-145.
34. HAERDER, T. Handling hot spot data in DB-sharing systems. *Inf. Syst. 13*, 2 (1988), 155-166.

35. HADERLE, D., AND JACKSON, R. IBM Database 2 overview. *IBM Syst. J. 23,* 2 (1984).
36. HAERDER, T., AND REUTER, A. Principles of transaction oriented database recovery—A taxonomy. *ACM Comput. Surv. 15*, 4 (Dec. 1983).
37. HELLAND, P. The TMF application programming interface: Program to program communication, transactions, and concurrency in the Tandem NonStop system. Tandem Tech. Rep. TR89.3, Tandem Computers, Feb. 1989.
38. HERLIHY, M., AND WEIHL, W. Hybrid concurrency control for abstract data types. In *Proceedings 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, Tex., March 1988).
39. HERLIHY, M., AND WING, J. M. Avalon: Language support for reliable distributed systems. In *Proceedings 17th International Symposium on Fault-Tolerant Computing* (Pittsburgh, Pa., July 1987).
40. HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. Recovery management in QuickSilver. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988), 82–108.
41. *IMS/VS Version 1 Release 3 Recovery/Restart*. Doc. GG24-1652, IBM, April 1984.
42. *IMS/VS Version 2 Application Programming*. Doc. SC26-4178, IBM, March 1986.
43. *IMS/VS Extended Recovery Facility (XRF): Technical Reference*. Doc. GG24-3153, IBM, April 1987.
44. *IBM Workstation Data Save Facility/VM: General Information*. Doc. GH24-5232, IBM, 1990.
45. KORTH, H. Locking primitives in a database system. *JACM 30*, 1 (Jan. 1983), 55–79.
46. LUM, V., DADAM, P., ERBE, R., GUENAUER, J., PISTOR, P., WALCH, G., WERNER, H., AND WOODFILL, J. Design of an integrated DBMS to support advanced applications. In *Proceedings International Conference on Foundations of Data Organization* (Kyoto, May 1985).
47. LEVINE, F., AND MOHAN, C. Method for concurrent record access, insertion, deletion and alteration using an index tree. U.S. Patent 4,914,569, IBM, April 1990.
48. LEWIS, R. Z. *IMS Program Isolation Locking*. Doc. GG66-3193, IBM Dallas Systems Center, Dec. 1990.
49. LINDSAY, B., HAAS, L., MOHAN, C., WILMS, P., AND YOST, R. Computation and communication in R*: A distributed database manager. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984). Also in *Proceedings 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, Oct. 1983). Also available as IBM Res. Rep. RJ3740, San Jose, Calif., Jan. 1983.
50. LINDSAY, B., MOHAN, C., AND PIRAHESH, H. Method for reserving space needed for "rollback" actions. *IBM Tech. Disclosure Bull. 29*, 6 (Nov. 1986).
51. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983).
52. LINDSAY, B., SELINGER, P., GALTIERI, C., GRAY, J., LORIE, R., PUTZOLU, F., TRAIGER, I., AND WADE, B. Notes on distributed databases. IBM Res. Rep. RJ2571, San Jose, Calif., July 1979.
53. MCGEE, W. C. The information management system IMS/VS—Part II: Data base facilities; Part V: Transaction processing facilities. *IBM Syst. J. 16*, 2 (1977).
54. MOHAN, C., HADERLE, D., WANG, Y., AND CHENG, J. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In *Proceedings International Conference on Extending Data Base Technology* (Venice, March 1990). An expanded version of this paper is available as IBM Res. Rep. RJ7341, IBM Almaden Research Center, March 1990.
55. MOHAN, C., FUSSELL, D., AND SILBERSCHATZ, A. Compatibility and commutativity of lock modes. *Inf. Control 61*, 1 (April 1984). Also available as IBM Res. Rep. RJ3948, San Jose, Calif., July 1983.
56. MOSS, E., GRIFFETH, N., AND GRAHAM, M. Abstraction in recovery management. In *Proceedings ACM SIGMOD International Conference on Management of Data* (Washington, D.C., May 1986).
57. MOHAN, C. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *Proceedings 16th International Conference on Very Large Data Bases* (Brisbane, Aug. 1990). Another version of this paper is available as IBM Res. Rep. RJ7008, IBM Almaden Research Center, Sept. 1989.

58. MOHAN, C. Commit-LSN: A novel and simple method for reducing locking and latching in transaction processing systems In *Proceedings 16th International Conference on Very Large Data Bases* (Brisbane, Aug. 1990). Also available as IBM Res. Rep. RJ7344, IBM Almaden Research Center, Feb. 1990.
59 MOHAN, C. ARIES/LHS: A concurrency control and recovery method using write-ahead logging for linear hashing with separators. IBM Res. Rep., IBM Almaden Research Center, Nov. 1990.
60. MOHAN, C. A cost-effective method for providing improved data availability during DBMS restart recovery after a failure In *Proceedings of the 4th International Workshop on High Performance Transaction Systems* (Asilomar, Calif., Sept. 1991). Also available as IBM Res. Rep. RJ8114, IBM Almaden Research Center, April 1991.
61. MOSS, E., LEBAN, B., AND CHRYSANTHIS, P. Fine grained concurrency for the database cache. In *Proceedings 3rd IEEE International Conference on Data Engineering* (Los Angeles, Feb. 1987).
62. MOHAN, C., AND LEVINE, F. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. IBM Res. Rep. RJ6846, IBM Almaden Research Center, Aug. 1989.
63. MOHAN, C., AND LINDSAY, B. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). Also available as IBM Res. Rep. RJ3881, IBM San Jose Research Laboratory, June 1983.
64. MOHAN, C., LINDSAY, B., AND OBERMARCK, R. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986).
65. MOHAN, C., AND NARANG, I. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings 17th International Conference on Very Large Data Bases* (Barcelona, Sept. 1991). A longer version is available as IBM Res. Rep. RJ8017, IBM Almaden Research Center, March 1991.
66. MOHAN, C., AND NARANG, I. Efficient locking and caching of data in the multisystem shared disks transaction environment. In *Proceedings of the International Conference on Extending Database Technology* (Vienna, Mar. 1992). Also available as IBM Res. Rep. RJ8301, IBM Almaden Research Center, Aug. 1991.
67. MOHAN, C., NARANG, I., AND PALMER, J. A case study of problems in migrating to distributed computing: Page recovery using multiple logs in the shared disks environment. IBM Res. Rep. RJ7343, IBM Almaden Research Center, March 1990.
68. MOHAN, C., NARANG, I., SILEN, S. Solutions to hot spot problems in a shared disks transaction environment. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems* (Asilomar, Calif., Sept. 1991). Also available as IBM Res Rep. 8281, IBM Almaden Research Center, Aug. 1991.
69. MOHAN, C., AND PIRAHESH, H. ARIES-RRH: Restricted repeating of history in the ARIES transaction recovery method. In *Proceedings 7th International Conference on Data Engineering* (Kobe, April 1991). Also available as IBM Res. Rep. RJ7342, IBM Almaden Research Center, Feb. 1990
70. MOHAN, C , AND ROTHERMEL, K. Recovery protocol for nested transactions using write-ahead logging. *IBM Tech. Disclosure Bull. 31*, 4 (Sept 1988).
71. MOSS, E. Checkpoint and restart in distributed transaction systems. In *Proceedings 3rd Symposium on Reliability in Distributed Software and Database Systems* (Clearwater Beach, Oct. 1983).
72. MOSS, E Log-based recovery for nested transactions. In *Proceedings 13th International Conference on Very Large Data Bases* (Brighton, Sept. 1987).
73. MOHAN, C., TRIEBER, K., AND OBERMARCK, R. Algorithms for the management of remote backup databases for disaster recovery. IBM Res. Rep. RJ7885, IBM Almaden Research Center, Nov. 1990.
74. NETT, E., KAISER, J., AND KROGER, R. Providing recoverability in a transaction oriented distributed operating system. In *Proceedings 6th International Conference on Distributed Computing Systems* (Cambridge, May 1986).

75. NOE, J., KAISER, J., KROGER, R., AND NETT, E. The commit/abort problem in type-specific locking. GMD Tech. Rep. 267, GMD mbH, Sankt Augustin, Sept. 1987.
76. OBERMARCK, R. IMS/VS program isolation feature. IBM Res. Rep. RJ2879, San Jose, Calif., July 1980.
77. O'NEILL, P. The Escrow transaction method. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986).
78. ONG, K. SYNAPSE approach to database recovery. In *Proceedings 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Waterloo, April 1984).
79. PEINL, P., REUTER, A., AND SAMMER, H. High contention in a stock trading database: A case study. In *Proceedings ACM SIGMOD International Conference on Management of Data* (Chicago, June 1988).
80. PETERSON, R. J., AND STRICKLAND, J. P. Log write-ahead protocols and IMS/VS logging. In *Proceedings 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Atlanta, Ga., March 1983).
81. RENGARAJAN, T. K., SPIRO, P., AND WRIGHT, W. High availability mechanisms of VAX DBMS software. *Digital Tech. J. 8* (Feb. 1989).
82. REUTER, A. A fast transaction-oriented logging scheme for UNDO recovery. *IEEE Trans. Softw. Eng. SE-6*, 4 (July 1980).
83. REUTER, A. Concurrency on high-traffic data elements. In *Proceedings ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Los Angeles, March 1982).
84. REUTER, A. Performance analysis of recovery techniques. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 526–559.
85. ROTHERMEL, K., AND MOHAN, C. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Proceedings 15th International Conference on Very Large Data Bases* (Amsterdam, Aug. 1989). A longer version of this paper is available as IBM Res. Rep. RJ6650, IBM Almaden Research Center, Jan. 1989.
86. ROWE, L., AND STONEBRAKER, M. The commercial INGRES epilogue. Ch. 3 in *The INGRES Papers*, Stonebraker, M., Ed., Addson-Wesley, Reading, Mass., 1986.
87. SCHWARZ, P., CHANG, W., FREYTAG, J., LOHMAN, G., MCPHERSON, J., MOHAN, C., AND PIRAHESH, H. Extensibility in the Starburst database system. In *Proceedings Workshop on Object-Oriented Data Base Systems* (Asilomar, Sept. 1986). Also available as IBM Res. Rep. RJ5311, San Jose, Calif., Sept. 1986.
88. SCHWARZ, P. Transactions on typed objects. Ph.D. dissertation, Tech. Rep. CMU-CS-84-166, Carnegie Mellon Univ., Dec. 1984.
89. SHASHA, D., AND GOODMAN, N. Concurrent search structure algorithms. *ACM Trans. Database Syst. 13*, 1 (March 1988).
90. SPECTOR, A., PAUSCH, R., AND BRUELL, G. Camelot: A flexible, distributed transaction processing system. In *Proceedings IEEE Compcon Spring '88* (San Francisco, Calif., March 1988).
91. SPRATT, L. The transaction resolution journal: Extending the before journal. *ACM Oper. Syst. Rev. 19*, 3 (July 1985).
92. STONEBRAKER, M. The design of the POSTGRES storage system. In *Proceedings 13th International Conference on Very Large Data Bases* (Brighton, Sept. 1987).
93. STILLWELL, J. W., AND RADER, P. M. *IMS/VS Version 1 Release 3 Fast Path Notebook*. Doc. G320-0149-0, IBM, Sept. 1984.
94. STRICKLAND, J., UHROWCZIK, P., AND WATTS, V. IMS/VS: An evolving system. *IBM Syst. J. 21*, 4 (1982).
95. THE TANDEM DATABASE GROUP. NonStop SQL: A distributed, high-performance, high-availability implementation of SQL. In *Lecture Notes in Computer Science Vol. 359*, D. Gawlick, M. Haynie, and A. Reuter, Eds., Springer-Verlag, New York, 1989.
96. TENG, J., AND GUMAER, R. Managing IBM Database 2 buffers to maximize performance. *IBM Syst. J. 23*, 2 (1984).
97. TRAIGER, I. Virtual memory management for database systems. *ACM Oper. Syst. Rev. 16*, 4 (Oct. 1982), 26–48.
98. VURAL, S. A simulation study for the performance analysis of the ARIES transaction recovery method. M.Sc. thesis, Middle East Technical Univ., Ankara, Feb. 1990.

99 WATSON, C. T., AND ABERLE, G. F System/38 machine database support. In *IBM Syst. 38/Tech. Dev.*, Doc. G580-0237, IBM July 1980.
100. WEIKUM, G. Principles and realization strategies of multi-level transaction management. *ACM Trans. Database Syst. 16*, 1 (Mar. 1991).
101. WEINSTEIN, M., PAGE, T., JR , LIVEZEY, B., AND POPEK, G. Transactions and synchronization in a distributed operating system. In *Proceedings 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Dec. 1985).

Received January 1989; revised November 1990; accepted April 1991

Segment-Based Recovery: Write-ahead logging revisited

Russell Sears
UC Berkeley
sears@cs.berkeley.edu

Eric Brewer
UC Berkeley
brewer@cs.berkeley.edu

Although existing write-ahead logging algorithms scale to conventional database workloads, their communication and synchronization overheads limit their usefulness for modern applications and distributed systems. We revisit write-ahead logging with an eye toward finer-grained concurrency and an increased range of workloads, then remove two core assumptions: that pages are the unit of recovery and that timestamps (LSNs) should be stored on each page.

Recovering individual application-level objects (rather than pages) simplifies the handing of systems with object sizes that differ from the page size.

We show how to remove the need for LSNs on the page, which in turn enables DMA or zero-copy I/O for large objects, increases concurrency, and reduces communication between the application, buffer manager and log manager. Our experiments show that the looser coupling significantly reduces the impact of latency among the components. This makes the approach particularly applicable to large scale distributed systems, and enables a "cross pollination" of ideas from distributed systems and transactional storage.

However, these advantages come at a cost; segments are incompatible with physiological redo, preventing a number of important optimizations. We show how allocation enables (or prevents) mixing of ARIES pages (and physiological redo) with segments. We present an allocation policy that avoids undesirable interactions that complicate other combinations of ARIES and LSN-free pages, and then present a proof that both approaches and our combination are correct.

Many optimizations presented here were proposed in the past. However, we believe this is the first unified approach.

1. INTRODUCTION

Transactional recovery is at the core of most durable storage systems, such as databases, journaling filesystems, and a wide range of web services and other scalable storage architectures. Write-ahead logging algorithms from the database literature were traditionally optimized for small, concurrent, update-in-place transactions, and later extended for larger objects such as images and other file types.

Although many systems, such as filesystems and web services, require weaker semantics than relational databases, they still rely upon durability and atomicity for some information. For example, filesystems must ensure that metadata (e.g. inodes) are kept consistent, while web services must not corrupt account or billing information.

In practice, this forces them to provide recovery for some subset of the information they handle. Many such systems opt to use special purpose *ad hoc* approaches to logging and recovery. We argue that database-style recovery provides a conceptually cleaner approach than such approaches and that, with a few extensions, can more efficiently address a wide range of workloads and trade off between full ACID and weaker semantics.

Given these broader goals, and roughly twenty years of innovation, we revisit the core of write-ahead logging. We present *segment-based recovery*, a new approach that provides more flexibility and higher concurrency, enables distributed solutions, and that is simple to implement and reason about.

In particular, we revisit and reject two traditional assumptions about write-ahead logging:

- The disk page is the basic unit of recovery.
- Each page contains a *log-sequence number (LSN)*.

This pair of assumptions permeates write-ahead logging from at least 1984 onward [7], and is codified in ARIES [26] and in early books on recovery [2]. ARIES is essentially a mechanism for transactional pages: updates are tracked per page in the log, a timestamp (the LSN) is stored per page, and pages can be recovered independently.

However, applications work with variable-sized records or objects, and thus there may be multiple objects per page or multiple pages per object. Both kinds of mismatch introduce problems, which we cover in Section 3. Our original motivation was that having an LSN on each page prevents use of contiguous disk layouts for multi-page objects. This is incompatible with DMA (zero-copy I/O), and worsens as object sizes increase over time.

Presumably, writing a page to disk was once an atomic operation, but that time has long passed. Nonetheless, traditional recovery stores the LSN in the page so it can be atomically written with the data [2, 5]. Several mechanisms have been created to make this assumption true with modern disks [8, 31, 34] (Section 2.1), but disk block atomicity is now *enforced* rather than inherent and thus is not a reason *per se* to use pages as the unit of recovery.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '09, August 24-28, 2009, Lyon, France
Copyright 2009 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

We present an approach that is similar to ARIES, but that works at the granularity of application data. We refer to this unit of recovery as a *segment*: a set of bytes that may span page boundaries. We also present a generalization of segment-based recovery and ARIES that allows the two to coexist. Aligning segment boundaries with higher-level primitives simplifies concurrency and enables new optimizations, such as zero-copy I/O for large objects.

Our distinction between segments and pages is similar to that of computer architecture. Our segments differ from those in architecture in that we are using them as a mechanism for recovery rather than for protection. Pages remain useful both for space management and as the unit of transfer to and from disk. Pages and segments work well together (as in architecture), and in our case preserve compatibility with conventional page-oriented data structures such as B-trees.

Our second contribution is to show how to use segment-based recovery to eliminate the need for LSNs on pages. *LSN-free pages* facilitate multi-page objects and, by making page timestamps implicit, allow us to reorder updates to the same page and leverage higher-level concurrency.

However, segment-based redo is restricted to *blind writes*: operations that do not examine the pages they modify. Typically, blind writes either zero out a range or write an array of bytes at an offset. In contrast, ARIES redo examines the contents of on-disk pages and supports *physiological* redo. Physiological redo assumes that each page is internally consistent, and stores headers on each page. This allows the system to reorganize the page then write back the update without generating a log entry. This is especially important for B-trees, which frequently consolidate space within pages. Also, with carefully ordered page write back, physiological operations make it possible to rebalance B-tree nodes without logging updates.

Third, we present a simple proof that segment-oriented recovery and ARIES are correct. We document the trade offs between page- and segment-oriented recovery in greater detail and show how to build hybrid systems that migrate pages between the two techniques. The main challenge in the hybrid case is page reallocation. Surprisingly, allocators have long-plagued implementers of transactional storage.

Finally, segment-oriented recovery enables a number of novel distributed recovery architectures that are hindered by the tight coupling of components required by page-oriented recovery. The distributed variations are quite flexible and enable recovery to be a large-scale distributed service.

2. WRITE-AHEAD LOGGING

Recovery algorithms are often categorized as either *update-in-place* or based on *shadow copies*. Shadow copy mechanisms work by writing data to a new location, syncing it to disk and then atomically updating a pointer to point to the new location. This works reasonably well for large objects, but incurs a number of overheads due to fragmentation and disk seeks. Write-ahead logging provides update-in-place changes: a redo and/or undo log entry is written to the log before the update-in-place so that it can be redone or undone in case of a crash. Write-ahead logging is generally considered superior to shadow pages [4].

ARIES and other modern transactional storage algorithms provide *steal/no-force* recovery [15]. *No-force* means that the page need not be written back on commit, because a redo log entry can recreate the page during recovery should it get lost. This avoids random writes during commit. *Steal* means that the buffer manager can write out dirty pages, as long as there is a durable undo log entry that can recreate the overwritten data after an abort/crash. This allows the buffer manager to reclaim buffer space even from in-progress transactions. Together, they allow the buffer manager to write back pages before (steal) or after (no-force) commit as convenient. This approach has stood the test of time and underlies a wide range of commercial databases.

The primary disadvantage of *steal/no-force* is that it must log undo and redo information for each object that is updated. In ARIES' original context (relational databases) this was unimportant, but as disk sizes increased, large objects became increasingly common and most systems introduced support for *steal/force* updates for large objects. *Steal/force* avoids redo logging. If the write goes to newly allocated (empty) space, it also avoids undo logging. In some respect, such updates are simply shadow pages in disguise.

2.1 Atomic Page Writes?

Hard disks corrupt data in a number of different ways, each of which must be dealt with by storage algorithms. Although segment-based recovery is not a panacea, it has some advantages over page-based techniques. Errors such as catastrophic failures and reported read and write errors are *detectable*. Others are more subtle, but nonetheless need to be handled by storage algorithms. *Silent data corruption* occurs when a drive read does not match a drive write. In principle, checksumming in modern hardware prevents this from happening. In practice, marginal drive controllers and motherboards may flip bits before the checksum is computed, and drives occasionally write valid checksummed data to the wrong location. Checksummed page offsets often allow such errors to be detected [8]. However, since the drive exhibits arbitrary behavior in these circumstances, the only reliable repair technique, *media recovery*, is quite expensive, and starts with a backup *checkpoint* of the page. It then applies every relevant log entry that was generated after the checkpoint was created.

A second, more easily handled, set of problems occurs not because of *what* data the drive stores, but *when* that data reaches disk. If write caching is enabled, some operating systems (such as Linux) return from synchronous writes before data reaches the platter, violating the write-ahead invariant [28]. This can be addressed by disabling write caching, adding an uninterruptable power supply, or by using an operating system that provides synchronous writes.

However, even synchronous writes do not atomically update pages. Two solutions to this problem are *torn page detection* [31], which writes the LSN of the page on each sector and *doublewrite* buffering [34], which, in addition to the recovery log, maintains a second write-ahead log of all requests issued to the hard disk. Torn page detection has minimal log overhead, but relies on media recovery to repair the page, while doublewrite buffering avoids media recovery, but greatly increases the number of bytes logged. Doublewrite buffering also avoids issuing synchronous seek requests, giving the operating system and hard drive more freedom to schedule disk head movement.

Assuming sector writes are atomic, segment-based recovery's blind writes repair torn pages without resorting to media recovery or introducing additional logging overhead (beyond preventing the use of physiological logging).

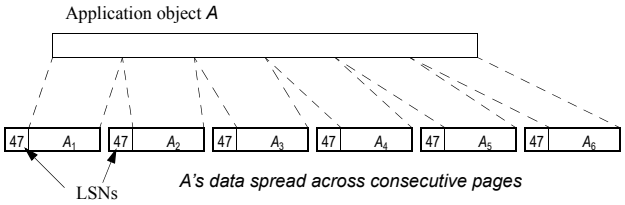


Figure 1: Per page LSNs break up large objects.

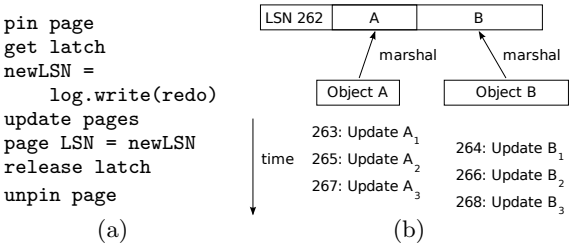


Figure 2: (a) Record update in ARIES. Pinning the page prevents the buffer manager from stealing it during the update, while the latch prevents races on the page LSN among independent updates. **(b)** A sequence of updates to two objects stored on the same page. With ARIES, A_1 is marshaled, then B_1, A_2 and so on. Segments avoid the page latch, and need only update the page once for each record.

3. PAGE-ORIENTED RECOVERY

In the next four subsections, we examine the fundamental constraints imposed by making pages the unit of recovery. A core invariant of page-oriented recovery is that each page is self-consistent and marked with an LSN. Recovery uses the LSN to ensure that each redo entry is applied exactly once.

3.1 Multi-page Objects

The most obvious limitation of page-oriented recovery is that it is awkward when the real record or object is larger than a page. Figure 1 shows a large object A broken up into six consecutive pages. Even though the pages are consecutive, the LSNs break up the continuity and require complex and expensive copying to reassemble the object on every read and spread it out on every write (analogous to *segmentation and reassembly* into packets in networking).

Segment-oriented recovery eschews per page LSNs, allowing it to store the object as a contiguous segment. This enables the use of DMA and zero-copy I/O, which have had significant impact in filesystems [9, 32].

3.2 Application/Buffer Interaction

Figure 2(a) shows the typical sequence for updating a single record on a page, which keeps the on-page version in sync with the log by updating them together atomically. In a traditional database, in which the page contains a record, this is not a problem; the in-memory version of the page is the natural place to keep the current version.

However, this creates problems when the in-memory page is not the natural place to keep the current version, such as when an application maintains its own working copies, and stores them in the database via either marshaling or an object-relational mapping [14, 16]. Other examples include BerkeleyDB [30], systems that treat relational databases as "key-value" storage [34], and systems that provide such primitives across many machines [6, 22].

Figure 2(b) shows two independent objects, A and B, that happen to share the same page. For each update, we would like to generate a log entry and update the object *without* having to serialize each update back onto the page. In theory, the log entries should be sufficient to roll forward the object from the page as is. However, with page-oriented recovery this will not work. Assume A has written the log entry for A_1 but has not yet updated the page. If B, which is completely independent, decides to then write the log entry for B_1 and update the page, the LSN will be that of B's entry. Since B_1 came after A_1, the LSN implies that the changes from A_1 are reflected in the page even though they are not, and recovery may fail.

In essence, the page LSN is imposing artificial ordering constraints between independent objects: updates from one object set the timestamp of the other. This is essentially *write through caching*: every update must be written all the way through to the page.

What we want is *write back caching*: updates affect only the cache copy and we need only write the page when we evict the object from the cache. One solution is to store a separate LSN with every object. However, when combined with dynamic allocation, this prevents recovery from determining whether or not a set of bytes contains an LSN (since the usage varies over time). This leads to a *second* write-ahead log, incurring significant overhead [3, 21].

Segment-oriented recovery avoids this and supports write back caching (Section 7.2). In the case above, the page has different LSNs for A and B, but neither LSN is explicitly stored. Instead, recovery estimates the LSNs and recovers A and B independently; each object is its own segment.

3.3 Log Reordering

Having an LSN on each page also makes it difficult to reorder log entries, even between independent transactions. This interferes with mechanisms that prioritize important requests, and as with the buffer manager, tightly couples the log to the application, increasing synchronization and communication overheads.

In theory, all independent log entries could be reordered, as long as the order within objects and within transactions (e.g. the commit record) is maintained. However, in general even updates in two independent transactions cannot be reordered because they might share pages. Once an LSN is assigned to log entries on a shared page, the order of the independent updates is fixed.

With segment-oriented recovery we do not need to even *know* the LSN at the time of a page update, and can assign LSNs later if we choose. In some cases we assign LSNs at the time of writing the log to disk, which allows us to place high-priority entries at the *front* of the log buffer. Section 7.3 presents the positive impact this has on high-priority transactions. Before journaling was common, local filesystems supported such reordering. The Echo [23] distributed filesystem preserved these optimizations by layering a cache on top of a no-steal, non-transactional journaled filesystem.

Note that for dependent transactions, higher-level locks (isolation) constrain the order, and the update will block before it creates a log entry. Thus we are reordering transactions only in ways that preserve serializability.

3.4 Distributed recovery

Page-oriented recovery leads to a tight coupling between the application, the buffer manager and the log manager. Looking again at Figure 2, we note that the buffer manager must hold the latch across the call to the log manager so that it can atomically update the page with the correct LSN.

The tight coupling might be fine on a traditional single core machine, but it leads to performance issues when distributing the components to different machines and to a lesser extent, to different cores. Segment-oriented recovery enables simpler and looser coupling among components.

- Write back caching reduces communication between the buffer manager and application, since the communication occurs only on cache eviction.
- There is no need to latch the page during an update, since there is no shared state. (Races within one object are handled by higher-level locking.) Thus calls to the buffer manager and log manager can be asynchronous, hiding network latency.
- The use of natural layouts for large objects allows DMA and zero-copy I/O in the local case. In the distributed case, this allows application data to be written without copying the data and the LSNs to the same machine.

In turn, the ability to distribute these components means that they can be independently sized, partitioned and replicated. It is up to the system designer to choose partitioning and replication schemes, which components will coexist on the same machines, and to what extent calls to the underlying network primitives may be amortized and reordered.

This allows for very flexible large-scale write-ahead logging as a service for cloud computing, much the same way that two-phase commit or Paxos [18] are useful services.

3.5 Benefits from Pages

Pages provide benefits that complement segment-based approaches. They provide a natural unit for partitioning storage for use by different components; in particular, they enable the use of page headers that describe the layout of information on disk. Also, data structures such as B-trees are organized along page boundaries. This guarantees good locality for data that is likely to be accessed as a unit.

Furthermore, some database operations are significantly less expensive with page-oriented recovery. The most important is *page compaction*. Systems with atomic pages can make use of *physiological* updates that examine metadata, such as on-page tables of slot offsets. To compact such a page, page-based systems simply pin the page, defragment the page's free space, then unpin the page. In contrast, segment-based systems cannot rely on page metadata at redo and record such modifications in the log.

It may also make sense to build a B-tree using pages for internal nodes, and segments for the leaves. This would allow index nodes to benefit from physiological logging, but would provide high concurrency updates, reduced fragmentation and the other benefits of segments for the operations that read and write the data (as opposed to the keys) stored in the tree.

Page-oriented recovery simplifies the buffer manager because all pages are the same size, and objects do not span pages. Thus, the buffer manager may place a page at any point in its address space, then pass that pointer to the code interested in the page. In contrast, segment boundaries are less predictable and may change over time. This makes it difficult for the buffer manager to ensure that segments are contiguous in memory, although this problem is less serious with modern systems and large address spaces. Because pages and segments have different advantages, we are careful to allow them to safely coexist.

4. SEGMENT-BASED RECOVERY

This section provides an overview of ARIES and segments, and sketches a possible implementation of segment-based storage. This implementation is only one variant of our approach, and is designed to highlight the changes made by our proposal, not explain how to best use segments. Section 5 presents segments in terms of invariants that encompass a wide range of implementations.

Write-ahead logging systems consist of four components:

- The *log file* contains an in-order record of each operation. It consists of entries that contain an LSN (the offset into the log), the id of the transaction that generated the entry, which segment (or object) the entry changed, a boolean to show if the segment contains an LSN, and enough information to allow the modification to be repeated (we treat this as an operation implemented by the entry, e.g., `entry->redo()`). Recent entries may still reside in RAM, but older entries are stored on disk. Log *truncation* limits the log's size by erasing the earliest entries once they are no longer needed.
- The *application cache* is not part of the storage implementation. Instead, it is whatever in-memory representation the application uses to represent the data. It is often overlooked in descriptions of recovery algorithms; in fact, database implementations often avoid such caches entirely.
- The *buffer manager* keeps copies of disk pages in main memory. It provides an API that tracks LSNs and applies segment changes from the application cache to the buffers. In traditional ARIES, it presents a coherent view of the data. *Coherent*[1] means that changes are reflected in log order, which means that reads from the buffer manager immediately reflect updates performed by the application. Segment-based recovery allows applications to log updates (and perhaps update their own state), then defer and reorder the writes to the buffer manager. This leads to *incoherent* buffer managers that may return stale, contradictory data to the application. It is up to the application to decide when it is safe to read recently updated segments.
- The *page file* backs the buffer manager on disk and is incoherent. ARIES (and our example implementation) manipulates entire pages at a time; though segment-based systems could manipulate segments instead.

In page-based systems, each page is exactly one segment. Segment-based systems relax this and define segments to

[1] *Coherent* refers to a set of invariants analogous to those ensured by cache coherency protocols.

```
if(s->lsn_volatile <= log_stable) {
  write(s);
  s->lsn_stable = infinity;
  s->lsn_volatile = 0;
}
```

(a) Flush segment s to disk

```
s->lsn_stable =
   min(s->lsn_stable, entry->lsn);
s->lsn_volatile =
   max(s->lsn_volatile, entry->lsn);
entry->redo(s);
```

(b) Apply log entry to segment s

```
op_lsn = min<lsn_of_current_operations>;
t_lsn = min<start_lsn_of_current_xacts>;
s_lsn = min<segments->lsn_stable>;
log->truncate(min(op_lsn,t_lsn,s_lsn));
```

(c) Truncate log

Figure 3: Runtime operations for a segmented buffer manager. Page based buffer managers are identical, except their operations work against pages, causing **(b)** to split updates into multiple operations.

be arbitrary sets of *individually updatable* bytes; flushing a segment to disk cannot inadvertently change bytes outside the segment, even during a crash. There may be many higher-level objects per segment (records in a B-tree node) or many segments per object (arbitrary-length records). In both cases, storage deals with updates to one segment at a time.

Crucially, segments decouple application primitives (redo entries) from buffer management (disk operations). Regardless of whether the buffer manager provides a page or segment API, the data it contains is organized in terms of segments that represent higher level objects and are backed by disk sectors.

With a page API, updates to segments that span pages pin each page, manipulate a piece of the segment, then release the page. This works because blind writes will repair any torn (partially updated) segments, and because we assume that higher level code will latch segments as they are being written.

The key idea is to use segments to decouple updates from pages, allowing the application to choose the update granularity. This allows the requests to be reordered without regard to page boundaries.

The primary changes to forward operation relate to LSN tracking. Figure 3 describes a buffer manager that works with segments; paged buffer managers are identical, except that LSN tracking and other operations are per page, rather than per segment. `s->lsn_stable` is the first LSN that changed the in-memory copy of a page; `s->lsn_volatile` is the latest such value. If a page contains an LSN, then flushing it to disk sets the on-disk LSN to `s->lsn_volatile`.

If updates are applied in order, `s->lsn_stable` will only be changed when the page first becomes dirty. However, with reordering every update must check the LSN.

Write-ahead is enforced at page flush, which compares `s->lsn_volatile` to `log_stable`, the LSN of the most recent log entry to reach disk.

Truncation uses `s->lsn_stable` to avoid deleting log entries that recovery would need in order to bring the on-disk version of the page up-to-date. Because of reordering, truncation must also consider updates that have not reached the buffer manager. It also must avoid deleting undo entries that were produced by incomplete transactions.

4.1 Recovery

Like ARIES, segment-based recovery has three phases:

1. *Analysis* examines the log and constructs an estimate of the buffer manager's contents at crash. This allows later phases to ignore portions of the log.
2. *Redo* brings the system back into a state that existed before crash, including any incomplete transactions. This process is called *repeating history*.
3. *Undo* rolls back incomplete transactions and logs *compensation* records to avoid redundant work due to multiple crashes.

Also like ARIES, our approach supports *steal/no-force*. The actions performed by log entries are constrained to *physical redo*, which can be applied even if the system is inconsistent, and *logical undo*, which is necessary for concurrent transactions. Logical undo allows transactions to safely roll back after the underlying data has changed, such as when another transaction's B-tree insertion has rebalanced a node.

Hybrid redo

```
foreach(redo entry) {
  if(entry->clears_contents())
    segment->corrupt = false;
  if(entry->is_lsn_free()) {
    entry->redo(segment);
  } else if(segment->LSN < entry->LSN) {
    segment->LSN = entry->LSN
    error = entry->redo(segment);
    if(error) segment->corrupt = true;
  }
}
```

Unlike ARIES, which uses `segment->LSN` to ensure that each redo is applied *exactly once*, recovery always applies LSN-free redos, guaranteeing they reach the segment *at-least-once*. Hybrid systems, which allow ARIES and segments to coexist, introduce an additional change; they allow redo to temporarily corrupt pages.

This happens because segments store application data where ARIES would store an LSN and page header, leaving redo with no way to tell whether or not to apply ARIES-style entries. To solve this problem, hybrid systems zero out pages that switch between the two methods:

Switch page between ARIES and segment-based recovery

```
log(transaction id, segment id, new page type);
clear_contents(segment);
initialize_page_header(segment, new page type);
```

This ensures that recovery repairs any corruption caused by earlier redos.

4.2 Examples

We now present pseudocode for segment-based indexes and large objects.

Insert value into B-Tree node

```
make in-memory preimage of page
insert value into M'th of N slots
log (transaction id, page id, binary diff of page)
```

Segment-based indexes must perform blind writes during redo. Depending on the page format and fragmentation,

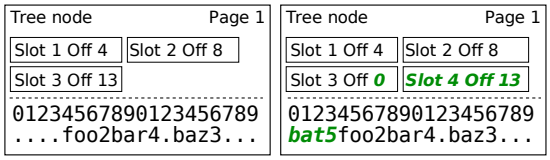


Figure 4: An internal tree node, before and after the pair (key="bat", page=5) is inserted.

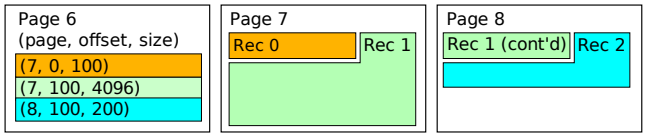


Figure 5: Records stored as segments. Colors correspond to (non-contiguous) bytes written by a single redo entry.

these entries could be relatively compact, as in Figure 4, or they could contain a preimage and postimage of the entire page, as would be the case if we inserted a longer key in Figure 4. In contrast, a conventional approach would simply log the slot number and the new value.

B-Tree concurrency is well-studied [20, 24], and largely unaffected by our approach. However, blind writes can incur significantly higher log overhead than physiological operations, especially for index operations. Fortunately, the two approaches coexist.

Update N segments

```
min_log = log->head

Spawn N parallel tasks; for each update:
  log (transaction id, offset, preimage, postimage)
Spawn N parallel tasks; for each update:
  pin and latch segment, s
  update s
  unlatch s
  s->lsn_stable = min(s->lsn_stable, min_log);

Wait for the 2N parallel tasks to complete
max_log = log->head

Spawn parallel tasks; for each segment, s:
  s->lsn_volatile = max(s->lsn_volatile, max_log);
  unpin s;
```

The latch is optional, and prevents concurrent access to the segment.[2] The pin prevents page flushes from violating the write-ahead invariant before `lsn_volatile` is updated. A system using the layout in Figure 5 and a page-based buffer manager would pin pages rather than segments and rely on higher level code to latch the segment.

Since the segments may happen to be stored on the same page, conventional approaches apply the writes in order, alternating between producing log entries and updating pages. Section 7 shows that this can incur significant overhead.

5. RECOVERY INVARIANTS

This section presents segment-based storage and ARIES in terms of first-order predicate logic. This allows us to prove the correctness of concurrent transactions and allocation. Unlike Kuo's proof [17] for ARIES, we do not present or prove correct a set of mechanisms that maintain our invariants, nor do we make use of I/O automata. Also unlike that work, we cover full, concurrent transactions and latching; two often misunderstood aspects of ARIES that are important to system designers.

5.1 Segments and objects

This paper uses the term *object* to refer to a piece of data that is written without regard to the contents of the rest of the database. Each object is physically backed by a set of *segments*: atomically logged, arbitrary length regions of disk. Segments are stored using *machine primitives*[3]; we assume the hardware is capable of updating segments independently, perhaps with the use of additional mechanisms. Like ARIES, segment-based storage is based on multi-level recovery [33], which imposes a nested structure upon objects; the nesting can be exploded to find all of the segments.

Let s denote an address, or set of addresses, i.e., a segment, and l denote the LSN of a log entry (an integer). Then, define s_l to be the value of that segment after applying a prefix of the log to the initial value of s:

$$s_l = log_l(log_{l-1}(...(log_1(s_0))))$$

Let s_t^{mem} be the value stored in the buffer manager at time t or $\perp$ if the segment is not in the buffer manager. Let s_t^{stable} be the value on disk. If $s_t^{mem} = s_t^{stable}$ or $s_t^{mem} = \perp$, then we say s is *clean*. Otherwise, s is *dirty*.

Finally, $s_t^{current}$ is the value stored in s:

$$s_t^{current} = \begin{cases} s_t^{mem} & \text{if } s_t^{mem} \neq \perp \\ s_t^{stable} & \text{otherwise} \end{cases}$$

Systems with coherent buffer managers maintain the invariant that $s_t^{current} = s_{l(t)}$, where $l(t)$ is the LSN of the most recent log entry at time t. Incoherent systems allow $s_t^{current}$ to be stale, and maintain the weaker invariant that $\exists\ l' \leq l(t) : s_t^{current} = s_{l'}$.

A *page* is a range of contiguous bytes with pre-determined boundaries. Although pages contain multiple application-level objects, if they are updated atomically then recovery treats them as a single segment/object. Otherwise, for the purposes of this section, we treat them as an array of single-byte segments. A *record* is an object that represents a simple piece of data, such as a tuple. Other examples of objects are indexes, schemas, or anything else stored by the system.

5.2 Coherency vs. Consistency

We define the set:

$$LSN(O) = \{l : \ O_l = O\} \tag{1}$$

to be the set of all LSNs l where O_l was equal to some version, O, of the object. With page-oriented storage, each page s contains an LSN, $s.lsn$. These systems ensure that $s.lsn \in LSN(s)$, usually by setting it to the LSN of the log entry that most recently modified the page. If s is not a page, or does not contain an explicit LSN, then $s.lsn = \perp$.

Object O is *corrupt* ($O = \top$) if it is a segment that never existed during forward operation, or if it contains a corrupt object:

$$\exists \text{ segment } s \in O : \forall \text{ LSN } l, \ s \neq s_l \tag{2}$$

[2] We assume `s->lsn_stable` and `s->lsn_volatile` are updated atomically.

[3] We take the term *machine* from the virtualization literature.

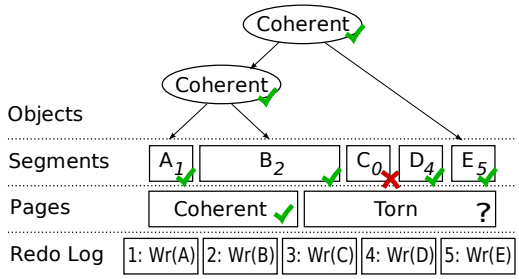


Figure 6: State of the system before redo; the data is incoherent (torn). Subscripts denote the most recent log entry to touch an object; Segment C is missing update 3. For the top level object $LSN(O) = \{5\}$. Segment B, the nested object and the coherent page have $LSN(O) = \{2, 3, 4, 5\}$. For the torn page, $LSN(O) = \emptyset$.

For the systems we consider, corruption only occurs due to faulty hardware or software, not system crashes. Repairing corrupted data is expensive, and requires access to a backed-up checkpoint of the database and all log entries generated since the checkpoint was taken. The process is analogous to recovery's redo phase; we omit the details.

Instead, the recovery algorithms we present here deal with two other classes of problems: torn (incoherent) data, and inconsistent data. An object O is *torn* if it is not corrupt and $LSN(O) = \emptyset$. In other words, the object was partially written to disk. Figure 6 shows some examples of torn objects as they might exist at the beginning of recovery.

An object O is *coherent* when it is in a state that arose during forward operation (perhaps mid-transaction):

$$\exists \text{ LSN } l : \forall \text{ object } o \in O, l \in LSN(o) \tag{3}$$

LEMMA 1. *O is coherent if and only if it is not torn.*

PROOF. *To show*

$$(\exists\, l : \forall\, s \in O, l \in LSN(s)) \iff (\exists\, l' \in LSN(O))$$

choose $l' = l$. For the $\Rightarrow$ case, each s is equal to s_l so O must be equal to O_l. By definition, $l \in LSN(O_l)$. The remaining case is analogous. □

Even though "torn" and "incoherent" are synonyms, we follow convention and reserve "torn" for discussions of partially written disk pages (or segments). We use "incoherent" when talking about multi-segment objects and the buffer manager.

An object is *consistent* if it is coherent at an LSN that was generated when there were no *in-progress* modifications to the object. Like objects, modifications are nested; a modification is in-progress if some of its sub-operations have not yet completed. As a special case; a transaction is an operation over the database; an ACID database is consistent when there are no in-progress transactions.

Physical operations can be applied when the database is incoherent, while logical operations rely on object consistency. For example, overwriting a byte at a known offset is a physical operation and always succeeds; traversing a multi-page index and inserting a key is a logical operation. If the index is inconsistent, it may contain partial updates normally protected by latches, and the traversal may fail.

Next, we explain how redo uses physical operations to bring the database to a coherent, but inconsistent state. This is not quite adequate for undo, which makes use of logical operations that can only be applied to consistent objects. Section 5.6 describes a runtime latching and logging protocol that guarantees undo's logical operations only encounter consistent objects.

5.3 The log and page files

Log entries are identified by an LSN, $e.lsn$, and specify an operation over a particular object, $e.object$, or segment, $e.segment$. If the entry modifies a segment, it applies a physical (or, in the case of ARIES, physiological) operation; if not, it applies a logical operation.

Log entries are associated with a transaction, $e.tid$, which is a set of operations that should be applied to the database in an atomic, durable fashion. The state of the log also includes three special LSNs: log_t^{trunc}, the beginning of the sequence that is stored on disk; log_t^{stable}, the last entry stored on disk; and $log_t^{volatile}$, the most recent entry in memory.

5.4 Write-ahead and checkpointing

Write-ahead ensures that updates reach the log file before they reach the page file:

$$\forall \text{ segment } s : \exists\, l \in LSN(s_t^{stable}) : l \leq log_t^{stable} \tag{4}$$

Log truncation and checkpointing ensure that all current information can be reconstructed from disk:

$$\forall \text{ segment } s, \exists\, l \in LSN(s_t^{stable}) : l \geq log_t^{trunc} \tag{5}$$

which ensures that the version of each object stored on disk existed at some point during the range of LSNs covered by the log.[4] Our proposed recovery scheme weakens this slightly; $\forall s$ that violate Equation 4 or 5:
$\exists$ redo $e : e.lsn \in \{l : log_t^{trunc} \leq l \leq log_t^{stable}\}$:

$$e.lsn \in LSN(e(\top)) \tag{6}$$

Where $e(\top)$ is the result of applying e to a corrupt segment. This will be needed for hybrid recovery (Section 6.2).

5.5 Three-pass recovery

Recall that recovery performs three passes; the first, analysis, is an optimization that determines portions of the log may be safely ignored.

The second pass, redo, is modified by segment based recovery. In both systems, the contents of the buffer manager are lost at crash, so at the beginning of redo, t_0:

$$\forall \text{ segment } s : s_{t_0}^{current} = s_{t_0}^{stable}$$

It then applies redo entries in log order, repeating history, and bringing the system into a coherent but perhaps inconsistent state. This maintains the following invariant:

$$\forall \text{ segment } s, \exists\, l \in LSN(s_t^{current}) : l \geq log_cursor_t(s) \tag{7}$$

where $log_cursor_t(s)$ is an LSN associated with the segment in question. During redo, $log_cursor_t(s)$ monotonically increases from log_t^{trunc} to log_t^{stable}. Redo is parallelizable; each segment can be recovered independently. This allows online *media recovery*, which rebuilds corrupted pages by applying the redo log to a backed up copy of the database.

Redo assumes that the log is *complete*; $\forall$ segment s, lsn l,

$$s_{l-1} = s_l \vee (\exists\, e : e.lsn = l \wedge e.segment = s) \tag{8}$$

[4]For rollback to succeed, truncation must also avoid deleting entries from in-process transactions.

Either a segment is unchanged at a particular timestep, or there is a redo entry for that object at that timestep.

We now show that ARIES and segment-based recovery maintain the redo invariant (Equation 7). The hybrid approach is more complex and relies on allocation policies (Section 6.2).

5.5.1 ARIES redo strategy

ARIES applies a redo entry e with $l.lsn = log_cursor(s)$ to a segment $s = e.segment$ if:

$$e.lsn > s.lsn$$

ARIES is able to apply this strategy because it stores an LSN from $LSN(s)$ with each segment (which is also a fixed-length page); therefore, $s.lsn$ is defined. Assuming the redo log is complete, this policy maintains the redo invariant.

This redo strategy maintains the further invariant that, before it applies e, $e.lsn - 1 \in LSN(s)$; log entries are always applied to the same version of a segment.

5.5.2 Segment-based redo strategy

Our proposed algorithm always applies e. Since redo entries are *blind writes*, this yields an s such that $e.lsn \in LSN(s)$, regardless of the original value of the segment. Combined with completeness, this maintains the redo invariant.

5.5.3 Proof of redo's correctness

THEOREM 1. *At the end of redo, the database is coherent.*

PROOF. *From the definition of coherency (Equation 3), we need to show:*

$$\exists \text{ LSN } l : \forall \text{ object } O,\ l \in LSN(O)$$

By the definition of LSN(O) and an object, this is equivalent to:

$$\exists \text{ LSN } l : \forall \text{ segment } s \in O,\ l \in LSN(s)$$

Equations 4 and 7 ensure that:

$$\forall s, \exists\ l \in LSN(s) : log_t^{trunc} \leq log_cursor_t(s) \leq l \leq log_t^{stable}$$

At the end of redo, $\forall s,\ log_cursor_t(s) = l = log_t^{stable}$, *allowing us to reorder the universal and existential quantifiers.* □

The third phase of recovery, *undo* assumes that redo leaves the system in a coherent state. Since the database is coherent at the beginning of undo, we can treat transaction rollbacks during recovery in the same manner as rollbacks during forward operation. Next we prove rollback's correctness, concluding our treatment of recovery.

5.6 Transaction rollback

Multi-level recovery is compatible with concurrent transactions and allocation, even in the face of rollback. This section presents a special case of multi-level recovery: a simple, correct logging and latching scheme (Figure 7).

Like any other concurrent primitive, actions that manipulate transactional data temporarily break then restore various invariants as they execute. While such invariants are broken, other transactions must not observe the intermediate, inconsistent state.

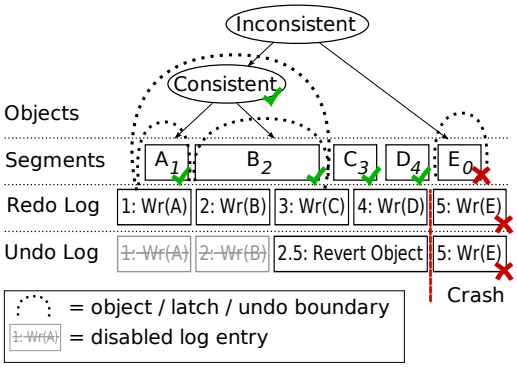


Figure 7: State of the system before undo; the data is coherent, but inconsistent. At runtime, updates hold each latch while manipulating the corresponding object, and release the latch when they log the undo. This ensures that undo entries never encounter inconsistent objects.

Recall that the definition of *coherent* (Equation 3) is based on nestings of recoverable objects. One approach to concurrent transactions obtains a latch on each object before modifying sub-objects, and then releases the latch before returning control to higher level operations. Establishing a partial ordering over the objects defines an ordering over the latches, guaranteeing that the system will not deadlock due to latch requests [13].

By construction, this scheme guarantees that all unlatched objects have no outstanding operations, and are therefore consistent. Atomically releasing latches and logging undo operations ties the undo to a point in time when the object was consistent; rollback ensures that undo operations will only be applied at such times. This latching scheme is more restrictive than necessary, but simplifies the implementation of logical operations [29]. More permissive approaches [20, 24] expose object state mid-operation.

The correctness of this scheme relies on the semantics of the undo operations. In particular, some are commutative (inserting x and y into a hashtable), while others are not ($z := 1$, $z := 2$). All operations from outstanding transactions must be commutative:

$\forall$ undo entry $e, f : e.tid \neq f.tid$,

$$o = e.object = f.object \Rightarrow e(f(o)) = f(e(o)) \qquad (9)$$

To support rollback, we log a logical undo for each higher level object update and a physical undo for each segment update. Each registration of a higher level undo *invalidates* lower level logical and physical undos, as does transaction commit. Invalidated undos are treated as though they no longer exist.[5] In addition to the truncation invariant for redo entries Equation 5, truncation waits for undo entries to be invalidated before deleting them. This is easily implemented by keeping track of the earliest LSN produced by ongoing transactions.

This, combined with our latching scheme guarantees that any violations of Equation 9 are due to two transactions directly invoking two non-commutative operations. This is a special case of *write-write conflicts* from the concurrency

[5] ARIES and segment-based recovery make use of logging mechanisms such as *nested top actions* and *compensation log records* to invalidate undo entries; we omit the details.

	Log preimage		Safety		Reuse before commit	
	Free	Alloc	LSN	Segment	Other xact	Same
1	Y		Y	Y	Y	Y
2		Y	Y	Y		Y
3		XOR	Y			Y
4	Never		Y	Y		

Figure 8: Allocation strategies.

control literature; in the absence of such conflicts, Equation 9 holds and the results of undo are unambiguous.

If we further assume that a concurrency control mechanism ensures the transactions are serializable, and if the undos are indeed the logical inverse of the corresponding forward operations, then rolling back a transaction places the system in a state logically equivalent to the one that would exist if the transaction were never initiated. This comes from the commutativity property in Equation 9.

Although concurrent data structure implementations are beyond the scope of this paper, there are two common approaches for dealing with lower-level conflicts. The first raises the level of abstraction before undoing an operation. For example, two transactions may update the same record while inserting different values into a B-tree. As each operation releases its latch, it logs an undo that will invoke the B-tree's "remove()" method instead of directly restoring the record. The second approach avoids lower-level conflicts. For example, some allocators guarantee space will not be reused until the transaction that freed the space commits.

6. ALLOCATION

The prior section treated allocation implicitly. A single object named the "database" spanned the entire page file, and allocation and deallocation were simply special operations over that object. In practice, recovery, allocation and concurrency control are tightly coupled. This section describes some possible approaches and identifies an efficient set that works with page- and segment-based recovery.

Transactional allocation algorithms must avoid unrecoverable states. In particular, reusing space or addresses that were freed by ongoing transactions leads to deadlock when those transactions rollback, as they attempt to reclaim the resources that they released. Unlike a deadlock in forward operation, deadlocks during rollback either halt the system or lead to cascading aborts.

Allocation consists of two sets of mechanisms. The first avoids unsafe conflicts by placing data appropriately and avoiding reuse of recently released resources. Data placement is a widely studied problem, though most discussions focus on performance. The second determines when data is written to log, ensuring that a copy of data freed by ongoing transactions exists somewhere in the system. Figure 8 summarizes four approaches.

The first two strategies log preimages, incurring the cost of extra logging; the fourth waits to reuse space until the transaction that freed the space commits. This makes it inappropriate for indexes and transactions that free space for immediate reuse.

The third option (labeled "XOR") refers to any *differential logging* [19] strategy that stores the new value as a function of the old value. Although differential updates and segment storage can coexist, differential page allocation is incompatible with our approach.

Differential logging was proposed as a way of increasing concurrency for main memory databases, and must apply log entries exactly once, but in any order. In contrast, our approach avoids the exactly once requirement, and is still able to parallelize redo (though to a lesser extent).

Logging preimages allows other transactions to overwrite the space that was taken up by the old object. This could happen due to page compaction, which consolidates free space on the page into a single region. Therefore, for pages that support reorganization, logging preimages at deallocation is the simplest approach.

For entire pages, or segments with unchanging boundaries, issues such as page compaction do not arise, so there is little reason to log at deallocation; instead a transaction can log preimages before reusing space it freed, or can avoid logging preimages altogether.

6.1 Existing hybrid allocation schemes

Recall that, without the benefit of per page version numbers, there is no way for redo to ensure that it is updating the correct version of a page. We could simply apply each redo entry in order, but there is no obvious way to decide whether or not a page contains an LSN. Inadvertently applying a redo to the wrong type of page corrupts the page.

Lotus Notes and Domino address the problem by recording synchronous page flushes and allocation events in the log, and adding extra passes to recovery [25]. The recovery passes ensure that page allocation information is coherent and matches the types of the pages that had made it to disk at crash. They extended this to multiple legacy allocation schemes and data types at the cost of great complexity [25].

Starburst records a table of current on-disk page maps in battery-backed RAM, skipping the extra recovery passes by keeping the appropriate state across crashes [4].

6.2 Correctness of hybrid redo

Here we prove Theorem 1 (redo's correctness) for hybrid ARIES and segment-based recovery. The hybrid allocator zeros out pages as they switch between LSN-free and segment-based formats. Also, page-oriented redo entries are only generated when the page contains an LSN, and segment-oriented redos are only generated when the page is LSN-free:

$$e.lsn_free \iff lsn_free(e.segment_{e.lsn}) \qquad (10)$$

THEOREM 2. *Hybrid redo leaves the database in a coherent state*

PROOF. *Equations 4 and 5 tell us each segment is coherent at the beginning of recovery. Although* $lsn_free(s)$ *or* $\neg lsn_free(s)$ *must be true, redo cannot distinguish between these two cases, and simply assumes the page starts in the format it was in when the beginning of the redo log was written.*

In the first case, this assumption is correct and redo will continue as normal for the pure LSN or LSN-free recovery algorithm. It will eventually complete or reach an entry that changes the page format, causing it to switch to the other redo algorithm. By the correctness of pure LSN and LSN-free redo (Section 5.5) this will maintain the invariant in Equation 7 until it completes.

In the second case, the assumption is incorrect. By Equation 10, the stable version of the page must have a different

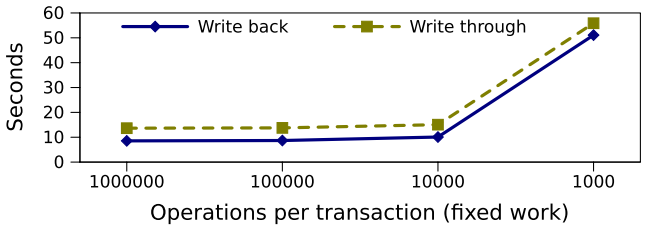


Figure 9: Time taken to transactionally update 10,000,000 `int` values. Write back reduces CPU overhead.

type than it did when the redo entry was generated. Nevertheless, redo applies all log entries to the page, temporarily corrupting it. The write-ahead and truncation invariants, and log completeness (Equations 4, 5, and 8) guarantee that the log entry that changed the page's format is in the redo log. Once this entry, e, is encountered, it zeros out the page, repairing the corruption and ensuring that $e.lsn \in LSN(s)$*, (Equation 6). At this point, the page format matches the current log entry, reducing this to the first case.* □

7. DISCUSSION AND EVALUATION

Our experiments were run on an AMD Athlon 64 Processor 3000+ with a 1TB Samsung HD103UJ with write caching disabled, running Linux 2.6.27, and Stasis r1156.

7.1 Zero-copy I/O

Most large object schemes avoid writing data to log, and instead force-write data to pages at commit. Since the pages contain the only copy of the data in the system, applying blind writes to them would corrupt application data. Instead, we augment recovery's analysis pass, which already infers that certain pages are up-to-date. When a segment is allocated for force-writes, analysis adds it to a known-updated list, and removes it when the segment is freed.

This means that analysis' list of known-updated pages is now required for correctness, and must be guaranteed to fit in memory. Fortunately, redo can be performed on a per segment basis; if the list becomes too large, we partition the database, then perform an independent analysis and redo pass for each partition.

Zero-copy I/O complicates buffer management. If it is desirable to bypass the buffer manager's cache, then zero-copy writes must invalidate cached pages. If not, then the zero-copy primitives must be compatible with the buffer managers' memory layout. Once the necessary changes to recovery and buffer management are made, we expect the performance of large zero-copy writes to match that of existing file servers; increased file sizes decrease the relative cost of maintaining metadata.

7.2 Write caching

Read caching is a fairly common approach, both in local and distributed [10] architectures. However, distributed, durable write caching is more difficult and we are not aware of any commonly used systems.

Instead, each time an object is updated, it is marshaled then atomically (and synchronously) sent to the storage layer and copied to log and the buffer pool. This approach wastes both memory and time [29]. Even with minimal marshaling overheads, locating then pinning a page from the buffer manager decreases memory locality and incurs extra synchronization costs across CPUs.

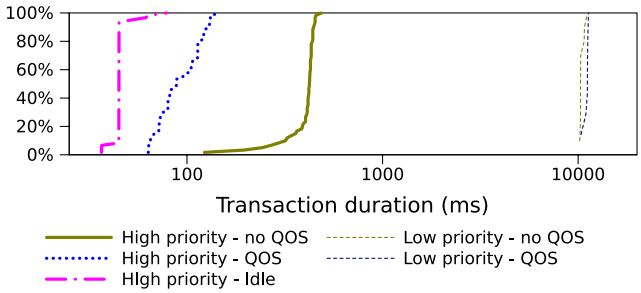


Figure 10: CDF of transaction completion times with and without log reordering.

To measure these costs, we extended Stasis with support for segments within pages, and removed LSNs from the header of such pages. We then built a simple application cache. To perform an LSN-free write, we append redo/undo entries to log, then update the application cache, causing the buffer manager to become incoherent. Before shutdown, we write back the contents of cache to the buffer manager.

To perform conventional write through, we do not set up the cache and instead call Stasis' existing record set method.

Because the buffer manager is incoherent, our optimization provides *no-force* between the application cache and buffer manager. In contrast, applications built on ARIES force data to the buffer pool at each update instead of once at shutdown. This increases CPU costs substantially.

The effects of extra buffer management overhead are noticeable even in the single-threaded case; Figure 9 compares the cost of durably updating 10,000,000 integers using transactions of varying size.

For small transactions, (less than about 10,000 updates) the cost of force writing the log at commit dominates performance. For larger transactions, most of the time is spent on asynchronous log writes and on buffer manager operations. We expect the gap between write back and write through to be higher in systems that marshal objects (instead of raw integers), and in systems with greater log bandwidth.

7.3 Quality of service

We again extend Stasis, this time allowing each transaction to optionally register a low-priority queue for its segment updates. To perform a write, transactions pin and update the page, then submit the log entry to the queue. As the queue writes back log entries, it unpins pages. We use these primitives to implement a simple quality of service mechanism. The disk supports a fixed number of synchronous writes per second, and Stasis maintains a log buffer in memory. Low priority transactions ensure that a fraction of Stasis' write queue is unused, reserving space for high-priority transactions. A subtle, but important detail of this scheme is that, because transactions unlatch pages before appending data to log, backpressure from the logger decreases page latch contention; page-based systems hold latches across log operations, leading to increased contention and lower read throughput.

For our experiment, we run "low priority" bulk transactions that continuously update records with no delay, and "high priority" transactions that only update a single record,

Storage algorithm	Small workload		Large workload	
	Local	Network	Local	Network
Pages	0.866s	61s	10.86s	6254s
Segments	0.820s	26s	5.893s	105s
Segs. (bulk messages)	”	8s	”	13s

Figure 11: Comparison of segment and page based recovery with simulated network latency. The small workload runs ten transactions of 1000 updates each; the large workload runs ten of 100,000 each.

but run once a second. This simulates a high-throughput bulk load running in parallel with low-latency application requests.

Figure 10 plots the cumulative distribution function of the transactions' response times. With log reordering (QOS) in place, worst case response time for high priority transactions is approximately 140ms; "idle" reports high priority transaction performance without background tasks.

7.4 Recovery for distributed systems

Data center and cloud computing architectures are often provisioned in terms of *applications*, *cache*, *storage* and *reliable queues*. Though their implementations are complex, highly available approaches with linear scalability are available for each service.

However, scaling these primitives is expensive, and operations against these systems are often heavy-weight, leading to poor response times and poor utilization of hardware. Write reordering and write caching help address these bottlenecks. For our evaluation, we focused on reordering requests to write to the log and writing-back updates to the buffer manager.

We modified Stasis with the intention of simulating a network environment. We add $2ms$ delays to each request to append data to Stasis' log buffer, or to read or write records in the buffer manager. We did not simulate the overhead of communicating LSN estimates between the log and page storage nodes. We ran our experiment with both write back and write reordering enabled (Figure 11), running one transaction at a time. For the "bulk messages" experiments, we batch requests rather than send one per network round trip.

For small transactions, the networked version is roughly ten times slower than the local versions, but approximately 20 times faster than a distributed, page-oriented approach. As transaction sizes increase, segment-based recovery is better able to amortize network round trips due to log and buffer manager requests, and network throughput improves to more than 400 times that of the page-based approach. As above, the local versions of these benchmarks are competitive with local page-oriented approaches, especially for long transactions.

A true distributed implementation would introduce additional overheads and opportunities for improved scalability. In particular, replication will allow the components to cope with partial failure and partitioning should provide linear scalability within each component. How such an approach interacts with real-world workloads is an open question.

As with any other distributed system, there will be tradeoffs between consistency and performance; we suspect that durability based upon distributed write-ahead logging will provide significantly greater performance and flexibility than systems based on synchronous updates of replicas.

8. RELATED WORK

Here, we focus on other approaches to the problems we address. First we discuss systems with support for log reordering, then we discuss distributed write-ahead logging.

Write reordering mechanisms provide the most benefit in systems with long running, non-durably committed requests. Therefore, most related work in this area comes from the filesystem community. Among filesystems, our design is perhaps most similar to Echo [23]. Its *write-behind* queues provide rich write reordering semantics and are a non-durable version of our reorderable write-ahead logs.

FeatherStitch [12] introduces filesystem *patches*; sets of atomic block writes (blind writes) with ordering constraints, and allows the block scheduler and applications to reorder patches. Rather than provide concurrent transactions, it provides filesystem semantics and a `pg_sync` mechanism that explicitly force-writes a patch and its dependencies to disk.

Although our distributed performance results are promising, designing a complete, scalable and fault tolerant storage system from our algorithm is non-trivial. Fortunately, the implementation of each component in our design is well understood. Read only caching technologies such as memcached [10] would provide a good starting point for linearly scalable write back application caches. Main-memory database techniques are increasingly sophisticated, and support compression, superscalar optimizations, and isolation.

Scalable data storage is also widely studied. Cluster hash tables [11], which partition data across independent index nodes, and Boxwood [22], which distributes indexes across clusters, are two extreme points in the scope of possible designs. A third approach, Sinfonia [1], has nodes expose a linear address space, then performs *minitransactions*; essentially atomic bundles of test and set operations against these nodes. In contrast, page writeback allows us to apply many transactions to the storage nodes with a single network round trip, but relies on a logging service.

A number of reliable log services are already available, including ones that scale up to data center and Internet scale workloads. In the context of cloud computing, indexes such as B-Trees have been implemented on top of Amazon SQS (a scalable, reliable log) and S3 (a scalable record store) using purely logical redo and undo; those approaches require write-ahead logging or other recovery mechanisms at each storage node [3]. Application specific systems also exist, and handle atomicity in the face of unreliable clients [27].

A second cloud computing approach is extremely similar to our distributed proposal, but handles concurrency and reordering with explicit per object LSNs and exactly-once redo [21]. Replicas store objects in durable key-value stores that are backed by a second, local, recovery mechanism. An additional set of mechanisms ensures that recovery's redo phase is idempotent. In contrast, implementing idempotent redo is straightforward in segment-based systems.

9. CONCLUSION

Segment-based recovery operates at the granularity of application requests, removing LSNs from pages. It brings request reordering and reduced communication costs to concurrent, steal/no-force database recovery algorithms. We presented ARIES-style and segment-based recovery in terms of the invariants they maintain, leading to a simple proof of their correctness.

The results of our experiments suggest segment-based recovery significantly improves performance, particularly for transactions run alongside application caches, run with different priorities, or run across large-scale distributed systems. We have not yet built practical segment-based storage. However, we are currently building a number of systems based on the ideas presented here.

10. ACKNOWLEDGMENTS

Special thanks to our shepherd, Alan Fekete for his help correcting and greatly improving the presentation of this work.

We would also like to thank Peter Alvaro, Brian Frank Cooper, Tyson Condie, Joe Hellerstein, Akshay Krishnamurthy, Blaine Nelson, Rick Spillane and the anonymous reviewers for suggestions regarding earlier drafts this paper.

Our discussions with Phil Bohannon, Catharine van Ingen, Jim Gray, C. Mohan, P.P.S. Narayan, Mehul Shah and David Wu clarified these ideas, and brought existing approaches and open challenges to our attention.

11. REFERENCES

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. 1987.

[3] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, 2008.

[4] L. Cabrera, J. McPherson, P. Schwarz, and J. Wyllie. Implementing atomicity in two systems: Techniques, tradeoffs, and experience. *TOSE*, 19(10), 1993.

[5] D. Chamberlin et al. A history and evaluation of system R. *CACM*, 24(10), 1981.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[7] R. A. Crus. Data recovery in IBM Database 2. *IBM Systems Journal*, 23(2), 1984.

[8] P. A. Dearnley. An investigation into database resilience. *Oxford Computer Journal*, July 1975.

[9] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *SOSP*, 1993.

[10] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, August 2004.

[11] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *SOSP*, 1997.

[12] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and Lei. Generalized file system dependencies. In *SOSP*, 2007.

[13] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. *Modelling in Data Base Management Systems*, pages 365–r394. North-Holland, Amsterdam, 1976.

[14] T. Greanier. Serialization API. In *JavaWorld*, 2000.

[15] T. Haerder and A. Reuter. Principles of transaction oriented database recovery—a taxonomy. *ACM Computing Surveys*, 1983.

[16] Hibernate. http://www.hibernate.org/.

[17] D. Kuo. Model and verification of a data manager based on ARIES. *TODS*, 21(4), 1996.

[18] L. Lamport. Paxos made simple. *SIGACT News*, 2001.

[19] J. Lee, K. Kim, and S. Cha. Differential logging: A commutative and associative logging scheme for highly parallel main memory databases. In *ICDE*, 2001.

[20] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *TODS*, 1981.

[21] D. Lomet, A. Fekete, G. Weikum, and M. Zwilling. Unbundling transaction services in the cloud. In *CIDR*, 2009.

[22] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.

[23] T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart. A coherent distributed file cache with directory write-behind. *TOCS*, May 1994.

[24] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control multiaction transactions operating on B-tree indexes. In *VLDB*, 1990.

[25] C. Mohan. A database perspective on Lotus Domino/Notes. In *SIGMOD Tutorial*, 1999.

[26] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1):94–162, 1992.

[27] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer. Making a cloud provenance-aware. In *TAPP*, 2009.

[28] E. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the sync. In *OSDI*, 2006.

[29] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.

[30] M. Seltzer and M. Olsen. LIBTP: Portable, modular transactions for UNIX. In *Usenix*, January 1992.

[31] *SQL Server 2008 Documetation*, chapter Buffer Management. Microsoft, 2009.

[32] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for Unix. Technical Report SMLI TR-95-39, Sun Microsystems, 1995.

[33] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multi-level recovery. In *PODS*, 1990.

[34] M. Widenius and D. Axmark. *MySQL Manual*.

Lightweight Recoverable Virtual Memory

M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, James J. Kistler

School of Computer Science
Carnegie Mellon University

Abstract

Recoverable virtual memory refers to regions of a virtual address space on which transactional guarantees are offered. This paper describes *RVM*, an efficient, portable, and easily used implementation of recoverable virtual memory for Unix environments. A unique characteristic of RVM is that it allows independent control over the transactional properties of atomicity, permanence, and serializability. This leads to considerable flexibility in the use of RVM, potentially enlarging the range of applications than can benefit from transactions. It also simplifies the layering of functionality such as nesting and distribution. The paper shows that RVM performs well over its intended range of usage even though it does not benefit from specialized operating system support. It also demonstrates the importance of intra- and inter-transaction optimizations.

1. Introduction

How simple can a transactional facility be, while remaining a potent tool for fault-tolerance? Our answer, as elaborated in this paper, is a user-level library with minimal programming constraints, implemented in about 10K lines of mainline code and no more intrusive than a typical runtime library for input-output. This transactional facility, called *RVM*, is implemented without specialized operating system support, and has been in use for over two years on a wide range of hardware from laptops to servers.

RVM is intended for Unix applications with persistent data structures that must be updated in a fault-tolerant manner. The total size of those data structures should be a small fraction of disk capacity, and their working set size must easily fit within main memory.

This work was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio, 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. James Kistler is now affiliated with the DEC Systems Research Center, Palo Alto, CA.

This paper appeared in ACM Transactions on Computer Systems, 12(1), Feb. 1994 and Proceedings of the 14th ACM Symposium on Operating Systems Principles, Dec. 1993.

This combination of circumstances is most likely to be found in situations involving the meta-data of storage repositories. Thus RVM can benefit a wide range of applications from distributed file systems and databases, to object-oriented repositories, CAD tools, and CASE tools. RVM can also provide runtime support for persistent programming languages. Since RVM allows independent control over the basic transactional properties of atomicity, permanence, and serializability, applications have considerable flexibility in how they use transactions.

It may often be tempting, and sometimes unavoidable, to use a mechanism that is richer in functionality or better integrated with the operating system. But our experience has been that such sophistication comes at the cost of portability, ease of use and more onerous programming constraints. Thus RVM represents a balance between the system-level concerns of *functionality* and *performance*, and the software engineering concerns of *usability* and *maintenance*. Alternatively, one can view RVM as an exercise in minimalism. Our design challenge lay not in conjuring up features to add, but in determining what could be omitted without crippling RVM.

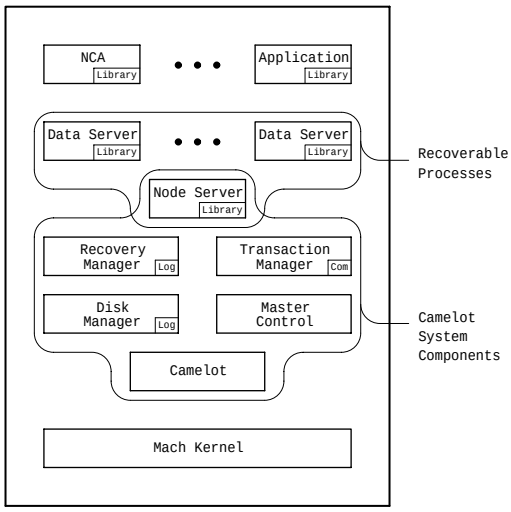
We begin this paper by describing our experience with Camelot [10], a predecessor of RVM. This experience, and our understanding of the fault-tolerance requirements of Coda [16, 30] and Venari [24, 37], were the dominant influences on our design. The description of RVM follows in three parts: rationale, architecture, and implementation. Wherever appropriate, we point out ways in which usage experience influenced our design. We conclude with an evaluation of RVM, a discussion of its use as a building block, and a summary of related work.

2. Lessons from Camelot

2.1. Overview

Camelot is a transactional facility built to validate the thesis that general-purpose transactional support would simplify and encourage the construction of reliable distributed systems [33]. It supports local and distributed nested transactions, and provides considerable flexibility in the choice of logging, synchronization, and transaction

commitment strategies. Camelot relies heavily on the external page management and interprocess communication facilities of the Mach operating system [2], which is binary compatible with the 4.3BSD Unix operating system [20]. Figure 1 shows the overall structure of a Camelot node. Each module is implemented as a Mach task and communication between modules is via Mach's interprocess communication facililty(IPC).



This figure shows the internal structure of Camelot as well as its relationship to application code. Camelot is composed of several Mach tasks: Master Control, Camelot, and Node Server, as well as the Recovery, Transaction, and Disk Managers. Camelot provides recoverable virtual memory for Data Servers; that is, transactional operations are supported on portions of the virtual address space of each Data Server. Application code can be split between Data Server and Application tasks (as in this figure), or may be entirely linked into a Data Server's address space. The latter approach was used in Coda. Camelot facilities are accessed via a library linked with application code.

Figure 1: Structure of a Camelot Node

2.2. Usage

Our interest in Camelot arose in the context of the two-phase optimistic replication protocol used by the Coda File System. Although the protocol does not require a distributed commit, it does require each server to ensure the atomicity and permanence of local updates to meta-data in the first phase. The simplest strategy for us would have been to implement an *ad hoc* fault tolerance mechanism for meta-data using some form of shadowing. But we were curious to see what Camelot could do for us.

The aspect of Camelot that we found most useful is its support for *recoverable virtual memory* [9]. This unique feature of Camelot enables regions of a process' virtual address space to be endowed with the transactional properties of atomicity, isolation and permanence. Since we did not find a need for features such as nested or distributed transactions, we realized that our use of Camelot would be something of an overkill. Yet we persisted, because it would give us first-hand experience in the use of transactions, and because it would contribute towards the validation of the Camelot thesis.

We placed data structures pertaining to Coda meta-data in recoverable memory[1] on servers. The meta-data included Coda directories as well as persistent data for replica control and internal housekeeping. The contents of each Coda file was kept in a Unix file on a server's local file system. Server recovery consisted of Camelot restoring recoverable memory to the last committed state, followed by a Coda *salvager* which ensured mutual consistency between meta-data and data.

2.3. Experience

The most valuable lesson we learned by using Camelot was that recoverable virtual memory was indeed a convenient and practically useful programming abstraction for systems like Coda. Crash recovery was simplified because data structures were restored *in situ* by Camelot. Directory operations were merely manipulations of in-memory data structures. The Coda salvager was simple because the range of error states it had to handle was small. Overall, the encapsulation of messy crash recovery details into Camelot considerably simplified Coda server code.

Unfortunately, these benefits came at a high price. The problems we encountered manifested themselves as *poor scalability*, *programming constraints*, and *difficulty of maintenance*. In spite of considerable effort, we were not able to circumvent these problems. Since they were direct consequences of the design of Camelot, we elaborate on these problems in the following paragraphs.

A key design goal of Coda was to preserve the scalability of AFS. But a set of carefully controlled experiments (described in an earlier paper [30]) showed that Coda was less scalable than AFS. These experiments also showed that the primary contributor to loss of scalability was increased server CPU utilization, and that Camelot was responsible for over a third of this increase. Examination of Coda servers in operation showed considerable paging and context switching overheads due to the fact that each Camelot operation involved interactions between many of the component processes shown in Figure 1. There was no obvious way to reduce this overhead, since it was inherent in the implementation structure of Camelot.

[1]For brevity, we often omit "virtual" from "recoverable virtual memory" in the rest of this paper.

A second obstacle to using Camelot was the set of programming constraints it imposed. These constraints came in a variety of guises. For example, Camelot required all processes using it to be descendants of the Disk Manager task shown in Figure 1. This meant that starting Coda servers required a rather convoluted procedure that made our system administration scripts complicated and fragile. It also made debugging more difficult because starting a Coda server under a debugger was complex. Another example of a programming constraint was that Camelot required us to use Mach kernel threads, even though Coda was capable of using user-level threads. Since kernel thread context switches were much more expensive, we ended up paying a hefty peformance cost with little to show for it.

A third limitation of Camelot was that its code size, complexity and tight dependence on rarely used combinations of Mach features made maintenance and porting difficult. Since Coda was the sternest test case for recoverable memory, we were usually the first to expose new bugs in Camelot. But it was often hard to decide whether a particular problem lay in Camelot or Mach.

As the cumulative toll of these problems mounted, we looked for ways to preserve the virtues of Camelot while avoiding its drawbacks. Since recoverable virtual memory was the only aspect of Camelot we relied on, we sought to distill the essence of this functionality into a realization that was cheap, easy-to-use and had few strings attached. That quest led to RVM.

3. Design Rationale

The central principle we adopted in designing RVM was to value *simplicity* over *generality*. In building a tool that did one thing well, we were heeding Lampson's sound advice on interface design [19]. We were also being faithful to the long Unix tradition of keeping building blocks simple. The change in focus from generality to simplicity allowed us to take radically different positions from Camelot in the areas of *functionality*, *operating system dependence*, and *structure*.

3.1. Functionality

Our first simplification was to eliminate support for *nesting* and *distribution*. A cost-benefit analysis showed us that each could be better provided as an independent layer on top of RVM[2]. While a layered implementation may be less efficient than a monolithic one, it has the attractive property of keeping each layer simple. Upper layers can count on the clean failure semantics of RVM, while the latter is only responsible for local, non-nested transactions.

A second area where we have simplified RVM is *concurrency control*. Rather than having RVM insist on a specific technique, we decided to factor out concurrency control. This allows applications to use a policy of their choice, and to perform synchronization at a granularity appropriate to the abstractions they are supporting. If serializability is required, a layer above RVM has to enforce it. That layer is also responsible for coping with deadlocks, starvation and other unpleasant concurrency control problems.

Internally, RVM is implemented to be multi-threaded and to function correctly in the presence of true parallelism. But it does not depend on kernel thread support, and can be used with no changes on user-level thread implementations. We have, in fact, used RVM with three different threading mechanisms: Mach kernel threads [8], coroutine C threads, and coroutine LWP [29].

Our final simplification was to factor out resiliency to media failure. Standard techniques such as mirroring can be used to achieve such resiliency. Our expectation is that this functionality will most likely be implemented in the device driver of a mirrored disk.

RVM thus adopts a layered approach to transactional support, as shown in Figure 2. This approach is simple and enhances flexibility: an application does not have to buy into those aspects of the transactional concept that are irrelevant to it.

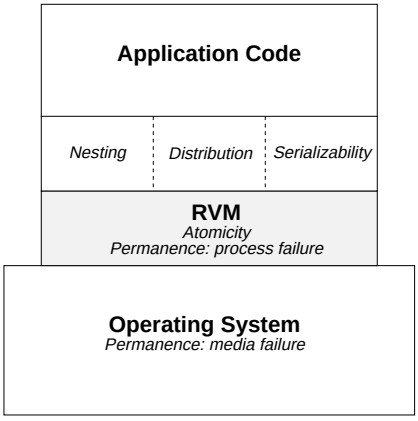


Figure 2: Layering of Functionality in RVM

3.2. Operating System Dependence

To make RVM portable, we decided to rely only on a small, widely supported, Unix subset of the Mach system call interface. A consequence of this decision was that we could not count on tight coupling between RVM and the VM subsystem. The Camelot Disk Manager module runs

[2]An implementation sketch is provided in Section 8.

as an *external pager* [39] and takes full responsibility for managing the backing store for recoverable regions of a process. The use of advisory VM calls (`pin` and `unpin`) in the Mach interface lets Camelot ensure that dirty recoverable regions of a process' address space are not paged out until transaction commit. This close alliance with Mach's VM subsystem allows Camelot to avoid double paging, and to support recoverable regions whose size approaches backing store or addressing limits. Efficient handling of large recoverable regions is critical to Camelot's goals.

Our goals in building RVM were more modest. We were not trying to replace traditional forms of persistent storage, such as file systems and databases. Rather, we saw RVM as a building block for meta-data in those systems, and in higher-level compositions of them. Consequently, we could assume that the recoverable memory requirements on a machine would only be a small fraction of its total disk storage. This in turn meant that it was acceptable to waste some disk space by duplicating the backing store for recoverable regions. Hence RVM's backing store for a recoverable region, called its *external data segment*, is completely independent of the region's VM swap space. Crash recovery relies only on the state of the external data segment. Since a VM pageout does not modify the external data segment, an uncommitted dirty page can be reclaimed by the VM subsystem without loss of correctness. Of course, good performance also requires that such pageouts be rare.

One way to characterize our strategy is to view it as a *complexity* versus *resource usage* tradeoff. By being generous with memory and disk space, we have been able to keep RVM simple and portable. Our design supports the optional use of external pagers, but we have not implemented support for this feature yet. The most apparent impact on Coda has been slower startup because a process' recoverable memory must be read in *en masse* rather than being paged in on demand.

Insulating RVM from the VM subsystem also hinders the sharing of recoverable virtual memory across address spaces. But this is not a serious limitation. After all, the primary reason to use a separate address space is to increase robustness by avoiding memory corruption. Sharing recoverable memory across address spaces defeats this purpose. In fact, it is worse than sharing (volatile) virtual memory because damage may be persistent! Hence, our view is that processes willing to share recoverable memory already trust each other enough to run as threads in a single address space.

3.3. Structure

The ability to communicate efficiently across address spaces allows robustness to be enhanced without sacrificing good performance. Camelot's modular decomposition, shown earlier in Figure 1, is predicated on fast IPC. Although it has been shown that IPC can be fast [4], its performance in commercial Unix implementations lags far behind that of the best experimental implementations. Even on Mach 2.5, the measurements reported by Stout et al [34] indicate that IPC is about 600 times more expensive than local procedure call[3]. To make matters worse, Ousterhout [26] reports that the context switching performance of operating systems is not improving linearly with raw hardware performance.

Given our desire to make RVM portable, we were not willing to make its design critically dependent on fast IPC. Instead, we have structured RVM as a library that is linked in with an application. No external communication of any kind is involved in the servicing of RVM calls. An implication of this is, of course, that we have to trust applications not to damage RVM data structures and vice versa.

A less obvious implication is that applications cannot share a single write-ahead log on a dedicated disk. Such sharing is common in transactional systems because disk head movement is a strong determinant of performance, and because the use of a separate disk per application is economically infeasible at present. In Camelot, for example, the Disk Manager serves as the multiplexing agent for the log. The inability to share one log is not a significant limitation for Coda, because we run only one file server process on a machine. But it may be a legitimate concern for other applications that wish to use RVM. Fortunately, there are two potential alleviating factors on the horizon.

First, independent of transaction processing considerations, there is considerable interest in log-structured implementations of the Unix file system [28]. If one were to place the RVM log for each application in a separate file on such a system, one would benefit from minimal disk head movement. No log multiplexor would be needed, because that role would be played by the file system.

Second, there is a trend toward using disks of small form factor, partly motivated by interest in disk array technology [27]. It has been predicted that the large disk capacity in the future will be achieved by using many small

[3] 430 microseconds versus 0.7 microseconds for a null call on a typical contemporary machine, the DECStation 5000/200

disks. If this turns out to be true, there will be considerably less economic incentive to avoiding a dedicated disk per process.

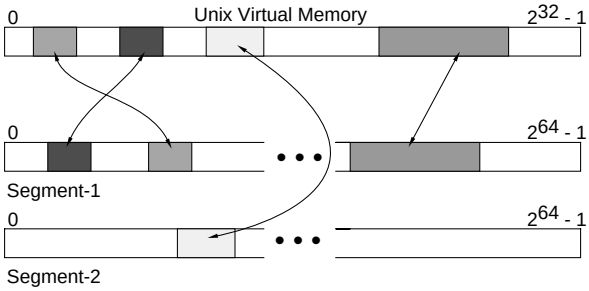
In summary, each process using RVM has a separate log. The log can be placed in a Unix file or on a raw disk partition. When the log is on a file, RVM uses the `fsync` system call to synchronously flush modifications onto disk. RVM's permanence guarantees rely on the correct implementation of this system call. For best performance, the log should either be in a raw partition on a dedicated disk or in a file on a log-structured Unix file system.

4. Architecture

The design of RVM follows logically from the rationale presented earlier. In the description below, we first present the major program-visible abstractions, and then describe the operations supported on them.

4.1. Segments and Regions

Recoverable memory is managed in *segments*, which are loosely analogous to Multics segments. RVM has been designed to accomodate segments up to 2^{64} bytes long, although current hardware and file system limitations restrict segment length to 2^{32} bytes. The number of segments on a machine is only limited by its storage resources. The backing store for a segment may be a file or a raw disk partition. Since the distinction is invisible to programs, we use the term ''external data segment'' to refer to either.



Each shaded area represents a region. The contents of a region are physically copied from its external data segment to the virtual memory address range specified during mapping.

Figure 3: Mapping Regions of Segments

As shown in Figure 3, applications explicitly map *regions* of segments into their virtual memory. RVM guarantees that newly mapped data represents the committed image of the region. A region typically corresponds to a related collection of objects, and may be as large as the entire segment. In the current implementation, the copying of data from external data segment to virtual memory occurs when a region is mapped. The limitation of this method is startup latency, as mentioned in Section 3.2. In the future, we plan to provide an optional Mach external pager to copy data on demand.

Restrictions on segment mapping are minimal. The most important restriction is that no region of a segment may be mapped more than once by the same process. Also, mappings cannot overlap in virtual memory. These restrictions eliminate the need for RVM to cope with aliasing. Mapping must be done in multiples of page size, and regions must be page-aligned.

Regions can be unmapped at any time, as long as they have no uncommitted transactions outstanding. RVM retains no information about a segment's mappings after its regions are unmapped. A *segment loader* package, built on top of RVM, allows the creation and maintenance of a load map for recoverable storage and takes care of mapping a segment into the same base address each time. This simplifies the use of absolute pointers in segments. A *recoverable memory allocator*, also layered on RVM, supports heap management of storage within a segment.

4.2. RVM Primitives

The operations provided by RVM for initialization, termination and segment mapping are shown in Figure 4(a). The log to be used by a process is specified at RVM initialization via the `options_desc` argument. The `map` operation is called once for each region to be mapped. The external data segment and the range of virtual memory addresses for the mapping are identified in the first argument. The `unmap` operation can be invoked at any time that a region is quiescent. Once unmapped, a region can be remapped to some other part of the process' address space.

After a region has been mapped, memory addresses within it may be used in the transactional operations shown in Figure 4(b). The `begin_transaction` operation returns a transaction identifier, `tid`, that is used in all further operations associated with that transaction. The `set_range` operation lets RVM know that a certain area of a region is about to be modified. This allows RVM to record the current value of the area so that it can undo changes in case of an abort. The `restore_mode` flag to `begin_transaction` lets an application indicate that it will never explicitly abort a transaction. Such a *no-restore* transaction is more efficient, since RVM does not have to copy data on a `set-range`. Read operations on mapped regions require no RVM intervention.

```
initialize(version, options_desc);
map(region_desc, options_desc);
unmap(region_desc);
terminate();
```

(a) Initialization & Mapping Operations

```
begin_transaction(tid, restore_mode);
set_range(tid, base_addr, nbytes);
end_transaction(tid, commit_mode);
abort_transaction(tid);
```

(b) Transactional Operations

```
flush();
truncate();
```

(c) Log Control Operations

```
query(options_desc, region_desc);
set_options(options_desc);
create_log(options, log_len, mode);
```

(d) Miscellaneous Operations

Figure 4: RVM Primitives

A transaction is committed by `end_transaction` and aborted via `abort_transaction`. By default, a successful commit guarantees permanence of changes made in a transaction. But an application can indicate its willingness to accept a weaker permanence guarantee via the `commit_mode` parameter of `end_transaction`. Such a *no-flush* or ''lazy'' transaction has reduced commit latency since a log force is avoided. To ensure persistence of its no-flush transactions the application must explicitly flush RVM's write-ahead log from time to time. When used in this manner, RVM provides *bounded persistence*, where the bound is the period between log flushes. Note that atomicity is guaranteed independent of permanence.

Figure 4(c) shows the two operations provided by RVM for controlling the use of the write-ahead log. The first operation, `flush`, blocks until all committed no-flush transactions have been forced to disk. The second operation, `truncate`, blocks until all committed changes in the write-ahead log have been reflected to external data segments. Log truncation is usually performed transparently in the background by RVM. But since this is a potentially long-running and resource-intensive operation, we have provided a mechanism for applications to control its timing.

The final set of primitives, shown in Figure 4(d), perform a variety of functions. The `query` operation allows an application to obtain information such as the number and identity of uncommited transactions in a region. The `set_options` operation sets a variety of tuning knobs such as the threshold for triggering log truncation and the sizes of internal buffers. Using `create_log`, an application can dynamically create a write-ahead log and then use it in an `initialize` operation.

5. Implementation

Since RVM draws upon well-known techniques for building transactional systems, we restrict our discussion here to two important aspects of its implementation: *log management* and *optimization*. The RVM manual [22] offers many further details, and a comprehensive treatment of transactional implementation techniques can be found in Gray and Reuter's text [14].

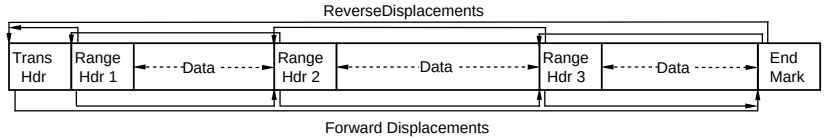
5.1. Log Management

5.1.1. Log Format

RVM is able to use a *no-undo/redo* value logging strategy [3] because it never reflects uncommitted changes to an external data segment. The implementation assumes that adequate buffer space is available in virtual memory for the old-value records of uncommitted transactions. Consequently, only the new-value records of committed transactions have to be written to the log. The format of a typical log record is shown in Figure 5.

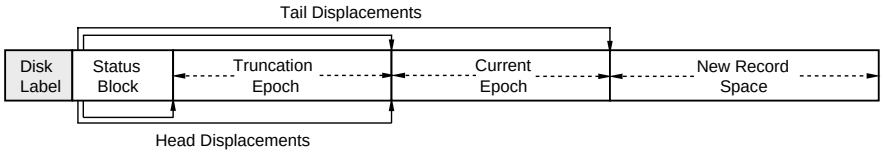
The bounds and contents of old-value records are known to RVM from the `set-range` operations issued during a transaction. Upon commit, old-value records are replaced by new-value records that reflect the current contents of the corresponding ranges of memory. Note that each modified range results in only one new-value record even if that range has been updated many times in a transaction. The final step of transaction commitment consists of forcing the new-value records to the log and writing out a commit record.

No-restore and no-flush transactions are more efficient. The former result in both time and space spacings since the contents of old-value records do not have to be copied or buffered. The latter result in considerably lower commit latency, since new-value and commit records can be spooled rather than forced to the log.



This log record has three modification ranges. The bidirectional displacements records allow the log to be read either way.

Figure 5: Format of a Typical Log Record



This figure shows the organization of a log during epoch truncation. The current tail of the log is to the right of the area marked "current epoch". The log wraps around logically, and internal synchronization in RVM allows forward processing in the current epoch while truncation is in progress. When truncation is complete, the area marked "truncation epoch" will be freed for new log records.

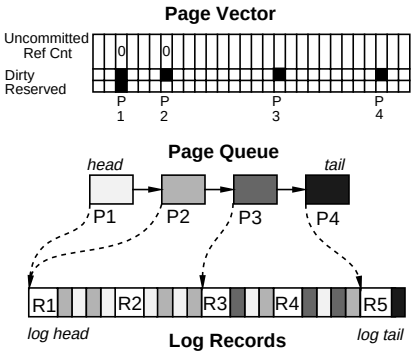
Figure 6: Epoch Truncation

5.1.2. Crash Recovery and Log Truncation

Crash recovery consists of RVM first reading the log from tail to head, then constructing an in-memory tree of the latest committed changes for each data segment encountered in the log. The trees are then traversed, applying modifications in them to the corresponding external data segment. Finally, the head and tail location information in the log status block is updated to reflect an empty log. The idempotency of recovery is achieved by delaying this step until all other recovery actions are complete.

Truncation is the process of reclaiming space allocated to log entries by applying the changes contained in them to the recoverable data segment. Periodic truncation is necessary because log space is finite, and is triggered whenever current log size exceeds a preset fraction of its total size. In our experience, log truncation has proved to be the hardest part of RVM to implement correctly. To minimize implementation effort, we initially chose to reuse crash recovery code for truncation. In this approach, referred to as *epoch truncation*, the crash recovery procedure described above is applied to an initial part of the log while concurrent forward processing occurs in the rest of the log. Figure 6 depicts the layout of a log while an epoch truncation is in progress.

Although exclusive reliance on epoch truncation is a logically correct strategy, it substantially increases log traffic, degrades forward processing more than necessary, and results in bursty system performance. Now that RVM is stable and robust, we are implementing a mechanism for *incremental truncation* during normal operation. This mechanism periodically renders the oldest log entries obsolete by writing out relevant pages directly from VM to the recoverable data segment. To preserve the no-undo/redo property of the log, pages that have been modified by uncommitted transactions cannot be written out to the recoverable data segment. RVM maintains internal locks to ensure that incremental truncation does not violate this property. Certain situations, such as the presence of long-running transactions or sustained high concurrency, may result in incremental truncation being blocked for so long that log space becomes critical. Under those circumstances, RVM reverts to epoch truncation.



This figure shows the key data structures involved in incremental truncation. R1 through R5 are log entries. The reserved bit in page vector entries is used as an internal lock. Since page P1 is at the head of the page queue and has an uncommitted reference count of zero, it is the first page to be written to the recoverable data segment. The log head does not move, since P2 has the same log offset as P1. P2 is written next, and the log head is moved to P3's log offset. Incremental truncation is now blocked until P3's uncommitted reference count drops to zero.

Figure 7: Incremental Truncation

Figure 7 shows the two data structures used in incremental truncation. The first data structure is a page vector for each mapped region that maintains the modification status of that region's pages. The page vector is loosely analogous to a VM page table: the entry for a page contains a *dirty bit* and an *uncommited reference count*. A page is marked

dirty when it has committed changes. The uncommitted reference count is incremented as `set_ranges` are executed, and decremented when the changes are committed or aborted. On commit, the affected pages are marked dirty. The second data structure is a FIFO queue of page modification descriptors that specifies the order in which dirty pages should be written out in order to move the log head. Each descriptor specifies the log offset of the first record referencing that page. The queue contains no duplicate page references: a page is mentioned only in the earliest descriptor in which it could appear. A step in incremental truncation consists of selecting the first descriptor in the queue, writing out the pages specified by it, deleting the descriptor, and moving the log head to the offset specified by the next descriptor. This step is repeated until the desired amount of log space has been reclaimed.

5.2. Optimizations

Early experience with RVM indicated two distinct opportunities for substantially reducing the volume of data written to the log. We refer to these as *intra-transaction* and *inter-transaction* optimizations respectively.

Intra-transaction optimizations arise when `set-range` calls specifying identical, overlapping, or adjacent memory addresses are issued within a single transaction. Such situations typically occur because of modularity and defensive programming in applications. Forgetting to issue a `set-range` call is an insidious bug, while issuing a duplicate call is harmless. Hence applications are often written to err on the side of caution. This is particularly common when one part of an application begins a transaction, and then invokes procedures elsewhere to perform actions within that transaction. Each of those procedures may perform `set-range` calls for the areas of recoverable memory it modifies, even if the caller or some other procedure is supposed to have done so already. Optimization code in RVM causes duplicate `set-range` calls to be ignored, and overlapping and adjacent log records to be coalesced.

Inter-transaction optimizations occur only in the context of no-flush transactions. Temporal locality of reference in input requests to an application often translates into locality of modifications to recoverable memory. For example, the command "`cp d1/* d2`" on a Coda client will cause as many no-flush transactions updating the data structure in RVM for `d2` as there are children of `d1`. Only the last of these updates needs to be forced to the log on a future flush. The check for inter-transaction optimization is performed at commit time. If the modifications being committed subsume those from an earlier unflushed transaction, the older log records are discarded.

6. Status and Experience

RVM has been in daily use for over two years on hardware platforms such as IBM RTs, DEC MIPS workstations, Sun Sparc workstations, and a variety of Intel 386/486-based laptops and workstations. Memory capacity on these machines ranges from 12MB to 64 MB, while disk capacity ranges from 60MB to 2.5GB. Our personal experience with RVM has only been on Mach 2.5 and 3.0. But RVM has been ported to SunOS and SGI IRIX at MIT, and we are confident that ports to other Unix platforms will be straightforward. Most applications using RVM have been written in C or C++, but a few have been written in Standard ML. A version of the system that uses incremental truncation is being debugged.

Our original intent was just to replace Camelot by RVM on servers, in the role described in Section 2.2. But positive experience with RVM has encouraged us to expand its use. For example, transparent *resolution* of directory updates made to partitioned server replicas is done using a log-based strategy [17]. The logs for resolution are maintained in RVM. Clients also use RVM now, particularly for supporting *disconnected operation* [16]. The persistence of changes made while disconnected is achieved by storing *replay logs* in RVM, and user advice for long-term cache management is stored in a *hoard database* in RVM.

An unexpected use of RVM has been in debugging Coda servers and clients [31]. As Coda matured, we ran into hard-to-reproduce bugs involving corrupted persistent data structures. We realized that the information in RVM's log offered excellent clues to the source of these corruptions. All we had to do was to save a copy of the log before truncation, and to build a post-mortem tool to search and display the history of modifications recorded by the log.

The most common source of programming problems in using RVM has been in forgetting to do a `set-range` call prior to modifying an area of recoverable memory. The result is disastrous, because RVM does not create a new-value record for this area upon transaction commit. Hence the restored state after a crash or shutdown will not reflect modifications by the transaction to that area of memory. The current solution, as described in Section 5.2, is to program defensively. A better solution would be language-based, as discussed in Section 8.

7. Evaluation

A fair assessment of RVM must consider two distinct issues. From a software engineering perspective, we need to ask whether RVM's code size and complexity are commensurate with its functionality. From a systems perspective, we need to know whether RVM's focus on simplicity has resulted in unacceptable loss of performance.

To address the first issue, we compared the source code of RVM and Camelot. RVM's mainline code is approximately 10K lines of C, while utilities, test programs and other auxiliary code contribute a further 10K lines. Camelot has a mainline code size of about 60K lines of C, and auxiliary code of about 10K lines. These numbers do not include code in Mach for features like IPC and the external pager that are critical to Camelot.

Thus the total size of code that has to be understood, debugged, and tuned is considerably smaller for RVM. This translates into a corresponding reduction of effort in maintenance and porting. What is being given up in return is support for nesting and distribution, as well as flexibility in areas such as choice of logging strategies — a fair trade by our reckoning.

To evaluate the performance of RVM we used controlled experimentation as well as measurements from Coda servers and clients in actual use. The specific questions of interest to us were:

- How serious is the lack of integration between RVM and VM?
- What is RVM's impact on scalability?
- How effective are intra- and inter-transaction optimizations?

7.1. Lack of RVM-VM Integration

As discussed in Section 3.2, the separation of RVM from the VM component of an operating system could hurt performance. To quantify this effect, we designed a variant of the industry-standard TPC-A benchmark [32] and used it in a series of carefully controlled experiments.

7.1.1. The Benchmark

The TPC-A benchmark is stated in terms of a hypothetical bank with one or more branches, multiple tellers per branch, and many customer accounts per branch. A transaction updates a randomly chosen account, updates branch and teller balances, and appends a history record to an audit trail.

In our variant of this benchmark, we represent all the data structures accessed by a transaction in recoverable memory. The number of accounts is a parameter of our benchmark. The accounts and the audit trail are represented as arrays of 128-byte and 64-byte records respectively. Each of these data structures occupies close to half the total recoverable memory. The sizes of the data structures for teller and branch balances are insignificant.

Access to the audit trail is always sequential, with wrap-around. The pattern of accesses to the account array is a second parameter of our benchmark. The best case for paging performance occurs when accesses are sequential. The worst case occurs when accesses are uniformly distributed across all accounts. To represent the average case, the benchmark uses an access pattern that exhibits considerable temporal locality. In this access pattern, referred to as *localized*, 70% of the transactions update accounts on 5% of the pages, 25% of the transactions update accounts on a different 15% of the pages, and the remaining 5% of the transactions update accounts on the remaining 80% of the pages. Within each set, accesses are uniformly distributed.

7.1.2. Results

Our primary goal in these experiments was to understand the throughput of RVM over its intended domain of use. This corresponds to situations where paging rates are low, as discussed in Section 3.2. A secondary goal was to observe performance degradation relative to Camelot as paging becomes more significant. We expected this to shed light on the importance of RVM-VM integration.

To meet these goals, we conducted experiments for account arrays ranging from 32K entries to about 450K entries. This roughly corresponds to ratios of 10% to 175% of total recoverable memory size to total physical memory size. At each account array size, we performed the experiment for sequential, random, and localized account access patterns. Table 1 and Figure 8 present our results. Hardware and other relevant experimental conditions are described in Table 1.

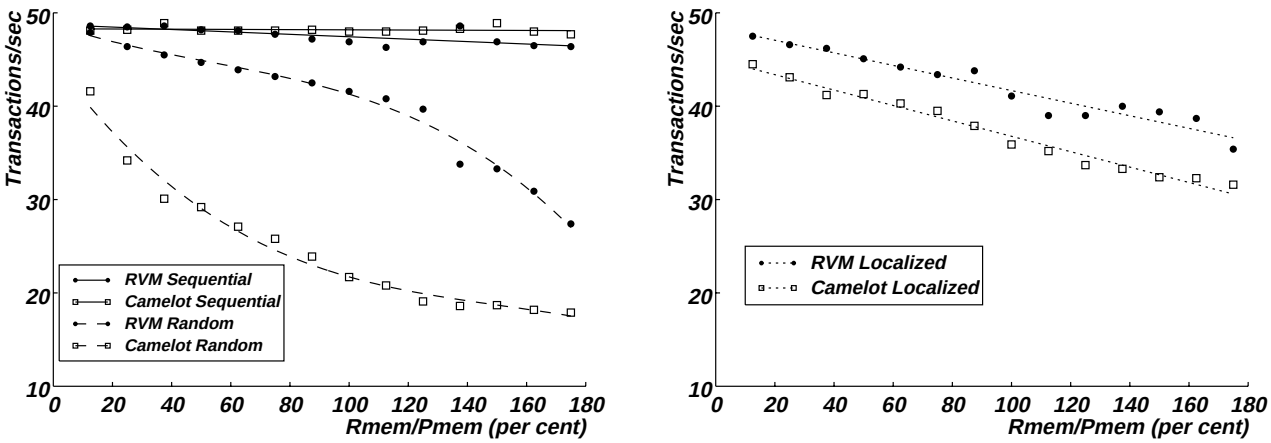
For sequential account access, Figure 8(a) shows that RVM and Camelot offer virtually identical throughput. This throughput hardly changes as the size of recoverable memory increases. The average time to perform a log force on the disks used in our experiments is about 17.4 milliseconds. This yields a theoretical maximum throughput of 57.4 transactions per second, which is within 15% of the observed best-case throughputs for RVM and Camelot.

When account access is random, Figure 8(a) shows that RVM's throughput is initially close to its value for sequential access. As recoverable memory size increases, the effects of paging become more significant, and throughput drops. But the drop does not become serious until recoverable memory size exceeds about 70% of physical memory size. The random access case is precisely where one would expect Camelot's integration with Mach to be most valuable. Indeed, the convexities of the curves in Figure 8(a) show that Camelot's degradation is more graceful than RVM's. But even at the highest ratio of recoverable to physical memory size, RVM's throughput is better than Camelot's.

No. of Accounts	Rmem / Pmem	RVM (Trans/Sec)			Camelot (Trans/Sec)		
		Sequential	Random	Localized	Sequential	Random	Localized
32768	12.5%	48.6 (0.0)	47.9 (0.0)	47.5 (0.0)	48.1 (0.0)	41.6 (0.4)	44.5 (0.2)
65536	25.0%	48.5 (0.2)	46.4 (0.1)	46.6 (0.0)	48.2 (0.0)	34.2 (0.3)	43.1 (0.6)
98304	37.5%	48.6 (0.0)	45.5 (0.0)	46.2 (0.0)	48.9 (0.1)	30.1 (0.2)	41.2 (0.2)
131072	50.0%	48.2 (0.0)	44.7 (0.2)	45.1 (0.0)	48.1 (0.0)	29.2 (0.0)	41.3 (0.1)
163840	62.5%	48.1 (0.0)	43.9 (0.0)	44.2 (0.1)	48.1 (0.0)	27.1 (0.2)	40.3 (0.2)
196608	75.0%	47.7 (0.0)	43.2 (0.0)	43.4 (0.0)	48.1 (0.4)	25.8 (1.2)	39.5 (0.8)
229376	87.5%	47.2 (0.1)	42.5 (0.0)	43.8 (0.1)	48.2 (0.2)	23.9 (0.1)	37.9 (0.2)
262144	100.0%	46.9 (0.0)	41.6 (0.0)	41.1 (0.0)	48.0 (0.0)	21.7 (0.0)	35.9 (0.2)
294912	112.5%	46.3 (0.6)	40.8 (0.5)	39.0 (0.6)	48.0 (0.0)	20.8 (0.2)	35.2 (0.1)
327680	125.0%	46.9 (0.7)	39.7 (0.0)	39.0 (0.5)	48.1 (0.1)	19.1 (0.0)	33.7 (0.0)
360448	137.5%	48.6 (0.0)	33.8 (0.9)	40.0 (0.0)	48.3 (0.0)	18.6 (0.0)	33.3 (0.1)
393216	150.0%	46.9 (0.2)	33.3 (1.4)	39.4 (0.4)	48.9 (0.0)	18.7 (0.1)	32.4 (0.2)
425984	162.5%	46.5 (0.4)	30.9 (0.3)	38.7 (0.2)	48.0 (0.0)	18.2 (0.0)	32.3 (0.2)
458752	175.0%	46.4 (0.4)	27.4 (0.2)	35.4 (1.0)	47.7 (0.0)	17.9 (0.1)	31.6 (0.0)

This table presents the measured steady-state throughput, in transactions per second, of RVM and Camelot on the benchmark described in Section 7.1.1. The column labelled "Rmem/Pmem" gives the ratio of recoverable to physical memory size. Each data point gives the mean and standard deviation (in parenthesis) of the three trials with most consistent results, chosen from a set of five to eight. The experiments were conducted on a DEC 5000/200 with 64MB of main memory and separate disks for the log, external data segment, and paging file. Only one thread was used to run the benchmark. Only processes relevant to the benchmark ran on the machine during the experiments. Transactions were required to be fully atomic and permanent. Inter- and intra-transaction optimizations were enabled in the case of RVM, but not effective for this benchmark. This version of RVM only supported epoch truncation; we expect incremental truncation to improve performance significantly.

Table 1: Transactional Throughput



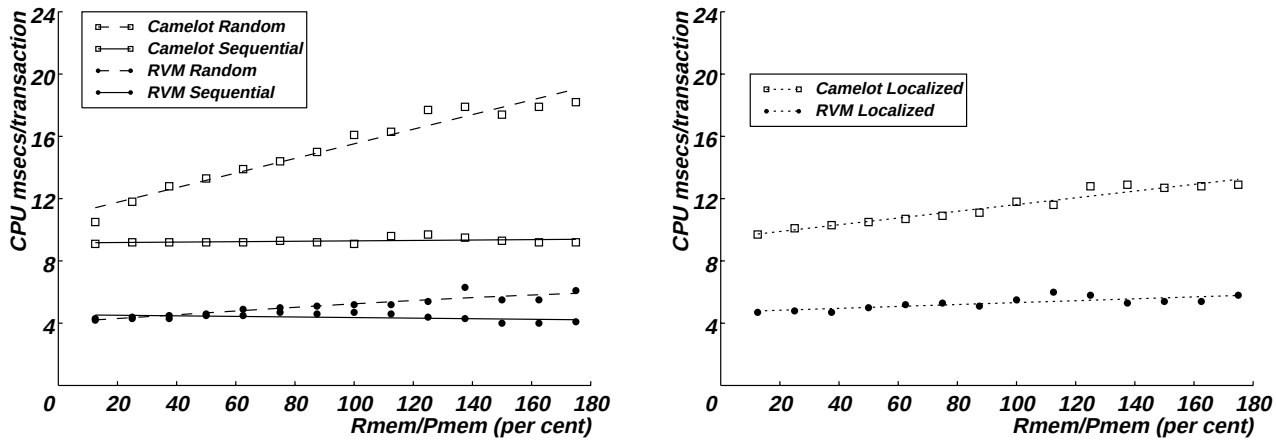
(a) Best and Worst Cases (b) Average Case

These plots illustrate the data in Table 1. For clarity, the average case is presented separately from the best and worst cases.

Figure 8: Transactional Throughput

For localized account access, Figure 8(b) shows that RVM's throughput drops almost linearly with increasing recoverable memory size. But the drop is relatively slow, and performance remains acceptable even when recoverable memory size approaches physical memory size. Camelot's throughput also drops linearly, and is consistently worse than RVM's throughput.

These measurements confirm that RVM's simplicity is not an impediment to good performance for its intended application domain. A conservative interpretation of the data in Table 1 indicates that applications with good locality can use up to 40% of physical memory for active recoverable data, while keeping throughput degradation to less than 10%. Applications with poor locality have to restrict active recoverable data to less than 25% for similar performance. Inactive recoverable data can be much larger, constrained only by startup latency and virtual memory limits imposed by the operating system. The comparison with Camelot is especially revealing. In spite of the fact that RVM is not integrated with VM, it is able to outperform Camelot over a broad range of workloads.



(a) Worst and Best Cases

(b) Average Case

These plots depict the measured CPU usage of RVM and Camelot during the experiments described in Section 7.1.2. As in Figure 8, we have separated the average case from the best and worst cases for visual clarity. To save space, we have omitted the table of data (similar to Table 1) on which these plots are based.

Figure 9: Amortized CPU Cost per Transaction

Although we were gratified by these results, we were puzzled by Camelot's behavior. For low ratios of recoverable to physical memory we had expected both Camelot's and RVM's throughputs to be independent of the degree of locality in the access pattern. The data shows that this is indeed the case for RVM. But in Camelot's case, throughput is highly sensistive to locality even at the lowest recoverable to physical memory ratio of 12.5%. At that ratio Camelot's throughput in transactions per second drops from 48.1 in the sequential case to 44.5 in the localized case, and to 41.6 in the random case.

Closer examination of the raw data indicates that the drop in throughput is attributable to much higher levels of paging activity sustained by the Camelot Disk Manager. We conjecture that this increased paging activity is induced by an overly aggressive log truncation strategy in the Disk Manager. During truncation, the Disk Manager writes out all dirty pages referenced by entries in the affected portion of the log. When truncation is frequent and account access is random, many opportunities to amortize the cost of writing out a dirty page across multiple transactions are lost. Less frequent truncation or sequential account access result in fewer such lost opportunities.

7.2. Scalability

As discussed in Section 2.3, Camelot's heavy toll on the scalability of Coda servers was a key influence on the design of RVM. It is therefore appropriate to ask whether RVM has yielded the anticipated gains in scalability. The ideal way to answer this question would be to repeat the experiment mentioned in Section 2.3, using RVM instead of Camelot. Unfortunately, such a direct comparison is not feasible because server hardware has changed considerably. Instead of IBM RTs we now use the much faster Decstation 5000/200s. Repeating the original experiment on current hardware is also not possible, because Coda servers now use RVM to the exclusion of Camelot.

Consequently, our evaluation of RVM's scalability is based on the same set of experiments described in Section 7.1. For each trial of that set of experiments, the total CPU usage on the machine was recorded. Since no extraneous activity was present on the machine, all CPU usage (whether in system or user mode) is attributable to the running of the benchmark. Dividing the total CPU usage by the number of transactions gives the average CPU cost per transaction, which is our metric of scalability. Note that this metric amortizes the cost of sporadic activities like log truncation and page fault servicing over all transactions.

Figure 9 compares the scalability of RVM and Camelot for each of the three access patterns described in Section 7.1.1. For sequential account access, RVM requires about half the CPU usage of Camelot. The actual values of CPU usage remain almost constant for both systems over all the recoverable memory sizes we examined.

For random account access, Figure 9(a) shows that both RVM and Camelot's CPU usage increase with recoverable memory size. But it is astonishing that even at the limit of our experimental range, RVM's CPU usage is less than Camelot's. In other words, the inefficiency of page fault handling in RVM is more than compensated for by its lower inherent overhead.

Machine name	Machine type	Transactions committed	Bytes Written to Log	Intra-Transaction Savings	Inter-Transaction Savings	Total Savings
grieg	server	267,224	289,215,032	20.7%	0.0%	20.7%
haydn	server	483,978	661,612,324	21.5%	0.0%	21.5%
wagner	server	248,169	264,557,372	20.9%	0.0%	20.9%
mozart	client	34,744	9,039,008	41.6%	26.7%	68.3%
ives	client	21,013	6,842,648	31.2%	22.0%	53.2%
verdi	client	21,907	5,789,696	28.1%	20.9%	49.0%
bach	client	26,209	10,787,736	25.8%	21.9%	47.7%
purcell	client	76,491	12,247,508	41.3%	36.2%	77.5%
berlioz	client	101,168	14,918,736	17.3%	64.3%	81.6%

This table presents the observed reduction in log traffic due to RVM optimizations. The column labelled "Bytes Written to Log" shows the log size after both optimizations were applied. The columns labelled "Intra-Transaction Savings" and "Inter-Transaction Savings" indicate the percentage of the original log size that was supressed by each type of optimization. This data was obtained over a 4-day period in March 1993 from Coda clients and servers.

Table 2: Savings Due to RVM Optimizations

For localized account access, Figure 9(b) shows that CPU usage increase linearly with recoverable memory size for both RVM and Camelot. For all sizes investigated, RVM's CPU usage remains well below that of Camelot's.

Overall, these measurements establish that RVM is considerably less of a CPU burden than Camelot. Over most of the workloads investigated, RVM typically requires about half the CPU usage of Camelot. We anticipate that refinements to RVM such as incremental truncation will further improve its scalability.

RVM's lower CPU usage follows directly from our decision to structure it as a library rather than as a collection of tasks communicating via IPC. As mentioned in Section 3.3, Mach IPC costs about 600 times as much as a procedure call on the hardware we used for our experiments. Further contributing to reduced CPU usage are the substantially smaller path lengths in various RVM components due to their inherently simpler functionality.

7.3. Effectiveness of Optimizations

To estimate the value of intra- and inter-transaction optimizations, we instrumented RVM to keep track of the total volume of log data eliminated by each technique. Table 2 presents the observed savings in log traffic for a representative sample of Coda clients and servers in our environment.

The data in Table 2 shows that both servers and clients benefit significantly from intra-transaction optimization. The savings in log traffic is typically between 20% and 30%, though some machines exhibit substantially higher savings. Inter-transaction optimizations typically reduce log traffic on clients by another 20-30%. Servers do not benefit from this type of optimization, because it is only applicable to no-flush transactions. RVM optimizations have proved to be especially valuable for good performance on portable Coda clients, because disks on those machines tend to be selected on the basis of size, weight, and power consumption rather than performance.

7.4. Broader Analysis

A fair criticism of the conclusions drawn in Sections 7.1 and 7.2 is that they are based solely on comparison with a research prototype, Camelot. A favorable comparison with well-tuned commercial products would strengthen the claim that RVM's simplicity does not come at the cost of good performance. Unfortunately, such a comparison is not currently possible because no widely used commercial product supports recoverable virtual memory. Hence a performance analysis of broader scope will have to await the future.

8. RVM as a Building Block

The simplicity of the abstraction offered by RVM makes it a versatile base on which to implement more complex functionality. In principle, any abstraction that requires persistent data structures with clean local failure semantics can be built on top of RVM. In some cases, minor extensions of the RVM interface may be necessary.

For example, nested transactions could be implemented using RVM as a substrate for bookkeeping state such as the undo logs of nested transactions. Only top-level `begin`, `commit`, and `abort` operations would be visible to RVM. Recovery would be simple, since the restoration of committed state would be handled entirely by RVM. The feasibility of this approach has been confirmed by the Venari project [37].

Support for distributed transactions could also be provided by a library built on RVM. Such a library would provide coordinator and subordinate routines for each phase of a two-phase commit, as well as for operations such as beginning a transaction and adding new sites to a transaction. Recovery after a coordinator crash would involve RVM recovery, followed by approriate termination

of distributed transactions in progress at the time of the crash. The communication mechanism could be left unspecified until runtime by using upcalls from the library to perform communications. RVM would have to be extended to enable a subordinate to undo the effects of a first-phase commit if the coordinator decides to abort. One way to do this would be to extend `end_transaction` to return a list of the old-value records generated by the transaction. These records could be preserved by the library at each subordinate until the outcome of the two-phase commit is clear. On a global commit, the records would be discarded. On a global abort, the library at each subordinate could use the saved records to construct a compensating RVM transaction.

RVM can also be used as the basis of runtime systems for languages that support persistence. Experience with Avalon [38], which was built on Camelot, confirms that recoverable virtual memory is indeed an appropriate abstraction for implementing language-based local persistence. Language support would alleviate the problem mentioned in Section 6 of programmers forgetting to issue `set-range` calls: compiler-generated code could issue these calls transparently. An approximation to a language-based solution would be to use a post-compilation augmentation phase to test for accesses to mapped RVM regions and to generate `set-range` calls.

Further evidence of the versatility of RVM is provided by the recent work of O'Toole et al [25]. In this work, RVM segments are used as the stable to-space and from-space of the heap for a language that supports concurrent garbage collection of persistent data. While the authors suggest some improvements to RVM for this application, their work establishes the suitability of RVM for a very different context from the one that motivated it.

9. Related Work

The field of transaction processing is enormous. In the space available, it is impossible to fully attribute all the past work that has indirectly influenced RVM. We therefore restrict our discussion here to placing RVM's contribution in proper perspective, and to clarifying its relationship to its closest relatives.

Since the original identification of transactional properties and techniques for their realization [13, 18], attention has been focused on three areas. One area has been the enrichment of the transactional concept along dimensions such as distribution, nesting [23], and longevity [11]. A second area has been the incorporation of support for transactions into languages [21], operating systems [15], and hardware [6]. A third area has been the development of techniques for achieving high performance in OLTP environments with very large data volumes and poor locality [12].

In contrast to those efforts, RVM represents a "back to basics" movement. Rather than embellishing the transactional abstraction or its implementation, RVM seeks to simplify both. It poses and answers the question "What is the simplest realization of essential transactional properties for the average application?" By doing so, it makes transactions accessible to applications that have hitherto balked at the baggage that comes with sophisticated transactional facilities.

The virtues of simplicity for small databases have been extolled previously by Birrell et al [5]. Their design is is even simpler than RVM's, and is based upon new-value logging and full-database checkpointing. Each transaction is constrained to update only a single data item. There is no support for explicit transaction abort. Updates are recorded in a log file on disk, then reflected in the in-memory database image. Periodically, the entire memory image is checkpointed to disk, the log file deleted, and the new checkpoint file renamed to be the current version of the database. Log truncation occurs only during crash recovery, not during normal operation.

The reliance of Birrell et al's technique on full-database checkpointing makes the technique practical only for applications which manage small amounts of recoverable data and which have moderate update rates. The absence of support for multi-item updates and for explicit abort further limits its domain of use. RVM is more versatile without being substantially more complex.

Transaction processing monitors (TPMs), such as Encina [35, 40] and Tuxedo [1, 36], are important commercial products. TPMs add distribution and support services to OLTP back-ends, and integrate heterogeneous systems. Like centralized database managers, TPM back-ends are usually monolithic in structure. They encapsulate all three of the basic transactional properties and provide data access via a query language interface. This is in contrast to RVM, which supports only atomicity and the process failure aspect of permanence, and which provides access to recoverable data as mapped virtual memory.

A more modular approach is used in the Transarc TP toolkit, which is the back-end for the Encina TPM. The functionality provided by RVM corresponds primarily to the recovery, logging, and physical storage modules of the Transarc toolkit. RVM differs from the corresponding Transarc toolkit components in two important ways. First, RVM is structured entirely as a library that is linked with

applications, while some of the toolkit's modules are separate processes. Second, recoverable storage is accessed as mapped memory in RVM, whereas the Transarc toolkit offers access via the conventional buffered I/O model.

Chew et al have recently reported on their efforts to enhance the Mach kernel to support recoverable virtual memory [7]. Their work carries Camelot's idea of providing system-level support for recoverable memory a step further, since their support is in the kernel rather than in a user-level Disk Manager. In contrast, RVM avoids the need for specialized operating system support, thereby enhancing portability.

RVM's debt to Camelot should be obvious by now. Camelot taught us the value of recoverable virtual memory and showed us the merits and pitfalls of a specific approach to its implementation. Whereas Camelot was willing to require operating system support to achieve generality, RVM has restrained generality within limits that preserve operating system independence.

10. Conclusion

In general, RVM has proved to be useful wherever we have encountered a need to maintain persistent data structures with clean failure semantics. The only constraints upon its use have been the need for the size of the data structures to be a small fraction of disk capacity, and for the working set size of accesses to them to be significantly less than main memory.

The term "lightweight" in the title of this paper connotes two distinct qualities. First, it implies ease of learning and use. Second, it signifies minimal impact upon system resource usage. RVM is indeed lightweight along both these dimensions. A Unix programmer thinks of RVM in essentially the same way he thinks of a typical subroutine library, such as the `stdio` package.

While the importance of the transactional abstraction has been known for many years, its use in low-end applications has been hampered by the lack of a lightweight implementation. Our hope is that RVM will remedy this situation. While integration with the operating system may be unavoidable for very demanding applications, it can be a double-edged sword, as this paper has shown. For a broad class of less demanding applications, we believe that RVM represents close to the limit of what is attainable without hardware or operating system support.

Acknowledgements

Marvin Theimer and Robert Hagmann participated in the early discussions leading to the design of RVM. We wish to thank the designers and implementors of Camelot, especially Peter Stout and Lily Mummert, for helping us understand and use their system. The comments of our SOSP shepherd, Bill Weihl, helped us improve the presentation significantly.

References

[1] Andrade, J.M., Carges, M.T., Kovach, K.R.
Building a Transaction Processing System on UNIX Systems.
In *UniForum Conference Proceedings*. San Francisco, CA, February, 1989.

[2] Baron, R.V., Black, D.L., Bolosky, W., Chew, J., Golub, D.B., Rashid, R.F., Tevanian, Jr, A., Young, M.W.
Mach Kernel Interface Manual
School of Computer Science, Carnegie Mellon University, 1987.

[3] Bernstein, P.A., Hadzilacos, V., Goodman, N.
Concurrency Control and Recovery in Database Systems.
Addison Wesley, 1987.

[4] Bershad, B.N., Anderson, T.E., Lazowska, E.D., Levy, H.M.
Lightweight Remote Procedure Call.
ACM Transactions on Computer Systems 8(1), February, 1990.

[5] Birrell, A.B., Jones, M.B., Wobber, E.P.
A Simple and Efficient Implementation for Small Databases.
In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*. Austin, TX, November, 1987.

[6] Chang, A., Mergen, M.F.
801 Storage: Architecture and Programming.
ACM Transactions on Computer Systems 6(1), February, 1988.

[7] Chew, K-M, Reddy, A.J., Romer, T.H., Silberschatz, A.
Kernel Support for Recoverable-Persistent Virtual Memory.
In *Proceedings of the USENIX Mach III Symposium*. Santa Fe, NM, April, 1993.

[8] Cooper, E.C., Draves, R.P.
C Threads.
Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, June, 1988.

[9] Eppinger, J.L.
Virtual Memory Management for Transaction Processing Systems.
PhD thesis, Department of Computer Science, Carnegie Mellon University, February, 1989.

[10] Eppinger, J.L., Mummert, L.B., Spector, A.Z.
Camelot and Avalon.
Morgan Kaufmann, 1991.

[11] Garcia-Molina, H., Salem, K.
Sagas.
In *Proceedings of the ACM Sigmod Conference*. 1987.

[12] Good, B., Homan, P.W., Gawlick, D.E., Sammer, H.
One thousand transactions per second.
In *Proceedings of IEEE Compcon*. San Francisco, CA, 1985.

[13] Gray, J.
Notes on Database Operating Systems.
In Goos, G., Hartmanis, J. (editor), *Operating Systems: An Advanced Course*. Springer Verlag, 1978.

[14] Gray, J., Reuter, A.
Transaction Processing: Concepts and Techniques.
Morgan Kaufmann, 1993.

[15] Haskin, R., Malachi, Y., Sawdon, W., Chan, G.
Recovery Management in QuickSilver.
ACM Transactions on Computer Systems 6(1), February, 1988.

[16] Kistler, J.J., Satyanarayanan, M.
Disconnected Operation in the Coda File System.
ACM Transactions on Computer Systems 10(1), February, 1992.

[17] Kumar, P., Satyanarayanan, M.
Log-based Directory Resolution in the Coda File System.
In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*. San Diego, CA, January, 1993.

[18] Lampson, B.W.
Atomic Transactions.
In Lampson, B.W., Paul, M., Siegert, H.J. (editors), *Distributed Systems -- Architecture and Implementation*. Springer Verlag, 1981.

[19] Lampson, B. W.
Hints for Computer System Design.
In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*. Bretton Woods, NH, October, 1983.

[20] Leffler, S.L., McKusick, M.K., Karels, M.J., Quarterman, J.S.
The Design and Implementation of the 4.3BSD Unix Operating System.
Addison Wesley, 1989.

[21] Liskov, B.H., Scheifler, R.W.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Languages 5(3), July, 1983.

[22] Mashburn, H., Satyanarayanan, M.
RVM User Manual
School of Computer Science, Carnegie Mellon University, 1992.

[23] Moss, J.E.B.
Nested Transactions: An Approach to Reliable Distributed Computing.
MIT Press, 1985.

[24] Nettles, S.M., Wing, J.M.
Persistence + Undoability = Transactions.
In *Proceedings of HICSS-25*. Hawaii, January, 1992.

[25] O'Toole, J., Nettles, S., Gifford, D.
Concurrent Compacting Garbage Collection of a Persistent Heap.
In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*. Asheville, NC, December, 1993.

[26] Ousterhout, J.K.
Why Aren't Operating Systems Getting Faster As Fast as Hardware?
In *Proceedings of the USENIX Summer Conference*. Anaheim, CA, June, 1990.

[27] Patterson, D.A., Gibson, G., Katz, R.
A Case for Redundant Arrays of Inexpensive Disks (RAID).
In *Proceedings of the ACM SIGMOD Conference*. 1988.

[28] Rosenblum, M., Ousterhout, J.K.
The Design and Implementation of a Log-Structured File System.
ACM Transactions on Computer Systems 10(1), February, 1992.

[29] Satyanarayanan, M.
RPC2 User Guide and Reference Manual
School of Computer Science, Carnegie Mellon University, 1991.

[30] Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.
Coda: A Highly Available File System for a Distributed Workstation Environment.
IEEE Transactions on Computers 39(4), April, 1990.

[31] Satyanarayanan, M., Steere, D.C., Kudo, M., Mashburn, H.
Transparent Logging as a Technique for Debugging Complex Distributed Systems.
In *Proceedings of the Fifth ACM SIGOPS European Workshop*. Mont St. Michel, France, September, 1992.

[32] Serlin, O.
The History of DebitCredit and the TPC.
In Gray, J. (editors), *The Benchmark Handbook*. Morgan Kaufman, 1991.

[33] Spector, A.Z.
The Design of Camelot.
In Eppinger, J.L., Mummert, L.B., Spector, A.Z. (editors), *Camelot and Avalon*. Morgan Kaufmann, 1991.

[34] Stout, P.D., Jaffe, E.D., Spector, A.Z.
Performance of Select Camelot Functions.
In Eppinger, J.L., Mummert, L.B., Spector, A.Z. (editors), *Camelot and Avalon*. Morgan Kaufmann, 1991.

[35] *Encina Product Overview*
Transarc Corporation, 1991.

[36] *TUXEDO System Product Overview*
Unix System Laboratories, 1993.

[37] Wing, J.M., Faehndrich, M., Morrisett, G., and Nettles, S.M.
Extensions to Standard ML to Support Transactions.
In *ACM SIGPLAN Workshop on ML and its Applications*. San Francisco, CA, June, 1992.

[38] Wing, J.M.
The Avalon Language.
In Eppinger, J.L., Mummert, L.B., Spector, A.Z. (editors), *Camelot and Avalon*. Morgan Kaufmann, 1991.

[39] Young, M.W.
Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.
PhD thesis, Department of Computer Science, Carnegie Mellon University, November, 1989.

[40] Young, M.W., Thompson, D.S., Jaffe, E.
A Modular Architecture for Distributed Transaction Processing.
In *Proceedings of the USENIX Winter Conference*. Dallas, TX, January, 1991.

Granularity of Locks and Degrees of Consistency in a Shared Data Base

J.N. Gray
R.A. Lorie
G.R. Putzolu
I.L. Traiger

IBM Research Laboratory
San Jose, California

ABSTRACT: In the first part of the paper the problem of choosing the granularity (size) of lockable objects is introduced and the related tradeoff between concurrency and overhead is discussed. A locking protocol which allows simultaneous locking at various granularities by different transactions is presented. It is based on the introduction of additional lock modes besides the conventional share mode and exclusive mode. A proof is given of the equivalence of this protocol to a conventional one.

In the second part of the paper the issue of consistency in a shared environment is analyzed. This discussion is motivated by the realization that some existing data base systems use automatic lock protocols which insure protection only from certain types of inconsistencies (for instance those arising from transaction backup), thereby automatically providing a limited degree of consistency. Four degrees of consistency are introduced. They can be roughly characterized as follows: degree 0 protects others from your updates, degree 1 additionally provides protection from losing updates, degree 2 additionally provides protection from reading incorrect data items, and degree 3 additionally provides protection from reading incorrect relationships among data items (i.e. total protection). A discussion follows on the relationships of the four degrees to locking protocols, concurrency, overhead, recovery and transaction structure.

Lastly, these ideas are related to existing data management systems.

RJ 1654

Presented at IFIP Working Conference on Modelling of Data Base Management Systems, Freudenstadt, Germany, Dec. 1975.

I. GRANULARITY OF LOCKS:

An important problem which arises in the design of a data base management system is choosing the lockable units, i.e. the data aggregates which are atomically locked to insure consistency. Examples of lockable units are areas, files, individual records, field values, intervals of field values, etc.

The choice of lockable units presents a tradeoff between concurrency and overhead, which is related to the size or granularity of the units themselves. On the one hand, concurrency is increased if a fine lockable unit (for example a record or field) is chosen. Such unit is appropriate for a "simple" transaction which accesses few records. On the other hand a fine unit of locking would be costly for a "complex" transaction which accesses a large number of records. Such a transaction would have to set/reset a large number of locks, hence incurring too many times the computational overhead of accessing the lock subsystem, and the storage overhead of representing a lock in memory. A coarse lockable unit (for example a file) is probably convenient for a transaction which accesses many records. However, such a coarse unit discriminates against transactions which only want to lock one member of the file. From this discussion it follows that it would be desirable to have lockable units of different granularities coexisting in the same system.

In the following a lock protocol satisfying these requirements will be described. Related implementation issues of scheduling,granting and converting lock requests are not discussed. They were covered in a companion paper [1].

Hierarchical locks:

We will first assume that the set of resources to be locked is organized in a hierarchy. Note that the concept of hierarchy is used in the context of a collection of resources and has nothing to do with the data model used in a data base system. The hierarchy of Figure 1 may be suggestive. We adopt the notation that each level of the hierarchy is given a node type which is a generic name for all the node instances of that type. For example, the data base has nodes of type area as its immediate descendants, each area in turn has nodes of type file as its immediate descendants and each file has nodes of type record as its immediate descendants in the hierarchy. Since it is a hierarchy each node has a unique parent.

Figure 1. A sample lock hierarchy.

Each node of the hierarchy can be locked. If one requests exclusive access (X) to a particular node, then when the request is granted, the requestor has exclusive access to that node and implicitly to each of its descendants. If one requests shared access (S) to a particular node, then when the request is granted, the requestor has shared access to that node and implicitly to each descendant of that node. These two access modes lock an entire subtree rooted at the requested node.

Our goal is to find some technique for implicitly locking an entire subtree. In order to lock a subtree rooted at node R in share or exclusive mode it is important to prevent share or exclusive locks on the ancestors of R which would implicitly lock R and its descendants. Hence a new access mode, intention mode (I), is introduced. Intention mode is used to "tag" (lock) all ancestors of a node to be locked in share or exclusive mode. These tags signal the fact that locking is being done at a "finer" level and prevent implicit or explicit exclusive or share locks on the ancestors.

The protocol to lock a subtree rooted at node R in exclusive or share mode is to lock all ancestors of R in intention mode and to lock node R in exclusive or share mode. So for example using Figure 1, to lock a particular file one should obtain intention access to the data base, to the area containing the file and then request exclusive (or share) access to the file itself. This implicitly locks all records of the file in exclusive (or share) mode.

Access modes and compatibility:

We say that two lock requests for the same node by two different transaction are compatible if they can be granted concurrently. The mode of the request determines its compatibility with requests made by other transactions. The three modes: X, S and I are incompatible with one another but distinct S requests may be granted together and distinct I requests may be granted together.

The compatibilities among modes derive from their semantics. Share mode allows reading but not modification of the

corresponding resource by the requestor and by other transactions. The semantics of exclusive mode is that the grantee may read and modify the resource and no other transaction may read or modify the resource while the exclusive lock is set. The reason for dichotomizing share and exclusive access is that several share requests can be granted concurrently (are compatible) whereas an exclusive request is not compatible with any other request. Intention mode was introduced to be incompatible with share and exclusive mode (to prevent share and exclusive locks). However, intention mode is compatible with itself since two transactions having intention access to a node will explicitly lock descendants of the node in X, S or I mode and thereby will either be compatible with one another or will be scheduled on the basis of their requests at the finer level. For example, two transactions can be concurrently granted the data base and some area and some file in intention mode. In this case their explicit locks on records in the file will resolve any conflicts among them.

The notion of intention mode is refined to <u>intention share mode</u> (IS) and <u>intention exclusive mode</u> (IX) for two reasons: the intention share mode only requests share or intention share locks at the lower nodes of the tree (i.e. never requests an exclusive lock below the intention share node). Since read-only is a common form of access it will be profitable to distinguish this for greater concurrency. Secondly, if a transaction has an intention share lock on a node it can convert this to a share lock at a later time, but one cannot convert an intention exclusive lock to a share lock on a node (see [1] for a discussion of this point).

We recognize one further refinement of modes, namely <u>share and intention exclusive mode</u> (SIX). Suppose one transaction wants to read an entire subtree and to update particular nodes of that subtree. Using the modes provided so far it would have the options of: (a) requesting exclusive access to the root of the subtree and doing no further locking or (b) requesting intention exclusive access to the root of the subtree and explicitly locking the lower nodes in intention, share or exclusive mode. Alternative (a) has low concurrency. If only a small fraction of the read nodes are updated then alternative (b) has high locking overhead. The correct access mode would be share access to the subtree thereby allowing the transaction to read all nodes of the subtree without further locking <u>and</u> intention exclusive access to the subtree thereby allowing the transaction to set exclusive locks on those nodes in the subtree which are to be updated and IX or SIX locks on the intervening nodes. Since this is such a common case, SIX mode is introduced for this purpose. It is compatible with IS mode since other transactions requesting IS mode will explicitly lock lower nodes in IS or S mode thereby avoiding any updates (IX or X mode) produced by the SIX mode transaction. However SIX mode is not compatible with IX, S, SIX or X mode requests. An equivalent approach would be to consider only four modes (IS,IX,S,X), but to assume that a transaction can request both S and IX lock privileges on a resource.

Table 1 gives the compatibility of the request modes, where for completeness we have also introduced the <u>null mode</u> (NL) which represents the absence of requests of a resource by a transaction.

	NL	IS	IX	S	SIX	X
NL	YES	YES	YES	YES	YES	YES
IS	YES	YES	YES	YES	YES	NO
IX	YES	YES	YES	NO	NO	NO
S	YES	YES	NO	YES	NO	NO
SIX	YES	YES	NO	NO	NO	NO
X	YES	NO	NO	NO	NO	NO

Table 1. Compatibilities among access modes.

To summarize, we recognize six modes of access to a resource:

NL: Gives no access to a node i.e. represents the absence of a request of a resource.

IS: Gives <u>intention share</u> access to the requested node and allows the requestor to lock descendant nodes in S or IS mode. (It does <u>no</u> implicit locking.)

IX: Gives <u>intention exclusive</u> access to the requested node and allows the requestor to <u>explicitly</u> lock descendants in X, S, SIX, IX or IS mode. (It does <u>no</u> implicit locking.)

S: Gives <u>share</u> access to the requested node and to all descendants of the requested node without setting further locks. (It implicitly sets S locks on all descendants of the requested node.)

SIX: Gives <u>share</u> and <u>intention exclusive</u> access to the requested node. In particular it implicitly locks all descendants of the node in share mode and allows the requestor to explicitly lock descendant nodes in X, SIX or IX mode.

X: Gives <u>exclusive</u> access to the requested node and to all descendants of the requested node without setting further locks. (It implicitly sets X locks on all descendants.) (Locking lower nodes in S or IS mode would give no increased access.)

IS mode is the weakest non-null form of access to a resource. It carries fewer privileges than IX or S modes. IX mode allows IS, IX, S, SIX and X mode locks to be set on descendant nodes while S mode allows read only access to all descendants of the node without further locking. SIX mode carries the privileges of S and of IX mode (hence the name SIX). X mode is the most privileged form of access and allows reading and writing of all descendants

of a node without further locking. Hence the modes can be ranked in the partial order (lattice) of privileges shown in Figure 2. Note that it is not a total order since IX and S are incomparable.

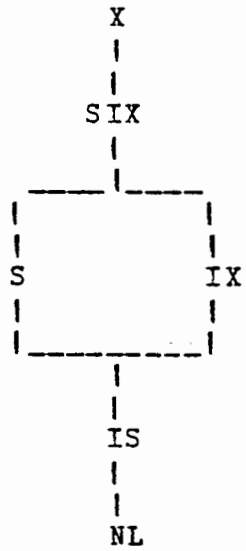


Figure 2. The partial ordering of modes by their privileges.

Rules for requesting nodes:

The implicit locking of nodes will not work if transactions are allowed to leap into the middle of the tree and begin locking nodes at random. The implicit locking implied by the S and X modes depends on all transactions obeying the following protocol:

(a) Before requesting an S or IS lock on a node, all ancestor nodes of the requested node must be held in IX or IS mode by the requestor.

(b) Before requesting an X, SIX or IX lock on a node, all ancestor nodes of the requested node must be held in SIX or IX mode by the requestor.

(c) Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to end of transaction, one should not hold a lower lock after releasing its ancestor.

To paraphrase this, locks are requested root to leaf, and released leaf to root. Notice that leaf nodes are never requested in intention mode since they have no descendants.

Several examples:

It may be instructive to give a few examples of hierarchical

request sequences:

To lock record R for read:

lock data-base	with mode = IS
lock area containing R	with mode = IS
lock file containing R	with mode = IS
lock record R	with mode = S

Don't panic, the transaction probably already has the data base, area and file lock.

To lock record R for write-exclusive access:

lock data-base	with mode = IX
lock area containing R	with mode = IX
lock file containing R	with mode = IX
lock record R	with mode = X

Note that if the records of this and the previous example are distinct, each request can be granted simultaneously to different transactions even though both refer to the same file.

To lock a file F for read and write access:

lock data-base	with mode = IX
lock area containing F	with mode = IX
lock file F	with mode = X

Since this reserves exclusive access to the file, if this request uses the same file as the previous two examples it or the other transactions will have to wait.

To lock a file F for complete scan and occasional update:

lock data-base	with mode = IX
lock area containing F	with mode = IX
lock file F	with mode = SIX

Thereafter, particular records in F can be locked for update by locking records in X mode. Notice that (unlike the previous example) this transaction is compatible with the first example. This is the reason for introducing SIX mode.

To quiesce the data base:
lock data base with mode = X.
Note that this locks everyone else out.

Directed acyclic graphs of locks:

The notions so far introduced can be generalized to work for directed acyclic graphs (DAG) of resources rather than simply hierarchies of resources. A tree is a simple DAG. The key observation is that to implicitly or explicitly lock a node, one should lock all the parents of the node in the DAG and so by induction lock all ancestors of the node. In particular, to lock a subgraph one must implicitly or explicitly lock all ancestors of the subgraph in the appropriate mode (for a tree there is only one parent). To give an example of a non-hierarchical structure, imagine the locks are organized as in Figure 3.

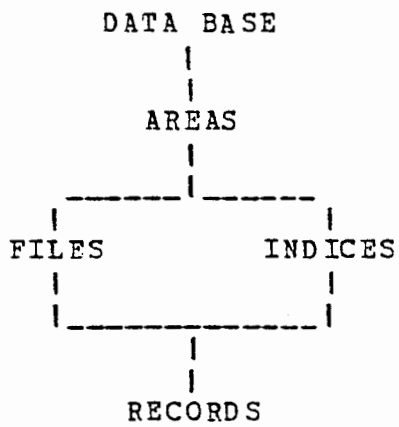


Figure 3. A non-hierarchical lock graph.

We postulate that areas are "physical" notions and that files, indices and records are logical notions. The data base is a collection of areas. Each area is a collection of files and indices. Each file has a corresponding index in the same area. Each record belongs to some file and to its corresponding index. A record is comprised of field values and some field is indexed by the index associated with the file containing the record. The file gives a sequential access path to the records and the index gives an associative access path to the records based on field values. Since individual fields are never locked, they do not appear in the lock graph.

To write a record R in file F with index I:

lock data base	with mode = IX
lock area containing F	with mode = IX
lock file F	with mode = IX
lock index I	with mode = IX
lock record R	with mode = X

Note that <u>all</u> paths to record R are locked. Alternaltively, one could lock F and I in exclusive mode thereby implicitly locking R in exclusive mode.

To give a more complete explanation we observe that a node can be locked <u>explicitly</u> (by requesting it) or <u>implicitly</u> (by appropriate explicit locks on the ancestors of the node) in one of five modes: IS, IX, S, SIX, X. However, the definition of implicit locks and the protocols for setting explicit locks have to be extended as follows:

A node is <u>implicitly granted in S</u> mode to a transaction if <u>at least one</u> of its parents is (implicitly or explicitly) granted to the transaction in S, SIX or X mode. By induction that means that at least one of the node's ancestors must be explicitly granted in S, SIX or X mode to the transaction.

A node is <u>implicitly granted in X</u> mode if <u>all</u> of its parents are

(implicitly or explicitly) granted to the transaction in X mode. By induction, this is equivalent to the condition that all nodes in some cut set of the collection of all paths leading from the node to the roots of the graph are explicitly granted to the transaction in X mode and all ancestors of nodes in the cut set are explicitly granted in IX or SIX mode.

From Figure 2, a node is implicitly granted in IS mode if it is implicitly granted in S mode, and a node is implicitly granted in IS, IX, S and SIX mode if it is implicitly granted in X mode.

<u>The protocol for explicitly requesting locks on a DAG</u>:

(a) Before requesting an S or IS lock on a node, one should request at least one parent (and by induction a path to a root) in IS (or greater) mode. As a consequence none of the ancestors along this path can be granted to another transaction in a mode incompatible with IS.

(b) Before requesting IX, SIX or X mode access to a node, one should request all parents of the node in IX (or greater) mode. As a consequence all ancestors will be held in IX (or greater mode) and cannot be held by other transactions in a mode incompatible with IX (i.e. S, SIX, X).

(c) Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to the end of transaction, one should not hold a lower lock after releasing its ancestors.

To give an example using Figure 3, a sequential scan of all records in file F need not use an index so one can get an implicit share lock on each record in the file by:

```
lock data base              with mode = IS
lock area containing F      with mode = IS
lock file F                 with mode = S
```

This gives implicit S mode access to all records in F. Conversely, to read a record in a file via the index I for file F, one need not get an implicit or explicit lock on file F:

```
lock data base              with mode = IS
lock area containing R      with mode = IS
lock index I                with mode = S
```

This again gives implicit S mode access to all records in index I (in file F). In both these cases, <u>only one path was locked for reading</u>.

But to insert, delete or update a record R in file F with index I one must get an implicit or explicit lock on all ancestors of R.

The first example of this section showed how an explicit X lock on

a record is obtained. To get an implicit X lock on all records in a file one can simply lock the index and file in X mode, or lock the area in X mode. The latter examples allow bulk load or update of a file without further locking since all records in the file are implicitly granted in X mode.

Proof of equivalence of the lock protocol.

We will now prove that the described lock protocol is equivalent to a conventional one which uses only two modes (S and X), and which locks only atomic resources (leaves of a tree or a directed graph).

Let G = (N,A) be a finite (directed) graph where N is the set of nodes and A is the set of arcs. G is assumed to be without circuits (i.e. there is no non-null path leading from a node n to itself). A node p is a parent of a node n and n is a child of p if there is an arc from p to n. A node n is a source (sink) if n has no parents (no children). Let SI be the set of sinks of G. An ancestor of node n is any node (including n) in a path from a source to n. A node-slice of a sink n is a collection of nodes such that each path from a source to n contains at least one of these nodes.

We also introduce the set of lock modes M = {NL,IS,IX,S,SIX,X} and the compatibility matrix C : MxM->{YES,NO} described in Table 1. We will call c : mxm->{YES,NO} the restriction of C to m = {NL,S,X}.

A lock-graph is a mapping L : N->M such that:

(a) if L(n) ∈ {IS,S} then either n is a source or there exists a parent p of n such that L(p) ∈ {IS,IX,S,SIX,X}. By induction there exists a path from a source to n such that L takes only values in {IS,IX,S,SIX,X} on it. Equivalently L is not equal to NL on the path.

(b) if L(n) ∈ {IX,SIX,X} then either n is a root or for all parents p1...pk of n we have L(pi) ∈ {IX,SIX,X} (i=1...k). By induction L takes only values in {IX,SIX,X} on all the ancestors of n.

The interpretation of a lock-graph is that it gives a map of the explicit locks held by a particular transaction observing the six state lock protocol described above. The notion of projection of a lock-graph is now introduced to model the set of implicit locks on atomic resources correspondingly acquired by a transaction.

The projection of a lock-graph L is the mapping l: SI->m constructed as follows:

(a) l(n)=X if there exist a node-slice {n1...ns} of n such that L(ni)=X (i=1...ns).

(b) l(n)=S if (a) is not satisfied and there exist an ancestor a of n such that L(a) ∈ {S,SIX,X}.

(c) l(n)=NL if (a) and (b) are not satisfied.

Two lock-graphs L1 and L2 are said to be <u>compatible</u> if C(L1(n),L2(n))=YES for all n ∈ N. Similarly two projections l1 and l2 are compatible if c(l1(n),l2(n))=YES for all n ∈ SI.

We are now in a position to prove the following <u>Theorem</u>:

If two lock-graphs L1 and L2 are compatible then their projections l1 and l2 are compatible.In other words if the explicit locks set by two transactions are not conflicting then also the three-state locks implicitely acquired are not conflicting.

<u>Proof</u>: Assume that l1 and l2 are incompatible. We want to prove that L1 and L2 are incompatible. By definition of compatibility there must exist a sink n such that l1(n)=X and l2(n) ∈ {S,X} (or vice versa). By definition of projection there must exist a node-slice {n1...ns} of n such that L1(n1)=...=L1(ns)=X. Also there must exist an ancestor n0 of n such that L2(n0) ∈ {S,SIX,X}. From the definition of lock-graph there is a path P1 from a source to n0 on which L2 does not take the value NL.

If P1 intersects the node-slice at ni then L1 and L2 are incompatible since L1(ni)=X which is incompatible with the non null value of L2(ni). Hence the theorem is proved.

Alternatively there is a path P2 from n0 to the sink n which intersects the node-slice at ni. From the definition of lock-graph L1 takes a value in {IX,SIX,X} on all ancestors of ni. In particular L1(n0) ∈ {IX,SIX,X}. Since L2(n0) ∈ {S,SIX,X} we have C(L1(n0),L2(n0))=NO. Q.E.D.

<u>Dynamic lock graphs</u>:

Thus far we have pretended that the lock graph is static. However, examination of Figure 3 suggests otherwise. Areas, files and indices are dynamically created and destroyed, and of course records are continually inserted, updated, and deleted. (If the data base is only read, then there is no need for locking at all.)

The lock protocol for such operations is nicely demonstrated by the implementation of <u>index interval locks</u>. Rather than being forced to lock entire indices or individual records, we would like to be able to lock all records with a certain index value; for example, lock all records in the bank account file with the location field equal to Napa. Therefore, the index is partitioned into lockable key value intervals. Each indexed record "belongs" to a particular index interval and all records in a file with the same field value on an indexed field will belong to the same key value interval (i.e. all Napa accounts will belong to the same interval). This new structure is depicted in Figure 4.

DATA BASE

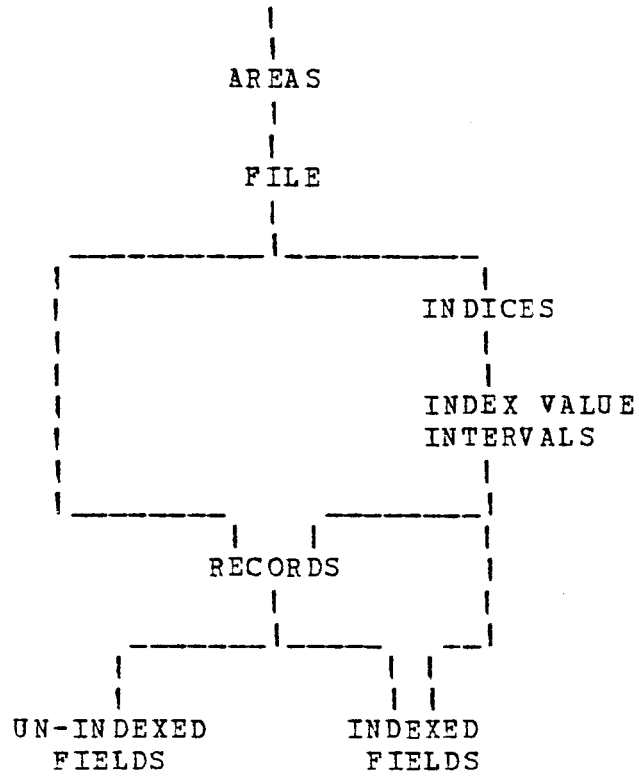


Figure 4. The lock graph with key interval locks.

The only subtle aspect of Figure 4 is the dichotomy between indexed and un-indexed fields and the fact that a key value interval is the parent of both the record and its indexed fields. Since the field value and record identifier (data base key) appear in the index, one can read the field directly (i.e. without touching the record). Hence a key value interval is a parent of the corresponding field values. On the other hand, the index "points" via record identifiers to all records with that value and so is a parent of all records with that field value.

Since Figure 4 defines a DAG, the protocol of the previous section can be used to lock the nodes of the graph. However, it should be extended as follows. When an indexed field is updated, it and its parent record move from one index interval to another. So for example when a Napa account is moved to the St. Helena branch, the account record and its location field "leave" the Napa interval of the location index and "join" the St. Helena index interval. When a new record is inserted it "joins" the interval containing the new field value and also it "joins" the file. Deletion removes the record from the index interval and from the file.

The lock protocol for changing the parents of a node is:

(d) Before moving a node in the lock graph, the node must be implicitly or explicitly granted in X mode in both its old and its new position in the graph. Further, the node must not be moved in such a way as to create a cycle in the graph.

So to carry out the example of this section, to move a Napa bank account to the St. Helena branch one would:

```
lock data base              in mode = IX
lock area containg accounts in mode = IX
lock accounts file          in mode = IX
lock location index         in mode = IX
lock Napa interval          in mode = IX
lock St. Helena interval    in mode = IX
lock record                 in mode = IX
lock field                  in mode = X.
```

Alternatively, one could get an implicit lock on the field by requesting explicit X mode locks on the record and index intervals.

II. DEGREES OF CONSISTENCY:

The data base consists of entities which are known to be structured in certain ways. This structure is best thought of as assertions about the data. Examples of such assertions are:
'Names is an index for Telephone_numbers.'
'The value of Count_of_x gives the number of employees in department x.'

The data base is said to be consistent if it satisfies all its assertions [2]. In some cases, the data base must become temporarily inconsistent in order to transform it to a new consistent state. For example, adding a new employee involves several atomic actions and the updating of several fields. The data base may be inconsistent until all these updates have been completed.

To cope with these temporary inconsistencies, sequences of atomic actions are grouped to form transactions. Transactions are the units of consistency. They are larger atomic actions on the data base which transform it from one consistent state to a new consistent state. Transactions preserve consistency. If some action of a transaction fails then the entire transaction is 'undone' thereby returning the data base to a consistent state. Thus transactions are also the units of recovery. Hardware failure, system error, deadlock, protection violations and program error are each a source of such failure. The system may enforce the consistency assertions and undo a transaction which tries to leave the data base in an inconsistent state.

If transactions are run one at a time then each transaction will see the consistent state left behind by its predecessor. But if several transactions are scheduled concurrently then locking is required to insure that the inputs to each transaction are consistent.

Responsibility for requesting and releasing locks can be either assumed by the user or delegated to the system. User controlled locking results in potentially fewer locks due to the user's knowledge of the semantics of the data. On the other hand, user controlled locking requires difficult and potentially unreliable application programming. Hence the approach taken by some data base systems is to use automatic lock protocols which insure protection from general types of inconsistencies, while still relying on the user to protect himself against other sources of inconsistencies. For example, a system may automatically lock updated records but not records which are read. Such a system prevents lost updates arising from transaction backup. Still, the user should explicitly lock records in a read-update sequence to insure that the read value does not change before the actual update. In other words, a user is guaranteed a limited automatic degree of consistency. This degree of consistency may be system wide or the system may provide options to select it (for instance a lock protocol may be associated with a transaction or with an

entity).

We now present several equivalent definitions of four consistency degrees:

Informal definition of consistency:

An output (write) of a transaction is committed when the transaction abdicates the right to 'undo' the write thereby making the new value available to all other transactions. Outputs are said to be uncommitted or dirty if they are not yet committed by the writer. Concurrent execution raises the problem that reading or writing other transactions' dirty data may yield inconsistent data.

Using this notion of dirty data, the degrees of consistency may be defined as:

Definition 1:

Degree 3: Transaction T sees degree 3 consistency if:
(a) T does not overwrite dirty data of other transactions.
(b) T does not commit any writes until it completes all its writes (i.e. until the end of transaction (EOT)).
(c) T does not read dirty data from other transactions.
(d) Other transactions do not dirty any data read by T before T completes.

Degree 2: Transaction T sees degree 2 consistency if:
(a) T does not overwrite dirty data of other transactions.
(b) T does not commit any writes before EOT.
(c) T does not read dirty data of other transactions.

Degree 1: Transaction T sees degree 1 consistency if:
(a) T does not overwrite dirty data of other transactions.
(b) T does not commit any writes before EOT.

Degree 0: Transaction T sees degree 0 consistency if:
(a) T does not overwrite dirty data of other transactions.

Note that if a transaction sees a high degree of consistency then it also sees all the lower degrees.

These definitions have implications for transaction recovery. Transactions are dichotomized as recoverable transactions which can be undone without affecting other transactions, and unrecoverable transactions which cannot be undone because they have committed data to other transactions and to the external world. Unrecoverable transactions cannot be undone without cascading transaction backup to other transactions and to the external world (e.g. 'unprinting' a message is usually impossible). If the system is to undo individual transactions without cascading backup to other transactions then none of the

transaction's writes can be committed before the end of the transaction. Otherwise some other transaction could further update the entity thereby making it impossible to perform transaction backup without propagating backup to the subsequent transaction.

Degree 0 consistent transactions are unrecoverable because they commit outputs before the end of transaction. If all transactions see at least degree 0 consistency, then any transaction which is at least degree 1 consistent is recoverable because it does not commit writes before the end of the transaction. For this reason, many data base systems require that all transactions see at least degree 1 consistency in order to guarantee that all transactions are recoverable.

Degree 2 consistency isolates a transaction from the uncommitted data of other transactions. With degree 1 consistency a transaction might read uncommitted values which are subsequently updated or are undone.

Degree 3 consistency isolates the transaction from dirty relationships among entities. For example, a degree 2 consistent transaction may read two different (committed) values if it reads the same entity twice. This is because a transaction which updates the entity could begin, update and end in the interval of time between the two reads. More elaborate kinds of anomalies due to concurrency are possible if one updates an entity after reading it or if more than one entity is involved (see example below). Degree 3 consistency completely isolates the transaction from inconsistencies due to concurrency.

To give an example which demonstrates the application of these several degrees of consistency, imagine a process control system in which some transaction is dedicated to reading a gauge and periodically writing batches of values into a list. Each gauge reading is an individual entity. For performance reasons, this transaction sees degree 0 consistency, committing all gauge readings as soon as they enter the data base. This transaction is not recoverable (can't be undone). A second transaction is run periodically which reads all the recent gauge readings, computes a mean and variance and writes these computed values as entities in the data base. Since we want these two values to be consistent with one another, they must be committed together (i.e. one cannot commit the first before the second is written). This allows transaction undo in the case that it aborts after writing only one of the two values. Hence this statistical summary transaction should see degree 1. A third transaction which reads the mean and writes it on a display sees degree 2 consistency. It will not read a mean which might be 'undone' by a backup. Another transaction which reads both the mean and the variance must see degree 3 consistency to insure that the mean and variance derive from the same computation (i.e. the same run which wrote the mean also wrote the variance).

Lock protocol definition of consistency:

Whether an instantiation of a transaction sees degree 0, 1, 2 or 3 consistency depends on the actions of other concurrent transactions. Lock protocols are used by a transaction to guarantee itself a certain degree of consistency independent of the behavior of other transactions (so long as all transactions at least observe the degree 0 protocol).

The degrees of consistency can be operationally defined by the lock protocols which produce them. A transaction locks its inputs to guarantee their consistency and locks its outputs to mark them as dirty (uncommitted). Degrees 0, 1 and 2 are important because of the efficiencies implicit in these protocols. Obviously, it is cheaper to lock less.

Locks are dichotomized as share mode locks which allow multiple readers of the same entity and exclusive mode locks which reserve exclusive access to an entity. Locks may also be characterized by their duration: locks held for the duration of a single action are called short duration locks while locks held to the end of the transaction are called long duration locks. Short duration locks are used to mark or test for dirty data for the duration of an action rather than for the duration of the transaction.

The lock protocols are:

Definition 2:

Degree 3: transaction T observes degree 3 lock protocol if:
(a) T sets a long exclusive lock on any data it dirties.
(b) T sets a long share lock on any data it reads.

Degree 2: transaction T observes degree 2 lock protocol if:
(a) T sets a long exclusive lock on any data it dirties.
(b) T sets a (posibly short) share lock on any data it reads.

Degree 1: transaction T observes degree 1 lock protocol if:
(a) T sets a long exclusive lock on any data it dirties.

Degree 0: transaction T observes degree 0 lock protocol if:
(a) T sets a (possibly short) exclusive lock on any data it dirties.

The lock protocol definitions can be stated more tersly with the introduction of the following notation. A transaction is well formed with respect to writes (reads) if it always locks an entity in exclusive (shared or exclusive) mode before writing (reading) it. The transaction is well formed if it is well formed with respect to reads and writes.

A transaction is two phase (with respect to reads or updates) if it does not (share or exclusive) lock an entity after unlocking some entity. A two phase transaction has a growing phase during

which it acquires locks and a shrinking phase during which it releases locks.

Definition 2 is too restrictive in the sense that consistency will not require that a transaction hold all locks to the EOT (i.e. the EOT is the shrinking phase); rather the constraint that the transaction be two phase is adaquate to insure consistency. On the other hand, once a transaction unlocks an updated entity, it has committed that entity and so cannot be undone without cascading backup to any transactions which may have subsequently read the entity. For that reason, the shrinking phase is usually deferred to the end of the transaction so that the transaction is always recoverable and so that all updates are committed together. The lock protocols can be redefined as:

Definition 2':

Degree 3: T is well formed
and T is two phase.

Degree 2: T is well formed
and T is two phase with respect to writes.

Degree 1: T is well formed with respect to writes
and T is two phase with respect to writes.

Degree 0: T is well formed with respect to writes.

All transactions are <u>required</u> to observe the degree 0 locking protocol so that they do not update the uncommitted updates of others. Degrees 1, 2 and 3 provide increasing system-guaranteed consistency.

<u>Consistency of schedules</u>:

The definition of what it means for a transaction to see a degree of consistency was originally given in terms of dirty data. In order to make the notion of dirty data explicit it is necessary to consider the execution of a transaction in the context of a set of concurrently executing transactions. To do this we introduce the notion of a schedule for a set of transactions. A schedule can be thought of as a history or audit trail of the actions performed by the set of transactions. Gven a schedule the notion of a particular entity being dirtied by a particular transaction is made explicit and hence the notion of seeing a certain degree of consistency is formalized. These notions may then be used to connect the various definitions of consistency and show their equivalence.

The system directly supports <u>entities</u> and <u>actions</u>. Actions are categorized as <u>begin</u> actions, <u>end</u> actions, <u>share lock</u> actions, <u>exclusive lock</u> actions, <u>unlock</u> actions, <u>read</u> actions, and <u>write</u> actions. An end action is presumed to unlock any locks held by

the transaction but not explicitly unlocked by the transaction. For the purposes of the following definitions, share lock actions and their corresponding unlock actions are additionally considered to be read actions and exclusive lock actions and their corresponding unlock actions are additionally considered to be write actions.

A <u>transaction</u> is any sequence of actions beginning with a begin action and ending with an end action and not containing other begin or end actions.

Any (sequence preserving) merging of the actions of a set of transactions into a single sequence is called a <u>schedule</u> for the set of transactions.

A schedule is a history of the order in which actions are executed (it does not record actions which are undone due to backup). The simplest schedules run all actions of one transaction and then all actions of another transaction,... Such one-transaction-at-a-time schedules are called <u>serial</u> because they have no concurrency among transactions. Clearly, a serial schedule has no concurrency induced inconsistency and no transaction sees dirty data.

Locking constrains the set of allowed schedules. In particular, a schedule is <u>legal</u> only if it does not schedule a lock action on an entity for one transaction when that entity is already locked by some other transaction in a conflicting mode.

An initial state and a schedule completely define the system's behavior. At each step of the schedule one can deduce which entity values have been committed and which are dirty: if locking is used, updated data is dirty until it is unlocked.

Since a schedule makes the definition of dirty data explicit, one can apply Definition 1 to define consistent schedules:

Definition 3:

A transaction <u>runs at degree 0 (1, 2 or 3) consistency in schedule S</u> if T sees degree 0 (1, 2 or 3) consistency in S.
If all transactions run at degree 0 (1,2 or 3) consistency in schedule S then S is said to be a <u>degree 0 (1, 2 or 3) consistent schedule</u>.

Given these definitions one can show:

<u>Assertion 1</u>:
(a) If each transaction observes the degree 0 (1, 2 or 3) lock protocol (Definition 2) then any legal schedule is degree 0 (1, 2 or 3) consistent (Definition 3) (i.e, each transaction sees degree 0 (1, 2 or 3) consistency in the sense of Definition 1).
(b) Unless transaction T observes the degree 1 (2 or 3) lock protocol then it is possible to define another transaction T'

which does observe the degree 1 (2 or 3) lock protocol such that T and T' have a legal schedule S but T does not run at degree 1 (2 or 3) consistency in S.

Assertion 1 says that if a transaction observes the lock protocol definition of consistency (Definition 2) then it is assured of the informal definition of consistency based on committed and dirty data (Definition 1). Unless a transaction actually sets the locks prescribed by degree 1 (2 or 3) consistency one can construct transaction mixes and schedules which will cause the transaction to run at (see) a lower degree of consistency. However, in particular cases such transaction mixes may never occur due to the structure or use of the system. In these cases an apparently low degree of locking may actually provide degree 3 consistency. For example, a data base reorganization usually need do no locking since it is run as an off-line utility which is never run concurrently with other transactions.

Assertion 2:
If each transaction in a set of transactions at least observes the degree 0 lock protocol and if transaction T observes the degree 1 (2 or 3) lock protocol then T runs at degree 1 (2 or 3) consistency (Definitions 1, 3) in any legal schedule for the set of transactions.

Assertion 2 says that each transaction can choose its degree of consistency so long as all transactions observe at least degree 0 protocols. Of course the outputs of degree 0, 1 or 2 consistent transactions may be degree 0, 1 or 2 consistent (i.e. inconsistent) because they were computed with potentially inconsistent inputs. One can imagine that each data entity is tagged with the degree of consistency of its writer. A transaction must beware of reading entities tagged with degrees lower than the degree of the transaction.

Dependencies among transactions:

One transaction is said to depend on another if the first takes some of its inputs from the second. The notion of dependency is defined differently for each degree of consistency. These dependency relations are completely defined by a schedule and can be useful in discussing consistency and recovery.

Each schedule defines three relations: <, << and <<< on the set of transactions as follows. Suppose that transaction T performs action a on entity e at some step in the schedule and that transaction T' performs action a' on entity e at a later step in the schedule. Further suppose that T does not equal T'. Then:

T <<< T' if a is a write action and a' is a write action
or a is a write action and a' is a read action
or a is a read action and a' is a write action

T << T' if a is a write action and a' is a write action

or a is a write action and a' is a read action

T < T' if a is a write action and a' is a write action

The following table is a notationally convenient way of seeing these definitions:

<<< : W->W | W->R | R->W

<< : W->W | W->R

< : W->W

meaning that (for example) T <<< T' if T writes (W) something later read (R) by T' or written (W) by T' or T reads (R) something later written (W) by T'.

Let <* be the transitive closure of <, then define:
BEFORE1(T) = {T'| T' <* T}
AFTER1(T) = {T'| T <* T'}.

The sets BEFORE2, AFTER2, BEFORE3 and AFTER3 are defined analogously for << and <<<.

The obvious interpretation for this is that each BEFORE set is the set of transactions which contribute inputs to T and each AFTER set is the set of transactions which take their inputs from T (where the ordering only considers dependencies induced by the corresponding consistency degree).

If some transaction is both before T and after T in some schedule then no serial schedule could give such results. In this case concurrency has introduced inconsistency. On the other hand, if all relevant transactions are either before or after T (but not both) then T will see a consistent state (of the corresponding degree). If all transactions dichotomize others in this way then the relation <* (<<* or <<<*) will be a partial order and the whole schedule will give degree 1 (2 or 3) consistency. This can be strengthened to:

<u>Assertion 3</u>:
A schedule is degree 1 (2 or 3) consistent if and only if the relation <* (<<* or <<<*) is a partial order.

The <, << and <<< relations are variants of the dependency sets introduced in [2]. In that paper only degree 3 consistency is introduced and Assertion 3 was proved for that case. In particular such a schedule is equivalent to the serial schedule obtained by running the transactions one at a time in <<< order. The proofs of [2] generalize fairly easily to handle assertion 1 in the case of degree 1 or 2 consistency.

Consider the following example:
T1 LOCK A

```
T1 READ    A
T1 UNLOCK  A
T2 LOCK    A
T2 WRITE   A
T2 LOCK    B
T2 WRITE   B
T2 UNLOCK  A
T2 UNLOCK  B
T1 LOCK    B
T1 WRITE   B
T1 UNLOCK  B
```

In this schedule T2 gives B to T1 and T2 updates A after T1 reads A so T2<T1, T2<<T1, T2<<<T1 and T1<<<T2. The schedule is degree 2 consistent but not degree 3 consistent. It runs T1 at degree 2 consistency and T2 at degree 3 consitency.

It would be nice to define a transaction to see degree 1 (2 or 3) consistency if and only if the BEFORE and AFTER sets are disjoint in some schedule. However, this is not restrictive enough, rather one must require that the before and after sets be disjoint in <u>all</u> schedules in order to state Definition 1 in terms of dependencies. Further, there seems to be no natural way to define the dependencies of degree 0 consistency. Hence the principal application of the dependency definition is as a proof technique and for discussing schedules and recovery issues.

<u>Relationship to transaction backup and system recovery</u>:

As mentioned previously, system wide degree 1 consistency allows transaction backup and system recovery without lost updates. (i.e. without affecting updates of transactions which are not being backed up). The transaction is unrecoverable after its first commit of an update (unlock) and so although degree 1 does not require it, the shrinking phase is usually deferred to the end of transaction so that the transaction is recoverable.

Given any current state and a time ordered log of the updates of transactions, one can return to a consistent state by un-doing any incomplete transactions (uncommitted updates). Given a checkpoint at time T0 and a log which records old and new values of entities up to time T0+e, one can construct the most recent consistent state by undoing all updates which were made before time T0 but were not yet committed at time T0+e; and by redoing all updates which were made and committed in the interval T0 to T0+e. If the schedule (log) is degree 0 consistent then the actions can be re-done LOG order (skipping uncommitted updates). If the schedule (log) is degree 1 consistent then the actions can be sorted by transaction in <* order and recovery performed with the sorted log. The outcome of this process will be a state reflecting all the changes made by all transactions which completed before the log stopped.

However, degree 1 consistent transactions may read uncommitted

(dirty) data. Transaction and system recovery may undo uncommitted updates. So if the degree 1 consistent transaction is re-run (i.e. re-executed by the system) in the absence of the undone transactions it may produce entirely different results than would be obtained if the transaction were blindly re-done (from the updates recorded in the log). If the system is degree 2 consistent then no transaction reads uncommitted data. So if the completed transactions are re-done in log order but in the absence of some undone (incomplete) transactions they will give exactly the same results as were obtained in the presence of the undone transactions. In particular, if the transactions were re-run in the order specified by the log but in the absence of the undone transactions the same consistent state would result.

ISSUE	DEGREE 0	DEGREE 1	DEGREE 2	DEGREE 3
COMMITTED DATA	WRITES ARE COMMITTED IMMEDIATELY	WRITES ARE COMMITTED AT EOT	SAME	SAME
DIRTY DATA	YOU DON'T UPDATE DIRTY DATA	0 AND NO ONE ELSE UPDATES YOUR DIRTY DATA	0,1 AND YOU DON'T READ DIRTY DATA	0,1,2 AND NO ONE ELSE DIRTIES DATA YOU READ
LOCK PROTOCOL	SET SHORT EXCL. LOCKS ON ANY DATA YOU WRITE	SET LONG EXCL. LOCKS ON ANY DATA YOU WRITE	1 AND SET SHORT SHARE LOCKS ON ANY DATA YOU READ	1 AND SET LONG SHARE LOCKS ON ANY DATA YOU READ
TRANSACTION STRUCTURE see [1]	WELL FORMED WRT WRITES	(WELL FORMED AND 2 PHASE) WRT WRITES	WELL FORMED (AND 2 PHASE WRT WRITES)	WELL FORMED AND TWO PHASE
CONCURRENCY	GREATEST: ONLY WAIT FOR SHORT WRITE LOCKS	GREAT: ONLY WAIT FOR WRITE LOCKS	MEDIUM: ALSO WAIT FOR READ LOCKS	LOWEST: ANY DATA TOUCHED IS LOCKED TO EOT
OVERHEAD	LEAST: ONLY SET SHORT WRITE LOCKS	SMALL: ONLY SET WRITE LOCKS	MEDIUM: SET BOTH KINDS OF LOCKS BUT NEED NOT STORE SHORT LOCKS	HIGHEST: SET AND STORE BOTH KINDS OF LOCKS
TRANSACTION BACKUP	CAN NOT UNDO WITHOUT CASCADING TO OTHERS	UN-DO INCOMPLETE TRANSACTIONS IN ANY ORDER	SAME	SAME
PROTECTION PROVIDED	LETS OTHERS RUN HIGHER CONSISTENCY	0 AND CAN'T LOSE WRITES	0,1 AND CAN'T READ BAD DATA ITEMS	0,1,2 AND CAN'T READ BAD DATA RELATIONSHIPS
SYSTEM RECOVERY TECHNIQUE	APPLY LOG IN ORDER OF ARRIVAL	APPLY LOG IN < ORDER	SAME AS 1: BUT RESULT IS SAME AS SOME SCHEDULE	RERUN TRANSACTIONS IN <<< ORDER
DEPENDENCIES	none	W->W	W->W W->R	W->W W->R R->W
ORDERING	NONE	< IS AN ORDERING OF THE TRANS-ACTIONS	<< IS AN ORDERING OF THE TRANS-ACTIONS	<<< IS AN ORDERING OF THE TRANS-ACTIONS

Table 2. Summary of consistency degrees.

III. LOCK HIERARCHIES AND DEGREES OF CONSISTENCY IN EXISTING SYSTEMS:

IMS/VS with the program isolation feature [3] has a two level lock hierarchy: segment types (sets of records), and segment instances (records) within a segment type. Segment types may be locked in EXCLUSIVE (E) mode (which corresponds to our exclusive (X) mode) or in EXPRESS READ (R), RETRIEVE (G), or UPDATE (U) (each of which correspond to our notion of intention (I) mode) [3 page 3.18-3.27]. Segment instances can be locked in share or exclusive mode. Segment type locks are requested at transaction initiation, usually in intention mode. Segment instance locks are dynamically set as the transaction proceeds. In addition IMS/VS has user controlled share locks on segment instances (the *Q option) which allow other read requests but not other *Q or exclusive requests. IMS/VS has no notion of S or SIX locks on segment types (which would allow a scan of all members of a segment type concurrent with other readers but without the overhead of locking each segment instance). Since IMS/VS does not support S mode on segment types one need not distinguish the two intention modes IS and IX (see the section introducing IS and IX modes). In general, IMS/VS has a notion of intention mode and does implicit locking but does not recognize all the modes described here. It uses a static two level lock tree.

IMS/VS with the program isolation feature basically provides degree 2 consistency. However degree 1 consistency can be obtained on a segment type basis in a PCB (view) by specifying the EXPRESS READ option for that segment. Similarly degree 3 consistency can be obtained by specifying the EXCLUSIVE option. IMS/VS also has the user controlled share locks discussed above which a program can request on selected segment instances to obtain additional consistency over the degree 1 or 2 consistency provided by the system.

IMS/VS without the program isolation feature (and also the previous version of IMS namely IMS/2) doesn't have a lock hierarchy since locking is done only on a segment type basis. It provides degree 1 consistency with degree 3 consistency obtainable for a segment type in a view by specifying the EXCLUSIVE option. User controlled locking is also provided on a limited basis via the HOLD option.

DMS 1100 has a two level lock hierarchy [4]: areas and pages within areas. Areas may be locked in one of seven modes when they are OPENed: EXCLUSIVE RETRIEVAL (which corresponds to our notion of exclusive mode), PROTECTED UPDATE (which corresponds to our notion of share and intention exclusive mode), PROTECTED RETRIEVAL (which we call share mode), UPDATE (which corresponds to our intention exclusive mode), and RETRIEVAL (which is our intention share mode). Given this transliteration, the compatibility matrix displayed in Table 1 is identical to the compatibility matrix of DMS 1100 [4, page 3.59]. However, DMS 1100 sets only exclusive locks on pages within areas (short term share locks are invisibly

set during internal pointer following). Further, even if a transaction locks an area in exclusive mode, DMS 1100 continues to set exclusive locks (and internal share locks) on the pages in the area, despite the fact that an exclusive lock on an area precludes reads or updates of the area by other transactions. Similar observations apply to the DMS 1100 implementation of S and SIX modes. In general, DMS 1100 recognizes all the modes described here and uses intention modes to detect conflicts but does not utilize implicit locking. It uses a static two level lock tree.

DMS 1100 provides level 2 consistency by setting exclusive locks on the modified pages and and a temporary lock on the page corresponding to the page which is "current of run unit". The temporary lock is released when the "current of run unit" is moved. In addition a run-unit can obtain additional locks via an explicit KEEP command.

The ideas presented were developed in the process of designing and implementing an experimental data base system at the IBM San Jose Research Laboratory. (We wish to emphasize that this system is a vehicle for research in data base architecture, and does not indicate plans for future IBM products.) A subsystem which provides the modes of locks herein described, plus the necessary logic to schedule requests and conversions, and to detect and resolve deadlocks has been implemented as one component of the data manager. The lock subsystem is in turn used by the data manager to automatically lock the nodes of its lock graph (see Figure 5). Users can be unaware of these lock protocols beyond the verbs "begin transaction" and "end transaction".

The data base is broken into several storage areas. Each area contains a set of relations (files), their indices, and their tuples(records) along with a catalog of the area. Each tuple has a unique tuple identifier (data base key) which can be used to quickly (directly) address the tuple. Each tuple identifier maps to a set of field values. All tuples are stored together in an area-wide heap to allow physical clustering of tuples from different relations. The unused slots in this heap are represented by an area-wide pool of free tuple identifiers (i.e. identifiers not allocated to any relation). Each tuple "belongs" to a unique relation, and all tuples in a relation have the same number and type of fields. One may construct an index on any subset of the fields of a relation. Tuple identifiers give fast direct access to tuples, while indices give fast associative access to field values and to their corresponding tuples. Each key value in an index is made a lockable object in order to solve the problem of "phantoms" [1] without locking the entire index. We do not explicitly lock individual fields or whole indices so those nodes appear in Figure 5 only for pedagogical reasons. Figure 5 gives only the "logical" lock graph, there is also a graph for physical page locks and for other low level resources.

As can be seen, Figure 5 is not a tree. Heavy use is made of the techniques mentioned in the section on locking DAG's. For

example, one can read via tuple identifier without setting any index locks but to lock a field for update its tuple identifier and the old and new index key values covering the updated field must be locked in X mode. Further, the tree is not static, since data base keys are dynamically allocated to relations; field values dynamically enter, move around in, and leave index value intervals when records are inserted, updated and deleted; relations and indices are dynamically created and destroyed within areas; and areas are dynamically allocated. The implementation of such operations observes the lock protocol presented in the section on dynamic graphs: When a node changes parents, all old and new parents must be held (explicitly or implicitly) in intention exclusive mode and the node to be moved must be held in exclusive mode.

The described system supports concurrently consistency degrees 1,2 and 3 which can be specified on a transaction basis. In addition share locks on individual tuples can be acquired by the user.

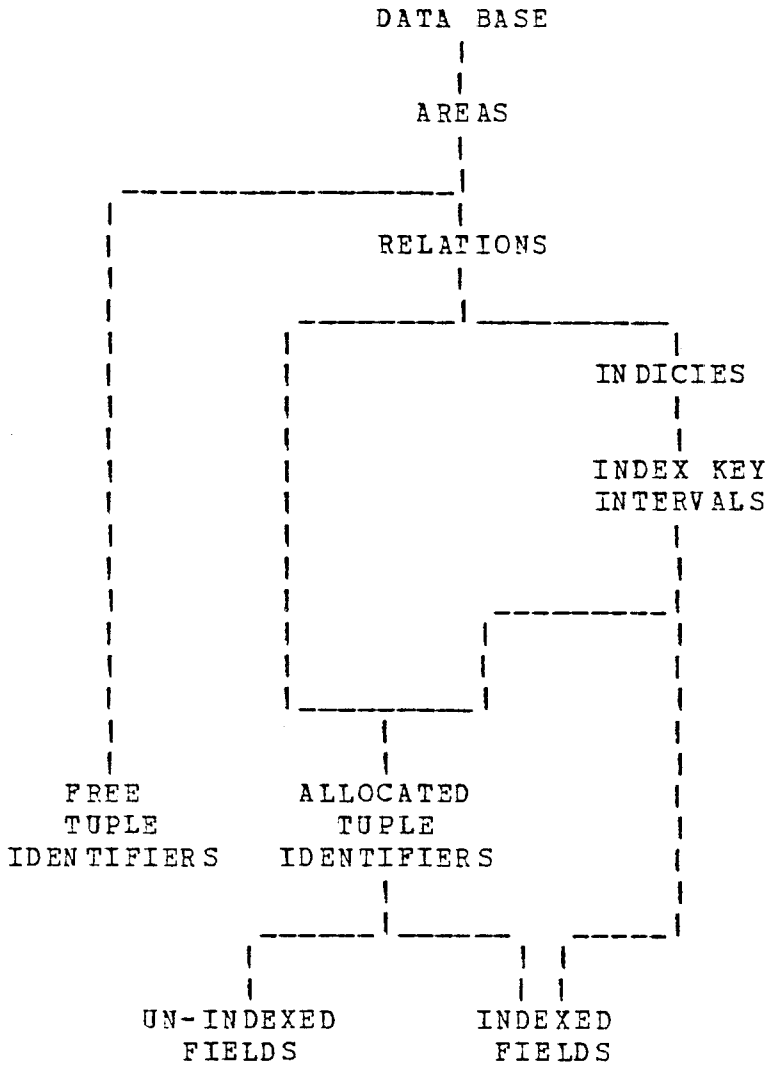


Figure 5. A lock graph.

ACKNOWLEDGMENT

We gratefully acknowledge many helpful discussions with Phil Macri, Jim Mehl and Brad Wade on how locking works in existing systems and how these results might be better presented. We are especially indebted to Paul McJones in this regard.

REFERENCES

[1] J.N. Gray, R.A. Lorie, G.R. Putzolu, "Granularity of Locks in a Shared Data Base," Proceedings of the International Conference on Very Large Data Bases, Boston, Mass., September 1975.

[2] K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger, On the Notions of Consistency and Predicate Locks, Technical Report RJ.1487, IBM Research Laboratory, San Jose, Ca., Nov. 1974.

[3] Information Management System Virtual Storage (IMS/VS). System Application Design Guide, Form No. SH20-9025-2, IBM Corp., 1975.

[4] UNIVAC 1100 Series Data Management System (DMS 1100). ANSI COBOL Field Data Manipulation Language. Order No. UP7908-2, Sperry Rand Corp., May 1973.

Concurrency Control in Distributed Database Systems

PHILIP A. BERNSTEIN AND NATHAN GOODMAN

Computer Corporation of America, Cambridge, Massachusetts 02139

In this paper we survey, consolidate, and present the state of the art in distributed database concurrency control. The heart of our analysis is a decomposition of the concurrency control problem into two major subproblems: read-write and write-write synchronization. We describe a series of synchronization techniques for solving each subproblem and show how to combine these techniques into algorithms for solving the entire concurrency control problem. Such algorithms are called "concurrency control methods." We describe 48 principal methods, including all practical algorithms that have appeared in the literature plus several new ones. We concentrate on the structure and correctness of concurrency control algorithms. Issues of performance are given only secondary treatment.

Keywords and Phrases: concurrency control, deadlock, distributed database management systems, locking, serializability, synchronization, timestamp ordering, timestamps, two-phase commit, two-phase locking

CR Categories: 4.33, 4.35

INTRODUCTION

The Concurrency Control Problem

Concurrency control is the activity of coordinating concurrent accesses to a database in a multiuser database management system (DBMS). Concurrency control permits users to access a database in a multiprogrammed fashion while preserving the illusion that each user is executing alone on a dedicated system. The main technical difficulty in attaining this goal is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another. The concurrency control problem is exacerbated in a distributed DBMS (DDBMS) because (1) users may access data stored in many different computers in a distributed system, and (2) a concurrency control mechanism at one computer cannot instantaneously know about interactions at other computers.

Concurrency control has been actively investigated for the past several years, and the problem for nondistributed DBMSs is well understood. A broad mathematical theory has been developed to analyze the problem, and one approach, called *two-phase locking*, has been accepted as a standard solution. Current research on nondistributed concurrency control is focused on evolutionary improvements to two-phase locking, detailed performance analysis and optimization, and extensions to the mathematical theory.

Distributed concurrency control, by contrast, is in a state of extreme turbulence. More than 20 concurrency control algorithms have been proposed for DDBMSs, and several have been, or are being, implemented. These algorithms are usually complex, hard to understand, and difficult to prove correct (indeed, many are incorrect). Because they are described in different terminologies and make different assumptions

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1981 ACM 0010-4892/81/0600-0185 $00.75

CONTENTS

INTRODUCTION
The Concurrency Control Problem
Examples of Concurrency Control Anomalies
Comparison to Mutual Exclusion Problems
1. TRANSACTION-PROCESSING MODEL
1.1 Preliminary Definitions and DDBMS Architecture
1.2 Centralized Transaction-Processing Model
1.3 Distributed Transaction-Processing Model
2 DECOMPOSITION OF THE CONCURRENCY CONTROL PROBLEM
2 1 Serializability
2.2 A Paradigm for Concurrency Control
3. SYNCHRONIZATION TECHNIQUES BASED ON TWO-PHASE LOCKING
3.1 Basic 2PL Implementation
3.2 Primary Copy 2PL
3.3 Voting 2PL
3.4 Centralized 2PL
3.5 Deadlock Detection and Prevention
4 SYNCHRONIZATION TECHNIQUES BASED ON TIMESTAMP ORDERING
4.1 Basic T/O Implementation
4.2 The Thomas Write Rule
4.3 Multiversion T/O
4.4 Conservative T/O
4 5 Timestamp Management
5 INTEGRATED CONCURRENCY CONTROL METHODS
5 1 Pure 2PL Methods
5.2 Pure T/O Methods
5.3 Mixed 2PL and T/O Methods
6. CONCLUSION
APPENDIX. OTHER CONCURRENCY CONTROL METHODS
A1. Certifiers
A2. Thomas' Majority Consensus Algorithm
A3. Ellis' Ring Algorithm
ACKNOWLEDGMENT
REFERENCES

about the underlying DDBMS environment, it is difficult to compare the many proposed algorithms, even in qualitative terms. Naturally each author proclaims his or her approach as best, but there is little compelling evidence to support the claims.

To survey the state of the art, we introduce a standard terminology for describing DDBMS concurrency control algorithms and a standard model for the DDBMS environment. For analysis purposes we decompose the concurrency control problem into two major subproblems, called read-write and write-write synchronization. Every concurrency control algorithm must include a subalgorithm to solve each subproblem. The first step toward understanding a concurrency control algorithm is to isolate the subalgorithm employed for each subproblem.

After studying the large number of proposed algorithms, we find that they are compositions of only a few subalgorithms. In fact, the subalgorithms used by all practical DDBMS concurrency control algorithms are variations of just two basic techniques: two-phase locking and timestamp ordering; thus the state of the art is far more coherent than a review of the literature would seem to indicate.

Examples of Concurrency Control Anomalies

The goal of concurrency control is to prevent interference among users who are simultaneously accessing a database. Let us illustrate the problem by presenting two "canonical" examples of interuser interference. Both are examples of an on-line electronic funds transfer system accessed via remote *automated teller machines* (*ATMs*). In response to customer requests, ATMs retrieve data from a database, perform computations, and store results back into the database.

Anomaly 1: Lost Updates. Suppose two customers simultaneously try to deposit money into the same account. In the absence of concurrency control, these two activities could interfere (see Figure 1). The two ATMs handling the two customers could read the account balance at approximately the same time, compute new balances in parallel, and then store the new balances back into the database. The net effect is incorrect: although two customers deposited money, the database only reflects one activity; the other deposit is lost by the system.

Anomaly 2: Inconsistent Retrievals. Suppose two customers simultaneously execute the following transactions.

Customer 1: Move $1,000,000 from Acme Corporation's savings account to its checking account.
Customer 2: Print Acme Corporation's total balance in savings and checking.

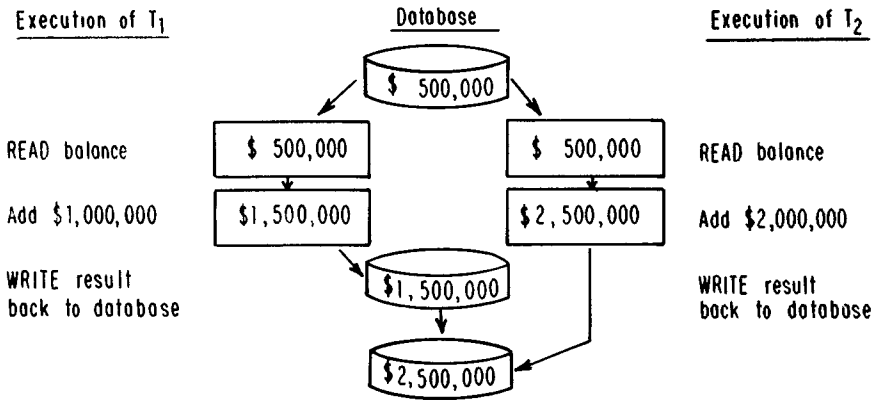


Figure 1. Lost update anomaly.

In the absence of concurrency control these two transactions could interfere (see Figure 2). The first transaction might read the savings account balance, subtract $1,000,000, and store the result back in the database. Then the second transaction might read the savings and checking account balances and print the total. Then the first transaction might finish the funds transfer by reading the checking account balance, adding $1,000,000, and finally storing the result in the database. Unlike Anomaly 1, the final values placed into the database by this execution are correct. Still, the execution is incorrect because the balance printed by Customer 2 is $1,000,000 short.

These two examples do not exhaust all possible ways in which concurrent users can interfere. However, these examples are typical of the concurrency control problems that arise in DBMSs.

Comparison to Mutual Exclusion Problems

The problem of database concurrency control is similar in some respects to that of mutual exclusion in operating systems. The latter problem is concerned with coordinating access by concurrent processes to system resources such as memory, I/O devices, and CPU. Many solution techniques have been developed, including locks, semaphores, monitors, and serializers [BRIN73, DIJK71, HEWI74, HOAR74].

The concurrency control and mutual exclusion problems are similar in that both are concerned with controlling concurrent access to shared resources. However, control schemes that work for one do not necessarily work for the other, as illustrated by the following example. Suppose processes P_1 and P_2 require access to resources R_1 and R_2 at different points in their execution. In an operating system, the following interleaved execution of these processes is perfectly acceptable: P_1 uses R_1, P_2 uses R_1, P_2 uses R_2, P_1 uses R_2. In a database, however, this execution is not always acceptable. Assume, for example, that P_2 transfers funds by debiting one account (R_1), then crediting another (R_2). If P_2 checks both balances, it will see R_1 after it has been debited, but see R_2 *before* it has been credited. Other differences between concurrency control and mutual exclusion are discussed in CHAM74.

1. TRANSACTION-PROCESSING MODEL

To understand how a concurrency control algorithm operates, one must understand how the algorithm fits into an overall DDBMS. In this section we present a simple model of a DDBMS, emphasizing how the DDBMS processes user interactions. Later we explain how concurrency control algorithms operate in the context of this model.

1.1 Preliminary Definitions and DDBMS Architecture

A distributed database management system (DDBMS) is a collection of sites interconnected by a network [DEPP76,

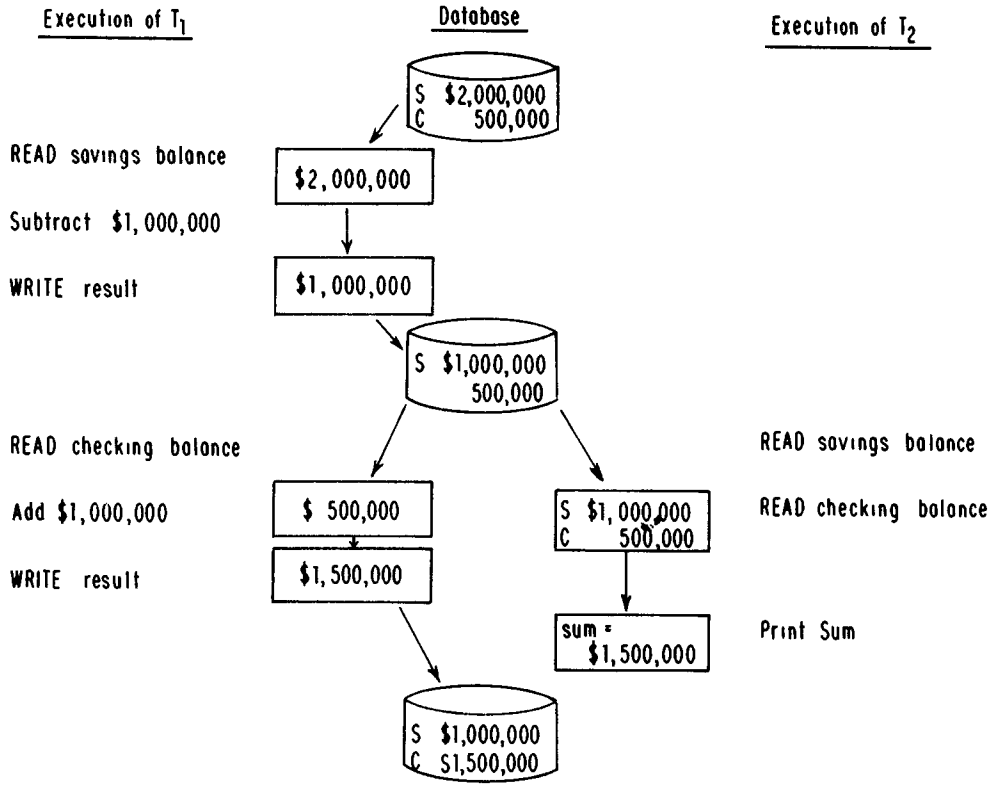


Figure 2. Inconsistent retrieval anomaly.

ROTH77]. Each *site* is a computer running one or both of the following software modules: a transaction manager (TM) or a data manager (DM). *TMs* supervise interactions between users and the DDBMS while *DMs* manage the actual database. A *network* is a computer-to-computer communication system. The network is assumed to be perfectly reliable: if site A sends a message to site B, site B is guaranteed to receive the message without error. In addition, we assume that between any pair of sites the network delivers messages in the order they were sent.

From a user's perspective, a *database* consists of a collection of *logical data items*, denoted X, Y, Z. We leave the granularity of logical data items unspecified; in practice, they may be files, records, etc. A *logical database state* is an assignment of values to the logical data items composing a database. Each logical data item may be stored at any DM in the system or redundantly at several DMs. A stored copy of a logical data item is called a *stored data item*. (When no confusion is possible, we use the term *data item* for stored data item.) The stored copies of logical data item X are denoted $x_1, \ldots, x_m$. We typically use x to denote an arbitrary stored data item. A *stored database state* is an assignment of values to the stored data items in a database.

Users interact with the DDBMS by executing *transactions*. Transactions may be on-line queries expressed in a self-contained query language, or application programs written in a general-purpose programming language. The concurrency control algorithms we study pay no attention to the computations performed by transactions. Instead, these algorithms make all of their decisions on the basis of the data items a transaction reads and writes, and so details of the form of transactions are unimportant in our analysis. However we do assume that transactions represent complete and correct computations; each transaction, if ex-

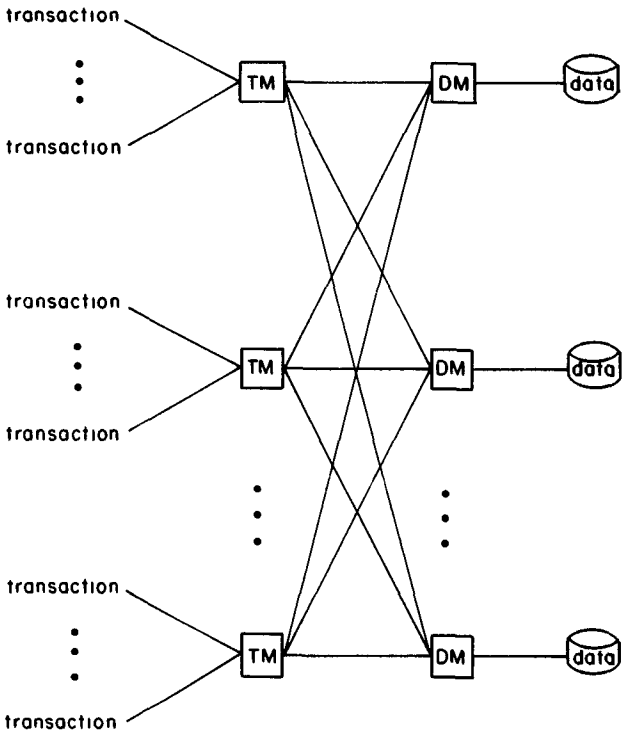


Figure 3. DDBMS system architecture.

ecuted alone on an initially consistent database, would terminate, produce correct results, and leave the database consistent. The *logical readset* (correspondingly, *writeset*) of a transaction is the set of logical data items the transaction reads (or writes). Similarly, *stored readsets* and *stored writesets* are the stored data items that a transaction reads and writes.

The correctness of a concurrency control algorithm is defined relative to users' expectations regarding transaction execution. There are two correctness criteria: (1) users expect that each transaction submitted to the system will eventually be executed; (2) users expect the computation performed by each transaction to be the same whether it executes alone in a dedicated system or in parallel with other transactions in a multiprogrammed system. Realizing this expectation is the principal issue in concurrency control.

A DDBMS contains four components (see Figure 3): transactions, TMs, DMs, and data. Transactions communicate with TMs, TMs communicate with DMs, and DMs manage the data. (TMs do not communicate with other TMs, nor do DMs communicate with other DMs.)

TMs supervise transactions. Each transaction executed in the DDBMS is supervised by a *single* TM, meaning that the transaction issues all of its database operations to that TM. Any distributed computation that is needed to execute the transaction is managed by the TM.

Four operations are defined at the transaction-TM interface. READ(X) returns the value of X (a logical data item) in the current logical database state. WRITE(X, new-value) creates a new logical database state in which X has the specified new value. Since transactions are assumed to represent complete computations, we use BEGIN and END operations to bracket transaction executions.

DMs manage the stored database, functioning as backend database processors. In response to commands from transactions, TMs issue commands to DMs specifying stored data items to be read or written. The details of the TM-DM interface constitute

the core of our transaction-processing model and are discussed in Sections 1.2 and 1.3. Section 1.2 describes the TM–DM interaction in a centralized database environment, and Section 1.3 extends the discussion to a distributed database setting.

1.2 Centralized Transaction-Processing Model

A centralized DBMS consists of one TM and one DM executing at one site. A transaction T accesses the DBMS by issuing BEGIN, READ, WRITE, and END operations, which are processed as follows.

BEGIN: The TM initializes for T a *private workspace* that functions as a temporary buffer for values read from and written into the database.

READ(X): The TM looks for a copy of X in T's private workspace. If the copy exists, its value is returned to T. Otherwise the TM issues *dm-read*(x) to the DM to retrieve a copy of X from the database, gives the retrieved value to T, and puts it into T's private workspace.

WRITE(X, new-value): The TM again checks the private workspace for a copy of X. If it finds one, the value is updated to new-value; otherwise a copy of X with the new value is created in the workspace. The new value of X is *not* stored in the database at this time.

END: The TM issues *dm-write*(x) for each logical data item X updated by T. Each dm-write(x) requests that the DM update the value of X in the stored database to the value of X in T's local workspace. When all dm-writes are processed, T is finished executing, and its private workspace is discarded.

The DBMS may *restart* T any time before a dm-write has been processed. The effect of restarting T is to obliterate its private workspace and to reexecute T from the beginning. As we will see, many concurrency control algorithms use transaction restarts as a tactic for attaining correct executions. However, once a single dm-write has been processed, T cannot be restarted; each dm-write permanently installs an update into the database, and we cannot permit the database to reflect partial effects of transactions.

A DBMS can avoid such partial results by having the property of *atomic commitment*, which requires that either all of a transaction's dm-writes are processed or none are. The "standard" implementation of atomic commitment is a procedure called *two-phase commit* [LAMP76, GRAY78].[1] Suppose T is updating data items X and Y. When T issues its END, the *first phase* of two-phase commit begins, during which the DM issues *prewrite* commands for X and Y. These commands instruct the DM to copy the values of X and Y from T's private workspace onto secure storage. If the DBMS fails during the first phase, no harm is done, since none of T's updates have yet been applied to the stored database. During the *second phase*, the TM issues *dm-write* commands for X and Y which instruct the DM to copy the values of X and Y into the stored database. If the DBMS fails during the second phase, the database may contain incorrect information, but since the values of X and Y are stored on secure storage, this inconsistency can be rectified when the system recovers: the recovery procedure reads the values of X and Y from secure storage and resumes the commitment activity.

We emphasize that this is a *mathematical model* of transaction processing, an approximation to the way DBMSs actually function. While the implementation details of atomic commitment are important in designing a DBMS, they are not central to an understanding of concurrency control. To explain concurrency control algorithms we need a model of transaction execution in which atomic commitment is visible, but not dominant.

1.3 Distributed Transaction-Processing Model

Our model of transaction processing in a distributed environment differs from that in a centralized one in two areas: handling private workspaces and implementing two-phase commit.

[1] The term "two-phase commit" is commonly used to denote the distributed version of this procedure. However, since the centralized and distributed versions are identical in structure, we use "two-phase commit" to describe both.

In a centralized DBMS we assumed that (1) private workspaces were part of the TM, and (2) data could freely move between a transaction and its workspace, and between a workspace and the DM. These assumptions are not appropriate in a DDBMS because TMs and DMs may run at different sites and the movement of data between a TM and a DM can be expensive. To reduce this cost, many DDBMSs employ *query optimization procedures* which regulate (and, it is hoped, reduce) the flow of data between sites. For example, in SDD-1 the private workspace for transaction T is distributed across all sites at which T accesses data [BERN81]. The details of how T reads and writes data in these workspaces is a query optimization problem and has no direct effect on concurrency control.

The problem of atomic commitment is aggravated in a DDBMS by the possibility of one site failing while the rest of the system continues to operate. Suppose T is updating x, y, z stored at DM_x, DM_y, DM_z, and suppose T's TM fails after issuing dm-write(x), but before issuing the dm-writes for y and z. At this point the database is incorrect. In a centralized DBMS this phenomenon is not harmful because no transaction can access the database until the TM recovers from the failure. However, in a DDBMS, other TMs remain operational and can access the incorrect database.

To avoid this problem, prewrite commands must be modified slightly. In addition to specifying data items to be copied onto secure storage, prewrites also specify which other DMs are involved in the commitment activity. Then if the TM fails during the second phase of two-phase commit, the DMs whose dm-writes were not issued can recognize the situation and consult the other DMs involved in the commitment. If any DM received a dm-write, the remaining ones act as if they had also received the command. The details of this procedure are complex and appear in HAMM80.

As in a centralized DBMS, a transaction T accesses the system by issuing BEGIN, READ, WRITE, and END operations. In a DDBMS these are processed as follows.

BEGIN: The TM creates a private workspace for T. We leave the location and organization of this workspace unspecified.

READ(X): The TM checks T's private workspace to see if a copy of X is present. If so, that copy's value is made available to T. Otherwise the TM selects some stored copy of X, say x_i, and issues dm-read(x_i) to the DM at which x_i is stored. The DM responds by retrieving the stored value of x_i from the database, placing it in the private workspace. The TM returns this value to T.

WRITE(X, new-value): The value of X in T's private workspace is updated to new-value, assuming the workspace contains a copy of X. Otherwise, a copy of X with the new value is created in the workspace.

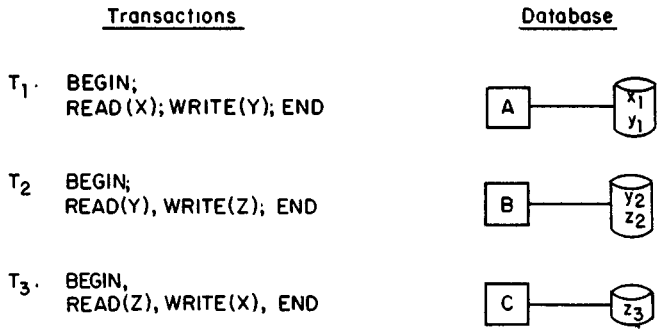
END: Two-phase commit begins. For each X updated by T, and for each stored copy x_i of X, the TM issues a prewrite (x_i) to the DM that stores x_i. The DM responds by copying the value of X from T's private workspace onto secure storage internal to the DM. After all prewrites are processed, the TM issues dm-writes for all copies of all logical data items updated by T. A DM responds to dm-write(x_i) by copying the value of x_i from secure storage into the stored database. After all dm-writes are installed, T's execution is finished.

2. DECOMPOSITION OF THE CONCURRENCY CONTROL PROBLEM

In this section we review concurrency control theory with two objectives: to define "correct executions" in precise terms, and to decompose the concurrency control problem into more tractable subproblems.

2.1 Serializability

Let E denote an execution of transactions $T_1, \ldots, T_n$. E is a *serial execution* if no transactions execute concurrently in E; that is, each transaction is executed to completion before the next one begins. Every serial execution is defined to be *correct*, because the properties of transactions (see Section 1.1) imply that a serial execution terminates properly and preserves database consistency. An execution is *serializable* if it is computationally equivalent to a serial execution, that is, if it produces the same output and has the same effect on the database as some serial execution. Since serial executions are correct and every serializable execution is equivalent to a serial one, every serializable execution is also correct. The



One possible execution of T_1, T_2, and T_3 is represented by the following logs. (Note. $r_i[x]$ denotes the operation dm-read(x) issued by T_i; $w_i[x]$ denotes a dm-write(x) issued by T_i.)

Log for DM A: $r_1[x_1]w_1[y_1]r_2[y_1]w_3[x_1]$
Log for DM B: $w_1[y_2]w_2[z_2]$
Log for DM C. $w_2[z_3]r_3[z_3]$

Figure 4. Modeling executions as logs.

- The execution modeled in Figure 4 is serial. Each log is itself serial; that is, there is no interleaving of operations from different transactions. At DM A, T_1 precedes T_2 precedes T_3; at DM B, T_1 precedes T_2; and at DM C, T_2 precedes T_3. Therefore, T_1, T_2, T_3 is a total order satisfying the definition of serial.
- The following execution is not serial. The logs themselves are not serial.

  DM A: $r_1[x_1]r_2[y_1]w_3[x_1]w_1[y_1]$
  DM B: $w_2[z_2]w_1[y_2]$
  DM C: $w_2[z_3]r_3[z_3]$

- The following execution is also not serial Although each log is serial, there is no total order consistent with all logs.

  DM A: $r_1[x_1]w_1[y_1]r_2[y_1]w_3[x_1]$
  DM B: $w_2[z_2]w_1[y_2]$
  DM C: $w_2[z_3]r_3[z_3]$

Figure 5. Serial and nonserial loops.

goal of database concurrency control is to ensure that all executions are serializable.

The only operations that access the stored database are dm-read and dm-write. Hence it is sufficient to model an execution of transactions by the execution of dm-reads and dm-writes at the various DMs of the DDBMS. In this spirit we formally model an execution of transactions by a set of *logs*, each of which indicates the order in which dm-reads and dm-writes are processed at one DM (see Figure 4). An execution is *serial* if there is a total order of transactions such that if T_i precedes T_j in the total order, then all of T_i's operations precede all of T_j's operations in every log where both appear (see Figure 5). Intuitively, this says that transactions execute serially and in the same order at all DMs.

Two operations *conflict* if they operate on the same data item and one of the operations is a dm-write. The order in which operations execute is computationally significant if and only if the operations conflict. To illustrate the notion of conflict, consider a data item x and transactions T_i and T_j. If T_i issues dm-read (x) and T_j issues dm-write(x), the value read by T_i will (in general) differ depending on whether the dm-read precedes or follows the dm-write. Similarly, if both transactions issue dm-write(x) operations, the final value of x depends on which dm-write happens last. Those conflict situations are called *read-write* (*rw*) *conflicts* and *write-write* (*ww*) *conflicts*, respectively.

The notion of conflict helps characterize the equivalence of executions. Two executions are *computationally equivalent* if (1) each dm-read operation reads data item values that were produced by the same dm-writes in both executions; and (2) the final dm-write on each data item is the same in both executions [PAPA77, PAPA79]. Condition (1) ensures that each transaction reads the same input in both executions (and therefore performs the same computation).

Combined with (2), it ensures that both executions leave the database in the same final state.

From this we can characterize serializable executions precisely.

Theorem 1 [PAPA77, PAPA79, STEA76]

Let $T = \{T_1, \ldots, T_m\}$ *be a set of transactions and let E be an execution of these transactions modeled by logs* $\{L_1, \ldots, L_m\}$. *E is serializable if there exists a total ordering of T such that for each pair of conflicting operations* O_i *and* O_j *from distinct transactions* T_i *and* T_j *(respectively),* O_i *precedes* O_j *in any log* $L_1, \ldots, L_m$ *if and only if* T_i *precedes* T_j *in the total ordering.*

The total order hypothesized in Theorem 1 is called a *serialization order*. If the transactions had executed serially in the serialization order, the computation performed by the transactions would have been identical to the computation represented by E.

To attain serializability, the DDBMS must guarantee that all executions satisfy the condition of Theorem 1, namely, that conflicting dm-reads and dm-writes be processed in certain relative orders. *Concurrency control* is the activity of controlling the relative order of conflicting operations; an algorithm to perform such control is called a *synchronization technique*. To be correct, a DDBMS must incorporate synchronization techniques that guarantee the conditions of Theorem 1.

2.2 A Paradigm for Concurrency Control

In Theorem 1, rw and ww conflicts are treated together under the general notion of conflict. However, we can decompose the concept of serializability by distinguishing these two types of conflict. Let E be an execution modeled by a set of logs. We define several binary relations on transactions in E, denoted by $\rightarrow$ with various subscripts. For each pair of transactions, T_i and T_j

(1) $T_i \rightarrow_{rw} T_j$ if in some log of E, T_i reads some data item into which T_j subsequently writes;
(2) $T_i \rightarrow_{wr} T_j$ if in some log of E, T_i writes into some data item that T_j subsequently reads;
(3) $T_i \rightarrow_{ww} T_j$ if in some log of E, T_i writes into some data item into which T_j subsequently writes;
(4) $T_i \rightarrow_{rwr} T_j$ if $T_i \rightarrow_{rw} T_j$ or $T_i \rightarrow_{wr} T_j$;
(5) $T_i \rightarrow T_j$ if $T_j \rightarrow_{rwr} T_j$ or $T_i \rightarrow_{ww} T_j$.

Intuitively, $\rightarrow$ (with any subscript) means "in any serialization must precede." For example, $T_i \rightarrow_{rw} T_j$ means "T_i in any serialization must precede T_j." This interpretation follows from Theorem 1: If T_i reads x before T_j writes into x, then the hypothetical serialization in Theorem 1 must have T_i preceding T_j.

Every conflict between operations in E is represented by an $\rightarrow$ relationship. Therefore, we can restate Theorem 1 in terms of $\rightarrow$. According to Theorem 1, E is serializable if there is a total order of transactions that is consistent with $\rightarrow$. This latter condition holds if and only if $\rightarrow$ is acyclic. (A relation, $\rightarrow$, is *acyclic* if there is no sequence $T_1 \rightarrow T_2, T_2 \rightarrow T_3, \ldots, T_n\text{-}1 \rightarrow T_n$ such that $T_1 = T_n$.) Let us decompose $\rightarrow$ into its components, $\rightarrow_{rwr}$ and $\rightarrow_{ww}$, and restate the theorem using them.

Theorem 2 [BERN80a]

Let $\rightarrow_{rwr}$ *and* $\rightarrow_{ww}$ *be associated with execution E. E is serializable if (a)* $\rightarrow_{rwr}$ *and* $\rightarrow_{ww}$ *are acyclic, and (b) there is a total ordering of the transactions consistent with all* $\rightarrow_{rwr}$ *and all* $\rightarrow_{ww}$ *relationships.*

Theorem 2 is an immediate consequence of Theorem 1. (Indeed, part (b) of Theorem 2 is essentially a restatement of the earlier theorem.) However, this way of characterizing serializability suggests a way of decomposing the problem into simpler parts. Theorem 2 implies that rw and ww conflicts can be synchronized independently except insofar as there must be a total ordering of the transactions consistent with both types of conflicts. This suggests that we can use one technique to guarantee an acyclic $\rightarrow_{rwr}$ relation (which amounts to *read-write synchronization*) and a different technique to guarantee an acyclic $\rightarrow_{ww}$ relation (*write-write synchronization*). However, in addition to both $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ being acyclic, there must also be *one* serial order

consistent with *all* $\rightarrow$ relations. This serial order is the cement that binds together the rw and ww synchronization techniques.

Decomposing serializability into rw and ww synchronization is the cornerstone of our paradigm for concurrency control. It will be important hereafter to distinguish algorithms that attain either rw or ww synchronization from algorithms that solve the entire distributed concurrency control problem. We use the term *synchronization technique* for the former type of algorithm, and *concurrency control method* for the latter.

3. SYNCHRONIZATION TECHNIQUES BASED ON TWO-PHASE LOCKING

Two-phase locking (2PL) synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations. Before reading data item x, a transaction must "own" a *readlock on* x. Before writing into x, it must "own" a *writelock on* x. The ownership of locks is governed by two rules: (1) different transactions cannot simultaneously own *conflicting locks*; and (2) once a transaction surrenders ownership of a lock, it may never obtain additional locks.

The definition of *conflicting lock* depends on the type of synchronization being performed: for rw synchronization two locks *conflict* if (a) both are locks on the same data item, and (b) one is a readlock and the other is a writelock; for ww synchronization two locks *conflict* if (a) both are locks on the same data item, and (b) both are writelocks.

The second lock ownership rule causes every transaction to obtain locks in a *two-phase* manner. During the *growing phase* the transaction obtains locks without releasing any locks. By releasing a lock the transaction enters the *shrinking phase*. During this phase the transaction releases locks, and, by rule 2, is prohibited from obtaining additional locks. When the transaction terminates (or aborts), all remaining locks are automatically released.

A common variation is to require that transactions obtain all locks before beginning their main execution. This variation is called *predeclaration*. Some systems also require that transactions hold all locks until termination

Two-phase locking is a correct synchronization technique, meaning that 2PL attains an acyclic $\rightarrow_{rwr}$ ($\rightarrow_{ww}$) relation when used for rw (ww) synchronization [BERN79b, ESWA76, PAPA79]. The serialization order attained by 2PL is determined by the order in which transactions obtain locks. The point at the end of the growing phase, when a transaction owns all the locks it ever will own, is called the *locked point* of the transaction [BERN79b]. Let E be an execution in which 2PL is used for rw (ww) synchronization. The $\rightarrow_{rwr}$ ($\rightarrow_{ww}$) relation induced by E is identical to the relation induced by a *serial* execution E′ in which every transaction executes at its locked point. Thus the locked points of E determine a serialization order for E.

3.1 Basic 2PL Implementation

An implementation of 2PL amounts to building a *2PL scheduler*, a software module that receives lock requests and lock releases and processes them according to the 2PL specification.

The basic way to implement 2PL in a distributed database is to distribute the schedulers along with the database, placing the scheduler for data item x at the DM were x is stored. In this implementation readlocks may be implicitly requested by dm-reads and writelocks may be implicitly requested by prewrites. If the requested lock cannot be granted, the operation is placed on a waiting queue for the desired data item. (This can produce a *deadlock*, as discussed in Section 3.5.) Writelocks are implicitly released by dm-writes. However, to release readlocks, special lock-release operations are required. These lock releases may be transmitted in parallel with the dm-writes, since the dm-writes signal the start of the shrinking phase. When a lock is released, the operations on the waiting queue of that data item are processed first-in/first-out (FIFO) order.

Notice that this implementation "automatically" handles redundant data correctly. Suppose logical data item X has copies $x_1, \ldots, x_m$. If basic 2PL is used for rw synchronization, a transaction may read any copy and need only obtain a readlock

on the copy of X it actually reads. However, if a transaction updates X, then it must update all copies of X, and so must obtain writelocks on all copies of X (whether basic 2PL is used for rw or ww synchronization).

3.2 Primary Copy 2PL

Primary copy 2PL is a 2PL technique that pays attention to data redundancy [STON79]. One copy of each logical data item is designated the *primary copy*; before accessing any copy of the logical data item, the appropriate lock must be obtained on the primary copy.

For readlocks this technique requires more communication than basic 2PL. Suppose x_1 is the primary copy of logical data item X, and suppose transaction T wishes to read some other copy, x_i, of X. To read x_i, T must communicate with two DMs, the DM where x_1 is stored (so T can lock x_1) and the DM where x_i is stored. By contrast, under basic 2PL, T would only communicate with x_i's DM. For writelocks, however, primary copy 2PL does not incur extra communication. Suppose T wishes to update X. Under basic 2PL, T would issue prewrites to all copies of X (thereby requesting writelocks on these data items) and then issue dm-writes to all copies. Under primary copy 2PL the same operations would be required, but only the prewrite (x_1) would request a writelock. That is, prewrites would be sent for $x_1, \ldots, x_m$, but the prewrites for $x_2, \ldots, x_m$ would not implicitly request writelocks.

3.3 Voting 2PL

Voting 2PL (or *majority consensus 2PL*) is another 2PL implementation that exploits data redundancy. Voting 2PL is derived from the majority consensus technique of Thomas [THOM79] and is only suitable for ww synchronization.

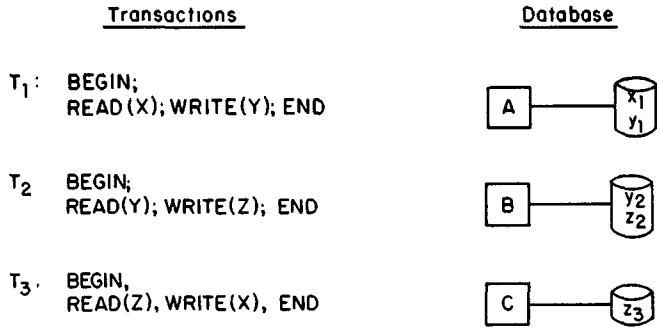
To understand voting, we must examine it in the context of two-phase commit. Suppose transaction T wants to write into X. Its TM sends prewrites to each DM holding a copy of X. For the voting protocol, the DM always responds immediately. It acknowledges receipt of the prewrite and says "lock set" or "lock blocked." (In the basic implementation it would not acknowledge at all until the lock is set.) After the TM receives acknowledgments from the DMs, it counts the number of "lockset" responses: if the number constitutes a majority, then the TM behaves as if all locks were set. Otherwise, it waits for "lockset" operations from DMs that originally said "lock blocked." Deadlocks aside (see Section 3.5), it will eventually receive enough "lockset" operations to proceed.

Since only one transaction can hold a majority of locks on X at a time, only one transaction writing into X can be in its second commit phase at any time. All copies of X thereby have the same sequence of writes applied to them. A transaction's locked point occurs when it has obtained a majority of its writelocks on each data item in its writeset. When updating many data items, a transaction must obtain a majority of locks on every data item before it issues any dm-writes.

In principle, voting 2PL could be adapted for rw synchronization. Before reading any copy of X a transaction requests readlocks on all copies of X; when a majority of locks are set, the transaction may read any copy. This technique works but is overly strong: Correctness only requires that a single copy of X be locked—namely, the copy that is read—yet this technique requests locks on all copies. For this reason we deem voting 2PL to be inappropriate for rw synchronization.

3.4 Centralized 2PL

Instead of distributing the 2PL schedulers, one can centralize the scheduler at a single site [ALSB76a, GARC79a]. Before accessing data at any site, appropriate locks must be obtained from the central 2PL scheduler. So, for example, to perform dm-read(x) where x is not stored at the central site, the TM must first request a readlock on x from the central site, wait for the central site to acknowledge that the lock has been set, then send dm-read(x) to the DM that holds x. (To save some communication, one can have the TM send both the lock request and dm-read (x) to the central site and let the central site directly forward dm-read(x) to x's DM; the DM then responds to the TM when dm-read (x) has been processed.) Like primary copy 2PL, this approach tends to require more communication than basic



- Suppose transactions execute concurrently, with each transaction issuing its READ before any transaction issues its END.
- This partial execution could be represented by the following logs

  DM A: $r_1[x_1]$
  DM B: $r_2[y_2]$
  DM C: $r_3[z_3]$

- At this point, T_1 has readlock on x_1
  T_2 has readlock on y_2
  T_3 has readlock on z_3
- Before proceeding, all transactions must obtain writelocks.
  T_1 requires writelocks on y_1 and y_2
  T_2 requires writelocks on z_2 and z_3
  T_3 requires writelock on x_1
- But

  T_1 cannot get writelock on y_2, until T_2 releases readlock
  T_2 cannot get writelock on z_3, until T_3 releases readlock
  T_3 cannot get writelock on x_1, until T_1 releases readlock

  This is a deadlock

Figure 6. Deadlock.

2PL, since dm-reads and prewrites usually cannot implicitly request locks.

3.5 Deadlock Detection and Prevention

The preceding implementations of 2PL force transactions to wait for unavailable locks. If this waiting is uncontrolled, deadlocks can arise (see Figure 6).

Deadlock situations can be characterized by *waits-for graphs* [HOLT72, KING74], directed graphs that indicate which transactions are waiting for which other transactions. Nodes of the graph represent transactions, and edges represent the "waiting-for" relationship: an edge is drawn from transaction T_i to transaction T_j if T_i is waiting for a lock currently owned by T_j. There is a deadlock in the system if and only if the waits-for graph contains a *cycle* (see Figure 7).

Two general techniques are available for deadlock resolution: *deadlock prevention* and *deadlock detection.*

3.5.1 Deadlock Prevention

Deadlock prevention is a "cautious" scheme in which a transaction is restarted when the system is "afraid" that deadlock might occur. To implement deadlock prevention, 2PL schedulers are modified as follows. When a lock request is denied, the scheduler tests the requesting transaction (say T_i) and the transaction that currently owns the lock (say T_j). If T_i and T_j pass the test, T_i is permitted to wait for T_j as usual. Otherwise, one of the two is aborted. If T_i is restarted, the deadlock prevention algorithm is called *nonpreemptive*; if T_j is restarted, the algorithm is called *preemptive.*

The test applied by the scheduler must

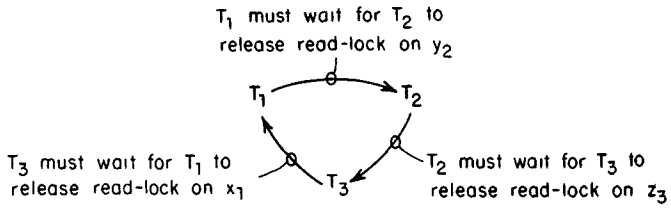


Figure 7. Waits-for graph for Figure 6.

guarantee that if T_i waits for T_j, then deadlock cannot result. One simple approach is never to let T_i wait for T_j. This trivially prevents deadlock but forces many restarts.

A better approach is to assign *priorities* to transactions and to test priorities to decide whether T_i can wait for T_j. For example, we could let T_i wait for T_j if T_i has lower priority than T_j (if T_i and T_j have *equal* priorities, T_i cannot wait for T_j, or vice versa). This test prevents deadlock because, for every edge $\langle T_i, T_j \rangle$ in the waits-for graph, T_i has lower priority than T_j. Since a cycle is a path from a node to itself and since T_i cannot have lower priority than itself, no cycle can exist.

One problem with the preceding approach is that *cyclic restart* is possible—some unfortunate transaction could be continually restarted without ever finishing. To avoid this problem, Rosenkrantz et al. [ROSE78] propose using "timestamps" as priorities. Intuitively, a transaction's timestamp is the time at which it begins executing, so old transactions have higher priority than young ones.

The technique of ROSE78 requires that each transaction be assigned a *unique* timestamp by its TM. When a transaction begins, the TM reads the local clock time and appends a unique TM identifier to the low-order bits [THOM79]. The resulting number is the desired timestamp. The TM also agrees not to assign another timestamp until the next clock tick. Thus timestamps assigned by different TMs differ in their low-order bits (since different TMs have different identifiers), while timestamps assigned by the same TM differ in their high-order bits (since the TM does not use the same clock time twice). Hence timestamps are unique throughout the system. Note that this algorithm does not require clocks at different sites to be precisely synchronized.

Two timestamp-based deadlock prevention schemes are proposed in ROSE78. *Wait-Die* is the nonpreemptive technique. Suppose transaction T_i tries to wait for T_j. If T_i has lower priority than T_j (i.e., T_i is younger than T_j), then T_i is permitted to wait. Otherwise, it is aborted ("dies") and forced to restart. It is important that T_i not be assigned a new timestamp when it restarts. *Wound-Wait* is the preemptive counterpart to *Wait-Die*. If T_i has higher priority than T_j, then T_i waits; otherwise T_j is aborted.

Both Wait-Die and Wound-Wait avoid cyclic restart. However, in Wound-Wait an old transaction may be restarted many times, while in Wait-Die old transactions never restart. It is suggested in ROSE78 that Wound-Wait induces fewer restarts in total.

Care must be exercised in using preemptive deadlock prevention with two-phase commit: a transaction must not be aborted once the second phase of two-phase commit has begun. If a preemptive technique wishes to abort T_j, it checks with T_j's TM and cancels the abort if T_j has entered the second phase. No deadlock can result because if T_j is in the second phase, it cannot be waiting for any transactions.

Preordering of resources is a deadlock avoidance technique that avoids restarts altogether. This technique requires predeclaration of locks (each transaction obtains all its locks before execution). Data items are numbered and each transaction requests locks one at a time in numeric order. The priority of a transaction is the number of the highest numbered lock it owns. Since a transaction can only wait for transactions with higher priority, no deadlocks can occur. In addition to requiring predeclaration, a principal disadvantage of this technique is that it forces locks to be obtained sequentially, which tends to increase response time.

- Consider the execution illustrated in Figures 6 and 7.
- Locks are requested at DMs in the following order:

DM A	DM B	DM C
readlock x_1 for T_1	readlock y_2 for T_2	readlock z_3 for T_3
writelock y_1 for T_1	writelock z_2 for T_2	
*writelock x_1 for T_3	*writelock y_2 for T_1	*writelock z_3 for T_2

- None of the "starred" locks can be granted and the system is in deadlock. However, the waits-for graphs at each DM are acyclic.

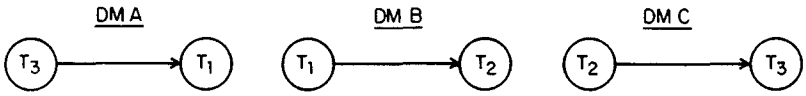


Figure 8. Multisite deadlock.

3.5.2 Deadlock Detection

In *deadlock detection*, transactions wait for each other in an uncontrolled manner and are only aborted if a deadlock actually occurs. Deadlocks are detected by explicitly constructing the waits-for graph and searching it for cycles. (Cycles in a graph can be found efficiently using, for example, Algorithm 5.2 in AHO75.) If a cycle is found, one transaction on the cycle, called the *victim*, is aborted, thereby breaking the deadlock. To minimize the cost of restarting the victim, victim selection is usually based on the amount of resources used by each transaction on the cycle.

The principal difficulty in implementing deadlock detection in a distributed database is constructing the waits-for graph efficiently. Each 2PL scheduler can easily construct the waits-for graph based on the waits-for relationships local to that scheduler. However, these local waits-for graphs are not sufficient to characterize all deadlocks in the distributed system (see Figure 8). Instead, local waits-for graphs must be combined into a more "global" waits-for graph. (Centralized 2PL does not have this problem, since there is only one scheduler.) We describe two techniques for constructing global waits-for graphs: centralized and hierarchical deadlock detection.

In the *centralized* approach, one site is designated the deadlock detector for the distributed system [GRAY78, STON79]. Periodically (e.g., every few minutes) each scheduler sends its local waits-for graph to the deadlock detector. The deadlock detector combines the local graphs into a system-wide waits-for graph by constructing the union of the local graphs.

In the *hierarchical* approach, the database sites are organized into a hierarchy (or tree), with a deadlock detector at each node of the hierarchy [MENA79]. For example, one might group sites by *region*, then by *country*, then by *continent*. Deadlocks that are local to a single site are detected at that site; deadlocks involving two or more sites of the same region are detected by the regional deadlock detector; and so on.

Although centralized and hierarchical deadlock detection differ in detail, both involve periodic transmission of local waits-for information to one or more deadlock detector sites. The periodic nature of the process introduces two problems. First, a deadlock may exist for several minutes without being detected, causing response-time degradation. The solution, executing the deadlock detector more frequently, increases the cost of deadlock detection. Second, a transaction T may be restarted for reasons other than concurrency control (e.g., its site crashed). Until T's restart propagates to the deadlock detector, the deadlock detector can find a cycle in the waits-for graph that includes T. Such a cycle is called a *phantom deadlock*. When the deadlock detector discovers a phantom deadlock, it may unnecessarily restart a transaction other than T. Special precautions are also needed to avoid unnecessary restarts for deadlocks in voting 2PL.[2]

[2] Suppose logical data item X has copies x_1, x_2, and x_3, and suppose using voting 2PL T_i owns write-locks on x_1 and x_2 but T_i's lock request for x_3 is blocked by T_j.

A major cost of deadlock detection is the restarting of partially executed transactions. Predeclaration can be used to reduce this cost. By obtaining a transaction's locks before it executes, the system will only restart transactions that have not yet executed. Thus little work is wasted by the restart.

4. SYNCHRONIZATION TECHNIQUES BASED ON TIMESTAMP ORDERING

Timestamp ordering (T/O) is a technique whereby a serialization order is selected a priori and transaction execution is forced to obey this order. Each transaction is assigned a unique timestamp by its TM. The TM attaches the timestamp to all dm-reads and dm-writes issued on behalf of the transaction, and DMs are required to process *conflicting operations* in timestamp order. The timestamp of operation O is denoted ts(O).

The definition of conflicting operations depends on the type of synchronization being performed and is analogous to conflicting locks. For rw synchronization, two operations *conflict* if (a) both operate on the same data item, and (b) one is a dm-read and the other is a dm-write. For ww synchronization, two operations *conflict* if (a) both operate on the same data item, and (b) both are dm-writes.

It is easy to prove that T/O attains an acyclic $\rightarrow_{rwr}$ ($\rightarrow_{ww}$) relation when used for rw (ww) synchronization. Since each DM processes conflicting operations in timestamp order, each edge of the $\rightarrow_{rwr}$ ($\rightarrow_{ww}$) relation is in timestamp order. Consequently, all paths in the relation are in timestamp order and, since all transactions have unique timestamps, no cycles are possible. In addition, the timestamp order is a valid serialization order.

4.1 Basic T/O Implementation

An implementation of T/O amounts to building a *T/O scheduler*, a software module that receives dm-reads and dm-writes and outputs these operations according to the T/O specification [SHAP77a, SHAP77b]. In practice, prewrites must also be processed through the T/O scheduler for two-phase commit to operate properly. As was the case with 2PL, the basic T/O implementation distributes the schedulers along with the database [BERN80a].

If we ignore two-phase commit, the basic T/O scheduler is quite simple. At each DM, and for each data item x stored at the DM, the scheduler records the largest timestamp of any dm-read(x) or dm-write(x) that has been processed. These are denoted R-ts(x) and W-ts(x), respectively. For rw synchronization, scheduler S operates as follows. Consider a dm-read(x) with timestamp TS. If TS < W-ts(x), S *rejects* the dm-read and aborts the issuing transaction. Otherwise S outputs the dm-read and sets R-ts(x) to max(R-ts(x),TS). For a dm-write(x) with timestamp TS, S rejects the dm-write if TS < R-ts(x); otherwise it outputs the dm-write and sets W-ts(x) to max(W-ts(x),TS). For ww synchronization, S rejects a dm-write(x) with timestamp TS if TS < W-ts(x); otherwise it outputs the dm-write and sets W-ts(x) to TS.

When a transaction is aborted, it is assigned a new and larger timestamp by its TM and is restarted. Restart issues are discussed further below.

Two-phase commit is incorporated by timestamping prewrites and accepting or rejecting prewrites instead of dm-writes. Once a scheduler accepts a prewrite, it must guarantee to accept the corresponding dm-write no matter when the dm-write arrives. For rw (or ww) synchronization, once S accepts a prewrite(x) with timestamp TS it must not output any dm-read(x) (or dm-write(x)) with timestamp greater than TS until the dm-write(x) is output. The effect is similar to setting a writelock on x for the duration of two-phase commit..

To implement the above rules, S *buffers* dm-reads, dm-writes, and prewrites. Let min-R-ts(x) be the minimum timestamp of any buffered dm-read(x), and define min-W-ts(x) and min-P-ts(x) analogously. Rw synchronization is accomplished as follows:

1. Let R be a dm-read(x). If ts(R) < W-ts(x), R is rejected. Else if ts(R) > min-P-ts(x), R is buffered. Else R is output.

Insofar as x_3's scheduler is concerned, T_i is waiting for T_j. However, since T_i has a *majority* of the copies locked, T_i can proceed without waiting for T_j. This fact should be incorporated into the deadlock resolution scheme to avoid unnecessary restarts.

Let R = dm-read (x).
Let W = dm-write (x).
R is *ready* if it precedes the earliest prewrite request:
if ts(R) < min-P-ts(x).
W is *ready* if it precedes the earliest dm-read request:
if ts (W) < min-R-ts(x).
When a dm-write(x) arrives, do the following:

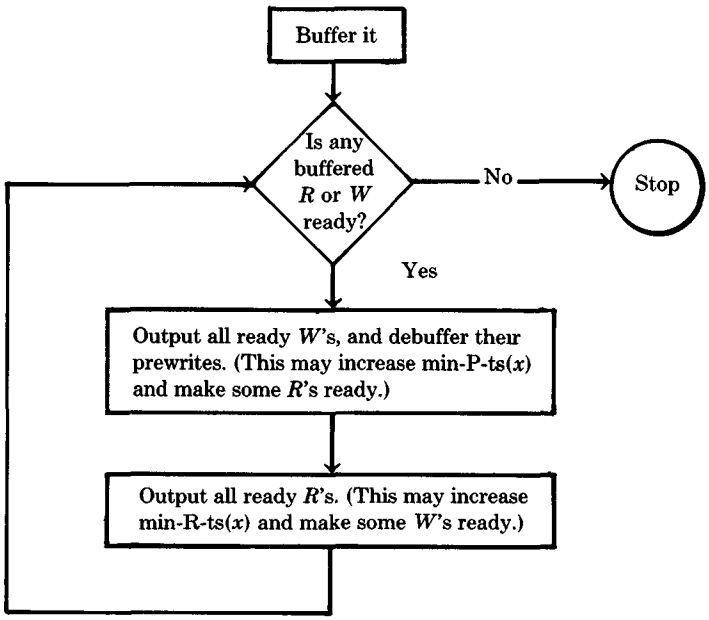


Figure 9. Buffer emptying for basic T/O rw synchronization.

2. Let P be a prewrite(x). If ts(P) < R-ts(x), P is rejected. Else P is buffered.
3. Let W be a dm-write(x). W is never rejected. If ts(W) > min-R-ts(x), W is buffered. (If W were output it would cause a buffered dm-read(x) to be rejected.) Else W is output.
4. When W is output, the corresponding prewrite is debuffered. If this causes min-P-ts(x) to increase, the buffered dm-reads are retested to see if any of them can be output. If this causes min-R-ts(x) to be increased, the buffered dm-writes are also retested, and so forth. This process is diagramed in Figure 9.

Ww synchronization is accomplished as follows:

1. Let P be a prewrite(x). If ts(P) < W-ts(x), P is rejected; else P is buffered.
2. Let W be a dm-write(x). W is never rejected. If ts(W) > min-P-ts(x), W is buffered; else W is output.
3. When W is output, the corresponding prewrite is debuffered. If this causes min-P-ts(x) to be increased, the buffered dm-writes are retested to see if any can now be output. See Figure 10.

As with 2PL, a common variation is to require that transactions *predeclare* their readsets and writesets, issuing all dm-reads and prewrites before beginning their main execution.[3] If all operations are accepted,

[3] These prewrites are nonstandard relative to the definition in Section 1.4. Since new values for the data items in the writeset are not yet known, these prewrites do *not* instruct DMs to store values on secure storage; instead, prewrite (x) merely "warns" the DM to expect a dm-write (x) in the near future. However, these prewrites are processed by synchronization algorithms exactly as "standard" ones are.

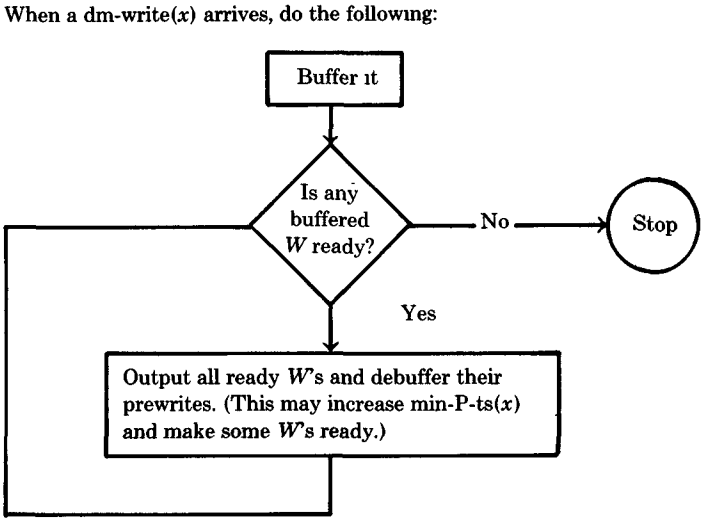


Figure 10. Buffer emptying for basic T/O ww synchronization.

the transaction is guaranteed to execute without danger of restart. Another variation is to *delay* the processing of operations to wait for operations with smaller timestamps. The extreme version of this heuristic is *conservative T/O*, described in Section 4.4.

4.2 The Thomas Write Rule

For ww synchronization the basic T/O scheduler can be optimized using an observation of THOM79. Let W be a dm-write(x), and suppose ts(W) < W-ts(x). Instead of rejecting W *we can simply ignore it.* We call this the *Thomas Write Rule* (*TWR*). Intuitively, TWR applies to a dm-write that tries to place obsolete information into the database. The rule guarantees that the effect of applying a set of dm-writes to x is identical to what would have happened had the dm-writes been applied in timestamp order.

If TWR is used, there is no need to incorporate two-phase commit into the ww synchronization algorithm; the ww scheduler always accepts prewrites and never buffers dm-writes.

4.3 Multiversion T/O

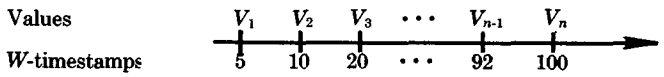
For rw synchronization the basic T/O scheduler can be improved using *multiversion data items* [REED78]. For each data item x there is a *set* of R-ts's and a set of ⟨W-ts, value⟩ pairs, called *versions*. The R-ts's of x record the timestamps of all executed dm-read(x) operations, and the versions record the timestamps and values of all executed dm-write(x) operations. (In practice one cannot store R-ts's and versions forever; techniques for deleting old versions and timestamps are described in Sections 4.5 and 5.2.2.)

Multiversion T/O accomplishes rw synchronization as follows (ignoring two-phase commit). Let R be a dm-read(x). R is processed by reading the version of x with largest timestamp less than ts(R) and adding ts(R) to x's set of R-ts's; see Figure 11a. R is never rejected. Let W be a dm-write(x), and let *interval*(W) be the interval from ts(W) to the smallest W-ts(x) > ts(W);[4] see Figure 11b. If any R-ts(x) lies in interval(W), W is rejected; otherwise W is output and creates a new version of x with timestamp ts(W).

To prove the correctness of multiversion T/O, we must show that every execution is equivalent to a serial execution in timestamp order [BERN80b]. Let R be a dm-read(x) that is processed "out of order"; that is, suppose R is executed after a dm-write(x) whose timestamp exceeds ts(R). Since R ignores all versions with time-

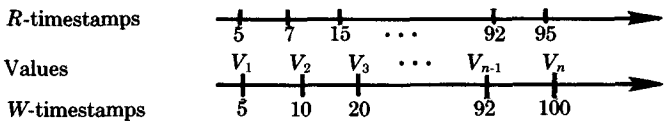
[4] Interval(W) = (ts(W),∞) if no W-ts(x) > ts (W) exists.

(a) Let us represent the versions of a data item x on a "time line":



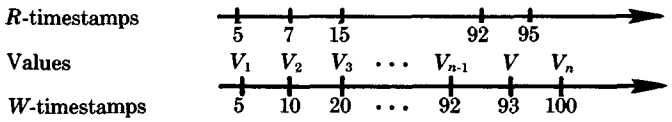
To process a dm-read(x) with timestamp 95, find the biggest W-timestamp less than 95; in this case 92. That is the version you read. So in this case, the value read by the dm-read is V_{n-1}.

(b) Let us represent the R-timestamps of x similarly:



Let W be dm-write(x) with timestamp 93. Interval(W) = (93,100).

To process W we create a new version of x with that timestamp.



However, this new version "invalidates" the dm-read of part (a), because if the dm-read had arrived after the dm-write, it would have read value V instead of V_{n-1}. Therefore, we must reject the dm-write.

Figure 11. Multiversion reading and writing.

stamps greater than ts(R), the value read by R is identical to the value it would have read had it been processed in timestamp order. Now let W be a dm-write(x) that is processed "out of order"; that is, suppose it is executed after a dm-read(x) whose timestamp exceeds ts(W). Since W was not rejected, there exists a version of x with timestamp TS such that ts(W) < TS < ts(dm-read). Again the effect is identical to a timestamp-ordered execution.

For ww synchronization, multiversion T/O is essentially an embellished version of TWR. A dm-write(x) always creates a new version of x with timestamp ts(dm-write) and is never rejected.

Integrating two-phase commit requires that dm-reads and prewrites (but not dm-writes) be buffered as in basic T/O. Let P be a buffered prewrite(x): *interval*(P) is the interval from ts(P) to the smallest W-ts(x) > ts(P). Rw synchronization is performed as follows:

1. Let R be a dm-read(x). R is never rejected. If ts(R) lies in interval(prewrite(x)) for some buffered prewrite(x), then R is buffered. Else R is output.
2. Let P be a prewrite(x). If some R-ts(x) lies in interval(P), P is rejected. Else P is buffered.
3. Let W be a dm-write(x). W is always output immediately.
4. When W is output, its prewrite is debuffered, and the buffered dm-reads are retested to see if they can now be output. See Figure 12.

Two-phase commit is not an issue for ww synchronization, since dm-writes are never rejected for ww synchronization.

Let R = dm-read(x). R is *ready* if ts(R) $\notin$ interval (P), where P is any buffered prewrite(x).

When a dm-write arrives do the following:

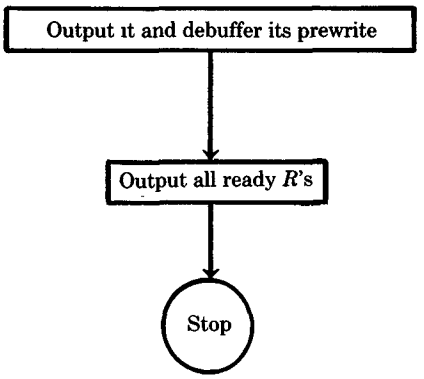


Figure 12. Buffer emptying for multiversion T/O.

4.4 Conservative T/O

Conservative timestamp ordering is a technique for eliminating restarts during T/O scheduling [BERN80a]. When a scheduler receives an operation O that might cause a future restart, the scheduler *delays* O until it is sure that no future restarts are possible.

Conservative T/O requires that each scheduler receive dm-reads (or dm-writes) from each TM in timestamp order. For example, if scheduler S_j receives dm-read(x) followed by dm-read(y) from TM_i, then ts(dm-read(x)) $\leq$ ts(dm-read(y)). Since the network is assumed to be a FIFO channel, this timestamp ordering is accomplished by requiring that TM_i *send* dm-reads (or dm-writes) to S_j in timestamp order.[5]

Conservative T/O buffers dm-reads and dm-writes as part of its normal operation. When a scheduler buffers an operation, it remembers the TM that sent it. Let min-R-ts(TM_i) be the minimum timestamp of any buffered dm-read from TM_i, with min-R-ts(TM_i) = $-\infty$ if no such dm-read is buffered. Define min-W-ts(TM_i) analogously.

Conservative T/O performs rw synchronization as follows:

1. Let R be a dm-read(x). If ts(R) > min-W-ts(TM) for any TM in the system, R is buffered. Else R is output.
2. Let W be a dm-write(x). If ts(W) > min-R-ts(TM) for any TM, W is buffered. Else W is output.
3. When R or W is output or buffered, this may increase min-R-ts(TM) or min-W-ts(TM); buffered operations are retested to see if they can now be output.

The effect is that R is output if and only if (a) the scheduler has a buffered dm-write from every TM, and (b) ts(R) < minimum timestamp of any buffered dm-write. Similarly, W is output if and only if (a) there is a buffered dm-read from every TM, and (b) ts(W) < minimum timestamp of any buffered dm-read. Thus R (or W) is output if and only if the scheduler has received every dm-write (or dm-read) with smaller timestamp that it will ever receive.

Ww synchronization is accomplished as follows:

1. Let W be a dm-write(x). If ts(W) > min-W-ts(TM) for any TM in the system, W is buffered; else it is output.
2. When W is buffered or output, this may increase min-W-ts(TM); buffered dm-writes are retested accordingly.

The effect is that the scheduler waits until it has a buffered dm-write from every TM and then outputs the dm-write with smallest timestamp.

Two-phase commit need not be tightly integrated into conservative T/O because dm-writes are never rejected. Although prewrites must be issued for all data items updated, these operations are not processed by the conservative T/O schedulers.

The above implementation of conservative T/O suffers three major problems: (1) If some TM never sends an operation to some scheduler, the scheduler will "get stuck" and stop outputting. (2) To avoid the first problem, every TM must communicate regularly with every scheduler; this is infeasible in large networks. (3) The im-

[5] This can be implemented by requiring that TMs process transactions serially Alternatively, we can require that transactions issue all dm-reads before beginning their main execution, and all dm-writes after terminating their main execution. Then transactions can execute concurrently, although they must terminate in timestamp order.

- A *class* is defined by a readset and a writeset. For example,

  C_1: readset = $\{x_1\}$, writeset = $\{y_1, y_2\}$
  C_2: readset = $\{x_1, y_2\}$, writeset = $\{y_1, y_2, z_2, z_3\}$
  C_3: readset = $\{y_2, z_3\}$, writeset = $\{x_1, z_2, z_3\}$

- A transaction is a member of a class if its readset is a subset of the class readset and its writeset is a subset of the class writeset. For example,

  T_1: readset = $\{x_1\}$, writeset = $\{y_1, y_2\}$
  T_2: readset = $\{y_2\}$, writeset = $\{z_2, z_3\}$
  T_3: readset = $\{z_3\}$, writeset = $\{x_1\}$

- T_1 is a member of C_1 and C_2
- T_2 is a member of C_2 and C_3
- T_3 is a member of C_3

Figure 13. Transaction classes.

plementation is overly conservative; the ww algorithm, for instance, processes *all* dm-writes in timestamp order, not merely conflicting ones. These problems are addressed below.

Null Operations. To solve the first problem, TMs are required to periodically send timestamped *null operations* to each scheduler in the absence of "real" traffic. A null operation is a dm-read or dm-write whose sole purpose is to convey timestamp information and thereby unblock "real" dm-reads and prewrites. An impatient scheduler can prompt a TM for a null operation by sending a "request message." For example, for rw synchronization suppose scheduler S wants to process a dm-read with timestamp TS, but does not have a buffered dm-write from TM_i. S can send a message to TM_i requesting a null-dm-write with timestamp greater than TS.

A variation is to use null operations with *very large* (perhaps infinite) timestamps. For example, if TM_i rarely needs to issue dm-reads to S, TM_i can send S a null-dm-read with infinite timestamp signifying that TM_i does not intend to communicate with S until further notice.

Transaction Classes. Transaction classes [BERN78a, BERN80d] is a technique for reducing communication in conservative T/O and for supporting a less conservative scheduling policy. As in predeclaration, assume that every transaction's readset and writeset are known in advance. A *class* is defined by a readset and a writeset (see Figure 13). Transaction T is a *member* of class C if readset(T) is a subset of readset(C) and writeset(T) is a subset of writeset(C). (Classes need not be disjoint.)

Class definitions are not expected to change frequently during normal operation of the system. Changing a class definition is akin to changing the database schema and requires mechanisms beyond the scope of this paper. We assume that class definitions are stored in static tables that are available to any site requiring them.

Classes are associated with TMs. Every transaction that executes at a TM must be a member of a class associated with the TM. If a transaction is submitted to a TM that has no class containing it, the transaction is forwarded to another TM that does. We assume that every class is associated with exactly one TM, and vice versa. The class associated with TM_i is denoted C_i. To execute transactions that are members of class C at two TMs, we define another class C′ with the same definition as C and associate C with one TM and C′ with the other. To execute transactions that are members of two classes at one site, we multiprogram two TMs at that site.

Classes are exploited by conservative T/O schedulers as follows. Consider rw synchronization and suppose scheduler S wants to output a dm-read(x). Instead of waiting for dm-writes with smaller timestamps from all TMs, S need only wait for dm-writes *from those TMs whose class writeset contains x*. Similarly, to process a dm-write (x), S need only wait for dm-reads with smaller timestamp from those TMs whose class readset contains x. Thus communication requirements are decreased, and the level of concurrency in the system is increased. Ww synchronization proceeds similarly.

Conflict Graph Analysis. Conflict graph analysis is a technique for further improving the performance of conservative T/O with classes. A *conflict graph* is an undirected graph that summarizes potential conflicts between transactions in different classes. For each class C_i the graph contains two nodes, denoted r_i and w_i, which represent the readset and writeset of C_i. The edges of the graph are defined as follows (see Figure 14): (1) For each class

Define C_1, C_2, C_3 as in Figure 13.

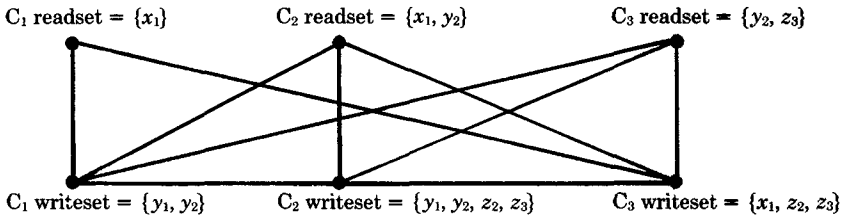


Figure 14. Conflict graph.

C_i there is a *vertical edge* between r_i and w_i; (2) for each pair of classes C_i and C_j (with $i \neq j$) there is a *horizontal edge* between w_i and w_j if and only if writeset(C_i) intersects writeset(C_j); (3) for each pair of classes C_i and C_j (with $i \neq j$) there is a *diagonal edge* between r_i and w_j if and only if readset(C_i) intersects writeset(C_j).

Intuitively, a horizontal edge indicates that a scheduler S may be forced to delay dm-writes for purposes of ww synchronization. Suppose classes C_i and C_j are connected by a horizontal edge (w_i, w_j), indicating that their class writesets intersect. If S receives a dm-write from C_i, it must delay the dm-write until it receives all dm-writes with smaller timestamps from C_j. Similarly, a diagonal edge indicates that S may need to delay operations for rw synchronization.

Conflict graph analysis improves the situation by identifying interclass conflicts that cannot cause nonserializable behavior. This corresponds to identifying horizontal and diagonal edges that do not require synchronization. In particular, schedulers need only synchronize dm-writes from C_i and C_j if either (1) the horizontal edge between w_i and w_j is embedded in a *cycle* of the conflict graph; or (2) portions of the intersection of C_i's writeset and C_j's writeset are stored at two or more DMs [BERN80c]. That is, if conditions (1) and (2) do not hold, scheduler S need not process dm-writes from C_i and C_j in timestamp order. Similarly, dm-reads from C_i and dm-writes from C_j need only be processed in timestamp order if either (1) the diagonal edge between r_i and w_j is embedded in a cycle of the conflict graph; or (2) portions of the intersection of C_i's readset and C_j's writeset are stored at two or more DMs.

Since classes are defined statically, conflict graph analysis is also performed statically. The analysis produces a table indicating which horizontal and vertical edges require synchronization and which do not. This table, like class definitions, is distributed in advance to all schedulers that need it.

4.5 Timestamp Management

A common criticism of T/O schedulers is that too much memory is needed to store timestamps. This problem can be overcome by "forgetting" old timestamps.

Timestamps are used in basic T/O to reject operations that "arrive late," for example, to reject a dm-read(x) with timestamp TS_1 that arrives after a dm-write(x) with timestamp TS_2, where $TS_1 < TS_2$. In principle, TS_1 and TS_2 can differ by an arbitrary amount. However, in practice it is unlikely that these timestamps will differ by more than a few minutes. Consequently, timestamps can be stored in small tables which are periodically purged.

R-ts's are stored in the *R-table* with entries of the form ⟨x, R-ts⟩; for any data item x, there is at most one entry. In addition, a variable, *R-min*, tells the maximum value of any timestamp that has been purged from the table. To find R-ts(x), a scheduler searches the R-table for an ⟨x, TS⟩ entry. If such an entry is found, R-ts(x) = TS; otherwise, R-ts(x) ≤ R-min. To err on the side of safety, the scheduler assumes R-ts(x) = R-min. To update R-ts(x), the scheduler modifies the ⟨x, TS⟩ entry, if one exists. Otherwise, a new entry is created and added to the table. When the R-table is full, the scheduler selects an appropriate value for R-min and deletes all entries from the table with smaller timestamp. W-ts's are managed similarly, and analogous techniques can be devised for multiversion databases.

Maintaining timestamps for conservative T/O is even cheaper, since conservative T/O requires only timestamped operations, not timestamped data. If conservative T/O is used for rw synchronization, the R-ts's of data items may be discarded. If conservative T/O is used for both rw and ww synchronization, W-ts's can be eliminated also.

5. INTEGRATED CONCURRENCY CONTROL METHODS

An integrated concurrency control method consists of two components—an rw and a ww synchronization technique—plus an interface between the components that attains condition (b) of Theorem 2: a total ordering of the transactions consistent with all $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ relationships. In this section we list 48 concurrency control methods that can be constructed using the techniques of Sections 3 and 4.

Approximately 20 concurrency control methods have been described in the literature. Virtually all of them use a *single* synchronization technique (either 2PL or T/O) for both rw and ww synchronization. Indeed, most methods use the same *variation* of a single technique for both kinds of synchronization. However, such homogeneity is neither necessary nor especially desirable.

For example, the analysis of Section 3.2 suggests that using basic 2PL for rw synchronization and primary copy 2PL for ww synchronization might be superior to using basic 2PL (or primary copy 2PL) for both. More outlandish combinations may be even better. For example, one can combine basic 2PL with TWR. In this method ww conflicts never cause transactions to be delayed or restarted; multiple transactions can write into the same data items concurrently (see Section 5.3).

In Sections 5.1 and 5.2 we describe methods that use 2PL and T/O techniques for both rw and ww synchronization. The concurrency control methods in these sections are easy to describe given the material of Sections 3 and 4; the description of each method is little more than a description of each component technique. In Section 5.3 we list 24 concurrency control methods that combine 2PL and T/O techniques. As we show in Section 5.3, methods of this type have useful properties that cannot be attained by pure 2PL or T/O methods.

5.1 Pure 2PL Methods

The 2PL synchronization techniques of Section 3 can be integrated to form 12 *principal 2PL methods*:

Method	rw technique	ww technique
1	Basic 2PL	Basic 2PL
2	Basic 2PL	Primary copy 2PL
3	Basic 2PL	Voting 2PL
4	Basic 2PL	Centralized 2PL
5	Primary copy 2PL	Basic 2PL
6	Primary copy 2PL	Primary copy 2PL
7	Primary copy 2PL	Voting 2PL
8	Primary copy 2PL	Centralized 2PL
9	Centralized 2PL	Basic 2PL
10	Centralized 2PL	Primary copy 2PL
11	Centralized 2PL	Voting 2PL
12	Centralized 2PL	Centralized 2PL

Each method can be further refined by the choice of deadlock resolution technique (see Section 3.5).

The interface between each 2PL rw technique and each 2PL ww technique is straightforward. It need only guarantee that "two-phasedness" is preserved, meaning that *all* locks needed for both the rw and ww technique must be obtained before any lock is released by *either* technique.

5.1.1 Methods Using Basic 2PL for rw Synchronization

Methods 1–4 use basic 2PL for rw synchronization. Consider a logical data item X with copies $x_1, \ldots, x_m$. To read X, a transaction sends a dm-read to *any* DM that stores a copy of X. This dm-read implicitly requests a readlock on the copy of X at that DM. To write X, a transaction sends prewrites to *every* DM that stores a copy of X. These prewrites implicitly request writelocks on the corresponding copies of X. For all four methods, these writelocks conflict with *readlocks* on the same copy, and may also conflict with other *writelocks* on the same copy, depending on the specific ww synchronization technique used by the method.

Since locking conflict rules for writelocks will vary from copy to copy, we distinguish three types. An *rw writelock* only conflicts with readlocks on the same data item. A *ww writelock* only conflicts with ww write-

locks on the same data item. And an *rww writelock* conflicts with readlocks, ww writelocks, and rww writelocks. Thus, using basic 2PL for rw synchronization, every prewrite sets rw writelocks, and may set stronger locks depending on the ww technique.

Method 1: *Basic 2PL for ww synchronization.* All writelocks are rww writelocks; that is, for $i = 1, \ldots, m$, a writelock on x_i conflicts with either a readlock or a writelock on x_i. This is the "standard" distributed implementation of 2PL.

Method 2: *Primary copy 2PL for ww synchronization.* Writelocks only conflict on the primary copy. An rww writelock is used on the primary copy, while rw writelocks are used on the others.

Method 3: *Voting 2PL for ww synchronization.* A DM responds to a prewrite(x_i) by *attempting* to set an rww writelock on x_i. However, if another transaction already owns an rww writelock on x_i, the DM only sets an rw writelock and leaves a request for an rww writelock pending. A transaction can write into any copy of X after it obtains rww writelocks on a majority of copies. This is similar to the method proposed in GIFF79.

Method 4: *Centralized 2PL for ww synchronization.* To write into X, a transaction must first explicitly request a ww writelock on X from a centralized 2PL scheduler. The rw writelocks set by prewrites *never* conflict with each other.

In all four methods, readlocks are explicitly released by lock releases while writelocks are implicitly released by dm-writes. Lock releases may be transmitted in parallel with dm-writes. In Method 4, *after all* dm-writes have been executed, additional lock releases must be sent to the centralized scheduler to release writelocks held there.

5.1.2 Methods Using Primary Copy 2PL for rw Synchronization

Methods 5–8 use primary copy 2PL for rw synchronization. Consider a logical data item X with copies $x_1, \ldots, x_m$, and assume x_1 is the primary copy. To read X, a transaction must obtain a readlock on x_1. It may obtain this lock by issuing a dm-read(x_1). Alternatively, the transaction can send an explicit lock request to x_1's DM; when the lock is granted the transaction can read *any* copy of X.

To write into X, a transaction sends prewrites to *every* DM that stores a copy of X. A prewrite(x_1) implicitly requests an rw writelock. Prewrites on other copies of X may also request writelocks depending on the ww technique.

Method 5: *Basic 2PL for ww synchronization.* For $i = 2, \ldots, m$, prewrite(x_i) requests a ww writelock. Since the writelock on x_1 must also conflict with readlocks on x_1, prewrite(x_1) requests an rww writelock.

Method 6: *Primary copy 2PL for ww synchronization.* Prewrite(x_1) requests an rww writelock on x_1. Prewrites on other copies do not request any locks. This method was originally proposed by STON79 and is used in Distributed INGRES [STON77].

Method 7: *Voting 2PL for ww synchronization.* When a scheduler receives a prewrite(x_i) for $i \neq 1$, it tries to set a ww writelock on x_i. When it receives a prewrite(x_1), it tries to set an rww writelock on x_1; if it cannot, then it sets an rw writelock on x_1 (if possible) before waiting for the rww writelock. A transaction can write into every copy of X after it obtains a ww (or rww) writelock on a majority of copies of X.

Method 8: *Centralized 2PL for ww synchronization.* Transactions obtain ww writelocks from a centralized 2PL scheduler. Thus a prewrite(x_1) requests an rw writelock on x_1; for $i = 2, \ldots, m$, prewrite(x_i) does not request any lock.

Lock releases for Methods 5–8 are handled as in Section 5.1.1.

5.1.3 Methods Using Centralized 2PL for rw Synchronization

The remaining 2PL methods use centralized 2PL for rw synchronization. Before reading (or writing) any copy of logical data item X, a transaction must obtain a readlock (or rw writelock) on X from a centralized 2PL scheduler. Before writing X, the transaction must also send prewrites to every DM that stores a copy of X. Some of these prewrites implicitly request ww

writelocks on copies of X, depending on the specific method.

Method 9: *Basic 2PL for ww synchronization.* Every prewrite requests a ww writelock.

Method 10: *Primary copy 2PL for ww synchronization.* If x_1 is the primary copy of X, a prewrite(x_1) requests a ww writelock. Prewrites on other copies do not request any writelocks.

Method 11: *Voting 2PL for ww synchronization.* Every prewrite *attempts* to set a ww writelock. A transaction can write into every copy of X after it obtains ww writelocks on a majority of copies of X.

Method 12: *Centralized 2PL for ww synchronization.* All locks are obtained at the centralized 2PL scheduler. Before writing into any copy of X, an rww writelock on X is obtained from the centralized scheduler. Prewrites set no locks at all. Method 12 is the "standard" implementation of centralized 2PL (called *primary site* in ALSB76a).

Lock releases for Methods 9–12 are handled as in Section 5.1.1.

5.2 Pure T/O Methods

The T/O synchronization techniques of Section 5 can also be integrated to form 12 *principal T/O methods*:

Method	rw technique	ww technique
1	Basic T/O	Basic T/O
2	Basic T/O	Thomas Write Rule (TWR)
3	Basic T/O	Multiversion T/O
4	Basic T/O	Conservative T/O
5	Multiversion T/O	Basic T/O
6	Multiversion T/O	TWR
7	Multiversion T/O	Multiversion T/O
8	Multiversion T/O	Conservative T/O
9	Conservative T/O	Basic T/O
10	Conservative T/O	TWR
11	Conservative T/O	Multiversion T/O
12	Conservative T/O	Conservative T/O

(That there are also 12 2PL methods is coincidental.)

Each T/O method that incorporates a conservative component can be refined by including classes and conflict graph analysis (see Sections 4.4.2 and 4.4.3).

The interface between rw and ww synchronization techniques is even simpler for T/O methods than for 2PL. The only requirement is that both techniques use the *same* timestamp for any given transaction.

5.2.1 Methods Using Basic T/O for rw Synchronization

Methods 1–4 use basic T/O for rw synchronization. All four methods require R-ts's for each data item. Methods 1, 2, and 4 require W-ts's, while in Method 3 each data item has a set of timestamped versions; for Method 3, let W-ts(x) denote x's largest timestamp. Each method buffers dm-reads and prewrites for two-phase commitment purposes; let min-R-ts(x) and min-P-ts(x) be the minimum timestamps of any buffered dm-read(x) and prewrite(x), respectively.

These methods can be described by the following steps. Let R be a dm-read(x), P a prewrite(x), and W a dm-write(x).

1. If ts(R) < W-ts(x), R is rejected. Else if ts(R) > min-P-ts(x), R is buffered. Else R is output and R-ts(x) is set to max(R-ts(x), ts(R)).
2. If ts(P) < R-ts(x) or *condition* (A)[6] holds, P is rejected. Else P is buffered.
3. If ts(W) > min-R-ts(x) or *condition* (B)[6] holds, W is buffered. Else W is output and W-ts(x) is set to max(W-ts(x), ts(W)). For Method 3, a new version of x is created with timestamp ts(W).
4. When W is output, its prewrite is debuffered and the buffered dm-reads and dm-writes are retested to see if any can now be output.

Method 1: *Basic T/O for ww synchronization.* Condition (A) is ts(P) < W-ts(x) and condition (B) is ts(W) > min-P-ts(x). Note that min-R-ts(x) > min-P-ts(x), since R is buffered only if ts(R) > min-P-ts(x). Also, when W is output, ts(W) > W-ts(x), since condition (B) forces dm-writes on a given x to be output in timestamp order. Thus step 3 simplifies to

3. If ts(W) > min-P-ts(x), W is buffered. Else W is output and W-ts(x) is set to ts(W).

Method 2: *TWR for ww synchronization.* Conditions (A) and (B) are null. However,

[6] Conditions (A) and (B) are determined by the new technique. See the following.

if ts(W) < W-ts(x), W has no effect on the database.

Method 3: *Multiversion T/O for ww synchronization.* Like Method 2 except that W always creates a new version of x.

Method 4: *Conservative T/O for ww synchronization.* Condition (A) is null. For each TM, let min-W-ts(TM) be the minimum timestamp of any buffered dm-write from that TM. Condition (B) is ts(W) > min-W-ts(TM) for some TM. As in Method 1, this causes dm-writes on a given x to be output in timestamp order, and step 3 simplifies to

3. If ts(W) > min-R-ts(x) or ts(W) > min-W-ts(TM) for some TM, W is buffered. Else W is output and W-ts(x) is set to ts(W).

5.2.2 Methods Using Multiversion T/O for rw Synchronization

Methods 5–8 use multiversion T/O for rw synchronization and require a set of R-ts's and a set of versions for each data item. These methods can be described by the following steps. Define R, P, W, min-R-ts, min-W-ts, and min-P-ts as above; let interval(P) be the interval from ts(P) to the smallest W-ts(x) > ts(P).

1. R is never rejected. If ts(R) lies in interval(prewrite(x)) for some buffered prewrite(x), then R is buffered. Else R is output and ts(R) is added to x's set of R-ts's.
2. If some R-ts(x) lies in interval(P) or *condition* (A) holds, then P is rejected. Else P is buffered.
3. If *condition* (B) holds, W is buffered. Else W is output and creates a new version of x with timestamp ts(W).
4. When W is output, its prewrite is debuffered, and buffered dm-reads and dm-writes are retested.

Method 5: *Basic T/O for ww synchronization.* Condition (A) is ts(P) < max-W-ts(x) and condition (B) is ts(W) > min-P-ts(x). Condition (A) implies that interval(P) = (ts(P), ∞); some R-ts(x) lies in that interval if and only if ts(P) < maximum R-ts(x). Thus step 2 simplifies to

2. If ts(P) < max W-ts(x) or ts(P) < max R-ts(x), then P is rejected. Else it is buffered.

Because of this simplification, the method only requires that the maximum R-ts(x) be stored.

Condition (B) forces dm-writes on a given data item to be output in timestamp order. This supports a systematic technique for "forgetting" old versions. Let max-W-ts(x) be the maximum W-ts(x) and let min-ts be the minimum of max-W-ts(x) over all data items in the database. No dm-write with timestamp less than min-ts can be output in the future. Therefore, insofar as update transactions are concerned, we can safely forget all versions timestamped less than min-ts. TMs should be kept informed of the current value of min-ts and queries (read-only transactions) should be assigned timestamps greater than min-ts. Also, after a new min-ts is selected, older versions should not be forgotten immediately, so that active queries with smaller timestamps have an opportunity to finish.

Method 6: *TWR for ww synchronization.* This method is *incorrect.* TWR requires that W be ignored if ts(W) < max W-ts(x). This may cause later dm-reads to be read incorrect data. See Figure 15. (Method 6 is the only incorrect method we will encounter.)

Method 7: *Multiversion T/O for ww synchronization.* Conditions (A) and (B) are null. Note that this method, unlike all previous ones, never buffers dm-writes.

Method 8: *Conservative T/O for ww synchronization.* Condition (A) is null. Condition (B) is ts(W) > min-W-ts(TM) for some TM. Condition (B) forces dm-writes to be output in timestamp order, implying interval(P) = (ts(P), ∞). As in Method 5, this simplifies step 2:

2. If ts(P) < max R-ts(x), P is rejected; else it is buffered.

Like Method 5, this method only requires that the maximum R-ts(x) be stored, and it supports systematic "forgetting" of old versions described above.

5.2.3 Methods Using Conservative T/O for rw Synchronization

The remaining T/O methods use conservative T/O for rw synchronization. Methods

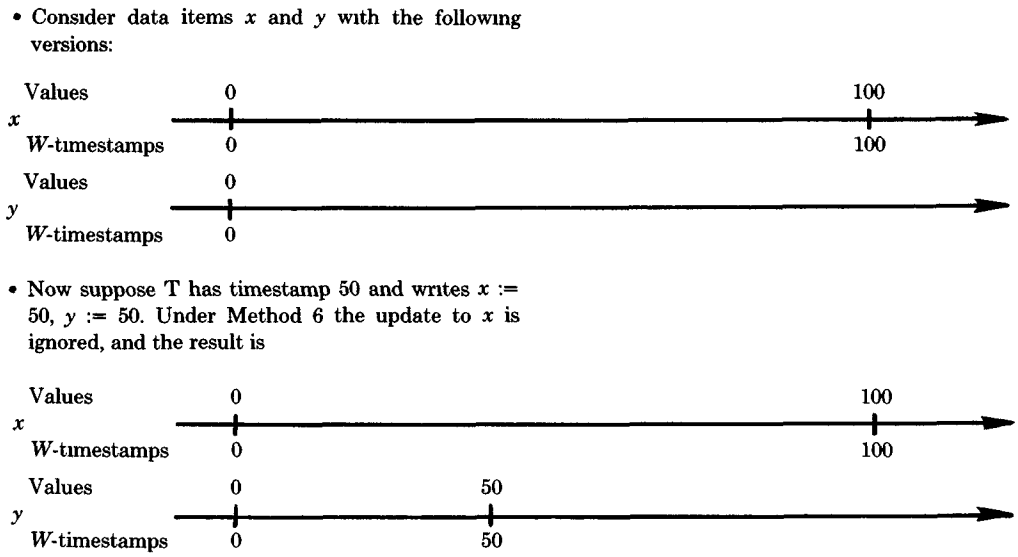


Figure 15. Inconsistent retrievals in Method 6.

9 and 10 require W-ts's for each data item, and Method 11 requires a set of versions for each data item. Method 12 needs no data item timestamps at all. Define R, P, W and min-P-ts as in Section 5.2.1; let min-R-ts(TM) (or min-W-ts(TM)) be the minimum timestamp of any buffered dm-read (or dm-write) from TM.

1. If ts(R) > min-W-ts(TM) for any TM, R is buffered; else it is output.
2. If *condition* (A) holds, P is rejected. Else P is buffered.
3. If ts(W) > min-R-ts(TM) for any TM or *condition* (B) holds, W is buffered. Else W is output.
4. When W is output, its prewrite is debuffered. When R or W is output or buffered, buffered dm-reads and dm-writes are retested to see if any can now be output.

Method 9: *Basic T/O for ww synchronization.* Condition (A) is ts(P) < W-ts(x), and condition (B) is ts(W) > min-P-ts(x).

Method 10: *TWR for ww synchronization.* Conditions (A) and (B) are null. However, if ts(W) < W-ts(x), W has no effect on the database.

This method is essentially the SDD-1 concurrency control [BERN80d], although in SDD-1 the method is refined in several ways. SDD-1 uses classes and conflict graph analysis to reduce communication and increase the level of concurrency. Also, SDD-1 requires predeclaration of read-sets and only enforces the conservative scheduling on dm-reads. By doing so, it forces dm-reads to wait for dm-writes, but does not insist that dm-writes wait for all dm-reads with smaller timestamps. Hence dm-reads can be rejected in SDD-1.

Method 11: *Multiversion T/O for ww synchronization.* Conditions (A) and (B) are null. When W is output, it creates a new version of x with timestamp ts(W). When R is output it reads the version with largest timestamp less than ts(R).

This method can be optimized by noting the multiversion T/O "automatically" prevents dm-reads from being rejected, and makes it unnecessary to buffer dm-writes. Thus step 3 can be simplified to

3. W is output immediately.

Method 12: *Conservative T/O for ww synchronization.* Condition (A) is null; con-

dition (B) is ts(W) > min-W-ts(TM) for some TM. The effect is to output W if the scheduler has received *all* operations with timestamps less than ts(W) that it will ever receive. Method 12 has been proposed in CHEN80, KANE79, and SHAP77a.

5.3 Mixed 2PL and T/O Methods

The major difficulty in constructing methods that combine 2PL and T/O lies in developing the interface between the two techniques. Each technique guarantees an acyclic $\rightarrow_{rwr}$ (or $\rightarrow_{ww}$) relation when used for rw (or ww) synchronization. The interface between a 2PL and a T/O technique must guarantee that the combined $\rightarrow$ relation (i.e., $\rightarrow_{rwr} \cup \rightarrow_{ww}$) remains acyclic. That is, the interface must ensure that the serialization order induced by the rw technique is consistent with that induced by the ww technique. In Section 5.3.1 we describe an interface that makes this guarantee. Given such an interface, *any* 2PL technique can be integrated with *any* T/O technique. Sections 5.3.2 and 5.3.3 describe such methods.

5.3.1 The Interface

The serialization order induced by any 2PL technique is determined by the locked points of the transactions that have been synchronized (see Section 3). The serialization order induced by any T/O technique is determined by the timestamps of the synchronized transactions. So to interface 2PL and T/O we *use locked points to induce timestamps* [BERN80b].

Associated with each data item is a *lock timestamp*, L-ts(x). When a transaction T sets a lock of x, it simultaneously retrieves L-ts(x). When T reaches its locked point it is assigned a timestamp, ts(T), greater than any L-ts it retrieved. When T releases its lock on x, it updates L-ts(x) to be max(L-ts(x), ts(T)).

Timestamps generated in this way are consistent with the serialization order induced by 2PL. That is, ts(T_j) < ts(T_k) if T_j must precede T_k in any serialization induced by 2PL. To see this, let T_1 and T_n be a pair of transactions such that T_1 must precede T_n in any serialization. Thus there exist transactions $T_1, T_2 \ldots, T_{n-1}, T_n$ such that for $i = 1, \ldots, n$-1 (a) T_i's locked point precedes T_{i+1}'s locked point, and (b) T_i released a lock on some data item x before T_{i+1} obtained a lock on x. Let L be the L-ts(x) retrieved by T_{i+1}. Then ts(T_i) < L < ts(T_{i+1}), and by induction ts(T_1) < ts(T_n).

5.3.2 Mixed Methods Using 2PL for rw Synchronization

There are 12 principal methods in which 2PL is used for rw synchronization and T/O is used for ww synchronization:

Method	rw technique	ww technique
1	Basic 2PL	Basic T/O
2	Basic 2PL	TWR
3	Basic 2PL	Multiversion T/O
4	Basic 2PL	Conservative T/O
5	Primary copy 2PL	Basic T/O
6	Primary copy 2PL	TWR
7	Primary copy 2PL	Multiversion T/O
8	Primary copy 2PL	Conservative T/O
9	Centralized 2PL	Basic T/O
10	Centralized 2PL	TWR
11	Centralized 2PL	Multiversion T/O
12	Centralized 2PL	Conservative T/O

Method 2 best exemplifies this class of methods, and it is the only one we describe in detail. Method 2 requires that every stored data item have an L-ts and a W-ts. (One timestamp can serve both roles, but we do not consider this optimization here.)

Let X be a logical data item with copies $x_1, \ldots, x_m$. To read X, transaction T issues a dm-read on any copy of X, say x_i. This dm-read implicitly requests a readlock on x_i, and when the readlock is granted, L-ts(x_i) is returned to T. To write into X, T issues prewrites on every copy of X. These prewrites implicitly request rw writelocks on the corresponding copies, and as each writelock is granted, the corresponding L-ts is returned to T. When T has obtained all of its locks, ts(T) is calculated as in Section 5.3.1. T attaches ts(T) to its dm-writes, which are then sent.

Dm-writes are processed using TWR. Let W be dm-write(x_j). If ts(W) > W-ts(x_j), the dm-write is processed as usual (x_j is updated). If, however, ts(W) < W-ts(x_j), W is *ignored*.

The interesting property of this method is that *writelocks never conflict with writelocks*. The writelocks obtained by prewrites are only used for rw synchronization, and only conflict with readlocks. This permits

transactions to execute concurrently to completion even if their writesets intersect. Such concurrency is never possible in a pure 2PL method.

5.3.3 Mixed Methods Using T/O for rw Synchronization

There are also 12 principal methods that use T/O for rw synchronization and 2PL for ww synchronization:

Method	rw technique	ww technique
13	Basic T/O	Basic 2PL
14	Basic T/O	Primary copy 2PL
15	Basic T/O	Voting 2PL
16	Basic T/O	Centralized 2PL
17	Multiversion T/O	Basic 2PL
18	Multiversion T/O	Primary copy 2PL
19	Multiversion T/O	Voting 2PL
20	Multiversion T/O	Centralized 2PL
21	Conservative T/O	Basic 2PL
22	Conservative T/O	Primary copy 2PL
23	Conservative T/O	Voting 2PL
24	Conservative T/O	Centralized 2PL

These methods all require *predeclaration of writelocks.* Since T/O is used for rw synchronization, transactions must be assigned timestamps before they issue dm-reads. However, the timestamp generation technique of Section 5.3.1 requires that a transaction be at its locked point before it is assigned its timestamp. Hence every transaction must be at its locked point before it issues any dm-reads; in other words, every transaction must obtain all of its writelocks before it begins its main execution.

To illustrate these methods, we describe Method 17. This method requires that each stored data item have a set of R-ts's and a set of ⟨W-ts, value⟩ pairs (i.e., versions). The L-ts of any data item is the maximum of its R-ts's and W-ts's.

Before beginning its main execution, transaction T issues prewrites on every copy of every data item in its writeset.[7] These prewrites play a role in ww synchronization, rw synchronization, and the interface between these techniques.

Let P be a prewrite(x). The ww role of P is to request a ww writelock on x. When the lock is granted, L-ts(x) is returned to T; this is the interface role of P. Also when the lock is granted, P is buffered and the rw synchronization mechanism is informed that a dm-write with timestamp greater than L-ts(x) is pending. This is its rw role.

When T has obtained all of its writelocks, ts(T) is calculated as in Section 5.3.1 and T begins its main execution. T attaches ts(T) to its dm-reads and dm-writes and rw synchronization is performed by multiversion T/O, as follows:

1. Let R be a dm-read(x). If there is a buffered prewrite(x) (other than one issued by T), and if L-ts(x) < ts(T), then R is buffered. Else R is output and reads the version of x with largest timestamp less than ts(T).
2. Let W be a dm-write(x). W is output immediately and creates a new version of x with timestamp ts(T).
3. When W is output, its prewrite is debuffered, and its writelock on x is released. This causes L-ts(x) to be updated to max(L-ts(x), ts(T)) = ts(T).

One interesting property of this method is that restarts are needed only to prevent or break deadlocks caused by ww synchronization; rw conflicts never cause restarts. This property cannot be attained by a pure 2PL method. It can be attained by pure T/O methods, but only if conservative T/O is used for rw synchronization; in many cases conservative T/O introduces excessive delay or is otherwise infeasible.

The behavior of this method for queries is also interesting. Since queries set no writelocks, the timestamp generation rule does not apply to them. Hence *the system is free to assign any timestamp it wishes to a query.* It may assign a small timestamp, in which case the query will read old data but is unlikely to be delayed by buffered prewrites; or it may assign a large timestamp, in which case the query will read current data but is more likely to be delayed. No matter which timestamp is selected, however, *a query can never cause an update to be rejected.* This property cannot be easily attained by any pure 2PL or T/O method.

We also observe that this method creates versions in timestamp order, and so sys-

[7] Since new values for the data items in the writeset are not yet known, these prewrites do not instruct DMs to store values on secure storage, they merely "warn" DMs to "expect" the corresponding dm-writes See footnote 3.

tematic forgetting of old versions is possible (see Section 5.2.2). In addition, the method requires only *maximum* R-ts's; smaller ones may be instantly forgotten.

CONCLUSION

We have presented a framework for the design and analysis of distributed database concurrency control algorithms. The framework has two main components: (1) a system model that provides common terminology and concepts for describing a variety of concurrency control algorithms, and (2) a problem decomposition that decomposes concurrency control algorithms into read-write and write-write synchronization subalgorithms.

We have considered synchronization subalgorithms outside the context of specific concurrency control algorithms. Virtually all known database synchronization algorithms are variations of two basic techniques—two-phase locking (2PL) and timestamp ordering (T/O). We have described the principal variations of each technique, though we do not claim to have exhausted all possible variations. In addition, we have described ancillary problems (e.g., deadlock resolution) that must be solved to make each variation effective.

We have shown how to integrate the described techniques to form complete concurrency control algorithms. We have listed 47 concurrency control algorithms, describing 25 in detail. This list includes almost all concurrency control algorithms described previously in the literature, plus several new ones. This extreme consolidation of the state of the art is possible in large part because of our framework set up earlier.

The focus of this paper has primarily been the *structure* and *correctness* of synchronization techniques and concurrency control algorithms. We have left open a very important issue, namely, performance.

The main performance metrics for concurrency control algorithms are system throughput and transaction response time. Four cost factors influence these metrics: intersite communication, local processing, transaction restarts, and transaction blocking. The impact of each cost factor on system throughput and response time varies from algorithm to algorithm, system to system, and application to application. This impact is not understood in detail, and a comprehensive quantitative analysis of performance is beyond the state of the art. Recent theses by Garcia-Molina [GARC79a] and Reis [REIS79a] have taken first steps toward such an analysis but there clearly remains much to be done.

We hope, and indeed recommend, that future work on distributed concurrency control will concentrate on the performance of algorithms. There are, as we have seen, many known methods; the question now is to determine which are best.

APPENDIX. OTHER CONCURRENCY CONTROL METHODS

In this appendix we describe three concurrency control methods that do not fit the framework of Sections 3–5: the certifier methods of Badal [BADA79], Bayer et al. [BAYE80], and Casanova [CASA79], the majority consensus algorithm of Thomas [THOM79], and the ring algorithm of Ellis [ELLI77]. We argue that these methods are not practical in DDBMSs. The certifier methods look promising for *centralized* DBMSs, but severe technical problems must be overcome before these methods can be extended correctly to distributed systems. The Thomas and Ellis algorithms, by contrast, are among the earliest algorithms proposed for DDBMS concurrency control. These algorithms introduced several important techniques into the field but, as we will see, have been surpassed by recent developments.

A1. Certifiers

A1.1 The Certification Approach

In the certification approach, dm-reads and prewrites are processed by DMs first-come/first-served, with no synchronization whatsoever. DMs do maintain summary information about rw and ww conflicts, which they update every time an operation is processed. However, dm-reads and prewrites are never blocked or rejected on the basis of the discovery of such a conflict.

Synchronization occurs when a transaction attempts to terminate. When a trans-

action T issues its END, the DBMS decides whether or not to *certify*, and thereby commit, T.

To understand how this decision is made, we must distinguish between "total" and "committed" executions. A *total execution* of transactions includes the execution of all operations processed by the system up to a particular moment. The *committed execution* is the portion of the total execution that only includes dm-reads and dm-writes processed on behalf of committed transactions. That is, the committed execution is the total execution that would result from aborting all active transactions (and not restarting them).

When T issues its END, the system tests whether the committed execution augmented by T's execution is serializable, that is, whether after committing T the resulting committed execution would still be serializable. If so, T is committed; otherwise T is restarted.

There are two properties of certification that distinguish it from other approaches. First, synchronization is accomplished entirely by restarts, never by blocking. And second, the decision to restart or not is made *after* the transaction has finished executing. No concurrency control method discussed in Sections 3–5 satisifies both these properties.

The rationale for certification is based on an optimistic assumption regarding run-time conflicts: if very few run-time conflicts are expected, assume that most executions are serializable. By processing dm-reads and prewrites without synchronization, the concurrency control method never delays a transaction while it is being processed. Only a (fast, it is hoped) certification test when the transaction terminates is required. Given optimistic transaction behavior, the test will usually result in committing the transaction, so there are very few restarts. Therefore certification simultaneously avoids blocking and restarts in optimistic situations.

A certification concurrency control method must include a *summarization algorithm* for storing information about dm-reads and prewrites when they are processed and a *certification algorithm* for using that information to certify transactions when they terminate. The main problem in the summarization algorithm is avoiding the need to store information about already-certified transactions. The main problem in the certification algorithm is obtaining a *consistent* copy of the summary information. To do so the certification algorithm often must perform some synchronization of its own, the cost of which must be included in the cost of the entire method.

A1.2 Certification Using the → Relation

One certification method is to construct the → relation as dm-reads and prewrites are processed. To certify a transaction, the system checks that → is acyclic [BADA79, BAYE80, CASA79].[8]

To construct →, each site remembers the most recent transaction that read or wrote each data item. Suppose transactions T_i and T_j were the last transactions to (respectively) read and write data item x. If transaction T_k now issues a dm-read(x), $T_j \rightarrow T_k$ is added to the summary information for the site and T_k replaces T_i as the last transaction to have read x. Thus pieces of → are distributed among the sites, reflecting run-time conflicts at each site.

To certify a transaction, the system must check that the transaction does not lie on a cycle in → (see Theorem 2, Section 2). Guaranteeing acyclicity is sufficient to guarantee serializability.

There are two problems with this approach. First, it is in general not correct to delete a certified transaction from →, even if all of its updates have been committed. For example, if $T_i \rightarrow T_j$ and T_i is active but T_j is committed, it is still possible for $T_j \rightarrow T_i$ to develop; deleting T_j would then cause the cycle $T_i \rightarrow T_j \rightarrow T_i$ to go unnoticed when T_i is certified. However, it is obviously not feasible to allow → to grow indefinitely. This problem is solved by Casanova [CASA79] by a method of encoding information about committed transactions in space proportional to the number of active transactions.

A second problem is that *all* sites must be checked to certify *any* transaction. Even

[8] In BAYE80 certification is only used for rw synchronization whereas 2PL is used for ww synchronization.

sites at which the transaction never accessed data must participate in the cycle checking of $\rightarrow$. For example, suppose we want to certify transaction T. T might be involved in a cycle $T \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_{n-1} \rightarrow T_n \rightarrow T$, where each conflict $T_k \rightarrow T_{k+1}$ occurred at a different site. Possibly T only accessed data at one site; yet the $\rightarrow$ relation must be examined at n sites to certify T. This problem is currently unsolved, as far as we know. That is, any *correct* certifier based on this approach of checking cycles in $\rightarrow$ must access the $\rightarrow$ relation at *all* sites to certify each and every transaction. Until this problem is solved, we judge the certification approach to be impractical in a distributed environment.

A2. Thomas' Majority Consensus Algorithm

A2.1 The Algorithm

One of the first published algorithms for distributed concurrency control is a certification method described in THOM79. Thomas introduced several important synchronization techniques in that algorithm, including the Thomas Write Rule (Section 3.2.3), majority voting (Section 3.1.1), and certification (Appendix A1). Although these techniques are valuable when considered in isolation, we argue that the overall Thomas algorithm is not suitable for distributed databases. We first describe the algorithm and then comment on its application to distributed databases.

Thomas' algorithm assumes a fully redundant database, with every logical data item stored at every site. Each copy carries the timestamp of the last transaction that wrote into it.

Transactions execute in two phases. In the first phase each transaction executes locally at one site called the transaction's *home site*. Since the database is fully redundant, any site can serve as the home site for any transaction. The transaction is assigned a unique timestamp when it begins executing. During execution it keeps a record of the timestamp of each data item it reads and, when its executes a write on a data item, processes the write by recording the new value in an *update list*. Note that each transaction must read a copy of a data item before it writes into it. When the transaction terminates, the system augments the update list with the list of data items read and their timestamps at the time they were read. In addition, the timestamp of the transaction itself is added to the update list. This completes the first phase of execution.

In the second phase the update list is sent to every site. Each site (including the site that produced the update list) *votes* on the update list. Intuitively speaking, a site votes yes on an update list if it can certify the transaction that produced it. After a site votes yes, the update list is said to be *pending* at that site. To cast the vote, the site sends a message to the transaction's home site, which, when it receives a majority of yes or no votes, informs all sites of the outcome. If a majority voted yes, then all sites are required to commit the update, which is then installed using TWR. If a majority voted no, all sites are told to discard the update, and the transaction is restarted.

The rule that determines when a site may vote "yes" on a transaction is pivotal to the correctness of the algorithm. To vote on an update list U, a site compares the timestamp of each data item in the readset of U with the timestamp of that same data item in the site's local database. If any data item has a timestamp in the database different from that in U, the site votes *no*. Otherwise, the site compares the readset and writeset of U with the readset and writeset of each pending update list at that site, and if there is no rw conflict between U and any of the pending update lists, it votes *yes*. If there is an rw conflict between U and one of those pending requests, the site votes *pass* (abstain) if U's timestamp is larger than that of all pending update lists with which it conflicts. If there is an rw conflict but U's timestamp is smaller than that of the conflicting pending update list, then it sets U aside on a *wait queue* and tries again when the conflicting request has either been committed or aborted at that site.

The voting rule is essentially a certification procedure. By making the timestamp comparison, a site is checking that the readset was not written into since the transaction read it. If the comparisons are satisfied, the situation is as if the transaction had locked its readset at that site and held the locks until it voted. The voting rule is

thereby guaranteeing rw synchronization with a certification rule approximating rw 2PL. (This fact is proved precisely in BERN79b.)

The second part of the voting rule, in which U is checked for rw conflicts against pending update lists, guarantees that conflicting requests are not certified concurrently. An example illustrates the problem. Suppose T_1 reads X and Y, and writes Y, while T_2 reads X and Y, and writes X. Suppose T_1 and T_2 execute at sites A and B, respectively, and X and Y have timestamps of 0 at both sites. Assume that T_1 and T_2 execute concurrently and produce update lists ready for voting at about the same time. Either T_1 or T_2 must be restarted, since neither read the other's output; if they were both committed, the result would be nonserializable. However both T_1's and T_2's update lists will (concurrently) satisfy the timestamp comparison at both A and B. What stops them from both obtaining unanimous yes votes is the second part of the voting rule. After a site votes on one of the transactions, it is prevented from voting on the other transaction until the first is no longer pending. Thus it is not possible to certify conflicting transactions concurrently. (We note that this problem of concurrent certification exists in the algorithms of Section A1.2, too. This is yet another technical difficulty with the certification approach in a distributed environment.)

With the second part of the voting rule, the algorithm behaves as if the certification step were atomically executed at a primary site. If certification were centralized at a primary site, the certification step at the primary site would serve the same role as the majority decision in the voting case.

A2.2 Correctness

No simple proof of the serializability of Thomas' algorithm has ever been demonstrated, although Thomas provided a detailed "plausibility" argument in THOM79. The first part of the voting rule can correctly be used in a centralized concurrency control method since it implies 2PL [BERN79b], and a centralized method based on this approach was proposed in KUNG81. The second part of the voting rule guarantees that for every pair of conflicting transactions that received a majority of yes votes, all sites that voted yes on both transactions voted on the two transactions in the same order. This makes the certification step behave just as it would if it were centralized, thereby avoiding the problem exemplified in the previous paragraph.

A2.3 Partially Redundant Databases

For the majority consensus algorithm to be useful in a distributed database environment, it must be generalized to operate correctly when the database is only partially redundant. There is reason to doubt that such a generalization can be accomplished without either serious degradation of performance or a complete change in the set of techniques that are used.

First, the majority consensus decision rule apparently must be dropped, since the voting algorithm depends on the fact that all sites perform exactly the same certification test. In a partially redundant database, each site would only be comparing the timestamps of the data items stored at that site, and the significance of the majority vote would vanish.

If majority voting cannot be used to synchronize concurrent certification tests, apparently some kind of mutual exclusion mechanism must be used instead. Its purpose would be to prevent the concurrent, and therefore potentially incorrect, certification of two conflicting transactions, and would amount to locking. The use of locks for synchronizing the certification step is not in the spirit of Thomas' algorithm, since a main goal of the algorithm was to avoid locking. However, it is worth examining such a locking mechanism to see how certification can be correctly accomplished in a partially redundant database.

To process a transaction T, a site produces an update list as usual. However, since the database is partially redundant, it may be necessary to read portions of T's readset from other sites. After T terminates, its update list is sent to every site that contains part of T's readset or writeset. To certify an update list, a site first sets local locks on the readset and writeset, and then (as in the fully redundant case) it

compares the update list's timestamps with the database's timestamps. If they are identical, it votes yes; otherwise it votes no. A unanimous vote of yes is needed to commit the updates. Local locks cannot be released until the voting decision is completed.

While this version of Thomas' algorithm for partially redundant data works correctly, its performance is inferior to standard 2PL. This algorithm requires that the same locks be set as in 2PL, and the same deadlocks can arise. Yet the probability of restart is higher than in 2PL, because even after all locks are obtained the certification step can still vote no (which cannot happen in 2PL).

One can improve this algorithm by designating a primary copy of each data item and only performing the timestamp comparison against the primary copy, making it analogous to primary copy 2PL. However, for the same reasons as above, we would expect primary copy 2PL to outperform this version of Thomas' algorithm too.

We therefore must leave open the problem of producing an efficient version of Thomas' algorithm for a partially redundant database.

A2.4 Performance

Even in the fully redundant case, the performance of the majority consensus algorithm is not very good. First, repeating the certification and conflict detection at each site is more than is needed to obtain serializability: a centralized certifier would work just as well and would only require that certification be performed at one site. Second, the algorithm is quite prone to restarts when there are run-time conflicts, since restarts are the only tactic available for synchronizing transactions, and so will only perform well under the most optimistic circumstances. Finally, even in optimistic situations, the analysis in GARC79a indicates that centralized 2PL outperforms the majority consensus algorithm.

A2.5 Reliability

Despite the performance problems of the majority consensus algorithm, one can try to justify the algorithms on reliability grounds. As long as a majority of sites are correctly running, the algorithm runs smoothly. Thus, handling a site failure is free, insofar as the voting procedure is concerned.

However, from current knowledge, this justification is not compelling for several reasons. First, although there is no cost when a site fails, substantial effort may be required when a site recovers. A centralized algorithm using backup sites, as in ALSB76a, lacks the symmetry of Thomas' algorithm, but may well be more efficient due to the simplicity of site recovery. In addition, the majority consensus algorithm does not consider the problem of atomic commitment and it is unclear how one would integrate two-phase commit into the algorithm.

Overall, the reliability threats that are handled by the majority consensus algorithm have not been explicitly listed, and alternative solutions have not been analyzed. While voting is certainly a possible technique for obtaining a measure of reliability, the circumstances under which it is cost-effective are unknown.

A3. Ellis' Ring Algorithm

Another early solution to the problem of distributed database concurrency control is the ring algorithm [ELLI77]. Ellis was principally interested in a proof technique, called *L systems*, for proving the correctness of concurrent algorithms. He developed his concurrency control method primarily as an example to illustrate L-system proofs, and never made claims about its performance. Because the algorithm was only intended to illustrate mathematical techniques, Ellis imposed a number of restrictions on the algorithm for mathematical convenience, which make it infeasible in practice. Nonetheless, the algorithm has received considerable attention in the literature, and in the interest of completeness, we briefly discuss it.

Ellis' algorithm solves the distributed concurrency control problem with the following restrictions:

(1) The database must be fully redundant.
(2) The communication medium must be a ring, so each site can only communicate with its successor on the ring.

(3) Each site-to-site communication link is pipelined.
(4) Each site can supervise no more than one active update transaction at a time.
(5) To update any copy of the database, a transaction must first obtain a lock on the entire database at all sites.

The effect of restriction 5 is to force all transactions to execute *serially*; no concurrent processing is ever possible. For this reason alone, the algorithm is fundamentally impractical.

To execute, an update transaction migrates around the ring, (essentially) obtaining a lock *on the entire database at each site*. However, the lock conflict rules are nonstandard. A lock request from a transaction that originated at site A conflicts at site C with a lock held by a transaction that originated from site B if B = C and either A = B or A's priority < B's priority. The daisy-chain communication induced by the ring combined with this locking rule produces a deadlock-free algorithm that does not require deadlock detection and never induces restarts. A detailed description of the algorithm appears in GARC79a.

There are several problems with this algorithm in a distributed database environment. First, as mentioned above, it forces transactions to execute serially. Second, it only applies to a fully redundant database. And third, the daily-chain communication requires that each transaction obtain its lock at one site at a time, which causes communication delay to be (at least) linearly proportional to the number of sites in the system.

A modified version of Ellis' algorithm that mitigates the first problem is proposed in GARC79a. Even with this improvement, performance analysis indicates that the ring algorithm is inferior to centralized 2PL. And, of course, the modified algorithm still suffers from the last two problems.

ACKNOWLEDGMENT

This work was supported by Rome Air Development Center under contract F30602-79-C-0191.

REFERENCES

AHO75 AHO, A. V., HOPCROFT, E., AND ULLMAN, J. D. *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., 1975.

ALSB76a ALSBERG, P. A , AND DAY, J. D. "A principle for resilient sharing of distributed resources," in *Proc. 2nd Int. Conf. Software Eng.*, Oct. 1976, pp. 562–570.

ALSB76b ALSBERG, P. A., BELFORD, G.C., DAY, J. D., AND GRAPLA, E. "Multi-copy resiliency techniques," Center for Advanced Computation, AC Document No. 202, Univ. Illinois at Urbana-Champaign, May 1976.

BADA78 BADAL, D. Z., AND POPEK, G. J. "A proposal for distributed concurrency control for partially redundant distributed data base system," in *Proc. 3rd Berkeley Workshop Distributed Data Management and Computer Networks*, 1978, pp. 273–288

BADA79 BADAL, D. Z. "Correctness of concurrency control and implications in distributed databases," in *Proc COMPSAC 79 Conf.*, Chicago, Ill., Nov. 1979.

BADA80 BADAL, D. Z. "On the degree of concurrency provided by concurrency control mechanisms for distributed databases," in *Proc. Int Symp. Distributed Databases*, Versailles, France, March 1980.

BAYE80 BAYER, R., HELLER, H., AND REISER, A. "Parallelism and recovery in database systems," *ACM Trans. Database Syst.* **5**, 2 (June 1980), 139–156.

BELF76 BELFORD, G. C., SCHWARTZ, P. M., AND SLUIZER, S. "The effect of back-up strategy on database availability," CAC Document No. 181, CCTCWAD Document No. 5515, Center for Advanced Computation, Univ. Illinois at Urbana-Champaign, Urbana, Feb. 1976.

BERN78a BERNSTEIN, P. A., GOODMAN, N., ROTHNIE, J B., AND PAPADIMITRIOU, C. A. "The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case)," *IEEE Trans. Softw. Eng.* **SE-4**, 3 (May 1978), 154–168.

BERN79a BERNSTEIN, P. A., AND GOODMAN, N. "Approaches to concurrency control in distributed databases," in *Proc. 1979 Natl. Computer Conf.*, AFIPS Press, Arlington, Va., June 1979.

BERN79b BERNSTEIN, P. A., SHIPMAN, D. W., AND WONG, W. S. "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. Softw Eng.* **SE-5**, 3 (May 1979), 203–215.

BERN80a BERNSTEIN, P. A., AND GOODMAN, N. "Timestamp based algorithms for concurrency control in distributed database systems," *Proc 6th Int. Conf. Very Large Data Bases*, Oct. 1980.

BERN80b BERNSTEIN, P. A., GOODMAN, N., AND LAI, M. Y. "Two Part Proof Schema for Database Concurrency Control," in *Proc 5th Berkeley Workshop Distributed Data Management and Computer Networks*, Feb. 1980.

BERN80c BERNSTEIN, P. A , AND SHIPMAN, D. W "The correctness of concurrency

control mechanisms in a system for distributed databases (SDD-1)," in *ACM Trans. Database Syst.* **5**, 1 (March 1980), 52–68.

BERN80d BERNSTEIN, P., SHIPMAN, D. W., AND ROTHNIE, J. B. "Concurrency control in a system for distributed databases (SDD-1)," in *ACM Trans. Database Syst* **5**, 1 (March 1980), 18–51.

BERN81 BERNSTEIN, P. A., GOODMAN, N., WONG, E., REEVE, C. L., AND ROTHNIE, J. B. "Query processing in SDD-1," *ACM Trans. Database Syst.* **6**, 2, to appear.

BREI79 BREITWIESER, H., AND KERSTEN, U. "Transaction and catalog management of the distributed file management system DISCO," in *Proc. Very Large Data Bases*, Rio de Janerio, 1979

BRIN73 BRINCH-HANSEN, P. *Operating system principles*, Prentice-Hall, Englewood Cliffs, N. J., 1973.

CASA79 CASANOVA, M. A. "The concurrency control problem for database systems," Ph.D. dissertation, Harvard Univ., Tech. Rep. TR-17-79, Center for Research in Computing Technology, 1979.

CHAM74 CHAMBERLIN, D. D., BOYCE, R. F., AND TRAIGER, I. L. "A deadlock-free scheme for resource allocation in a database environment," *Info. Proc.* **74**, North-Holland, Amsterdam, 1974.

CHEN80 CHENG, W. K., AND BELFORD, G. C. "Update Synchronization in Distributed Databases," in *Proc. 6th Int. Conf. Very Large Data Bases*, Oct. 1980.

DEPP76 DEPPE, M. E., AND FRY, J. P. "Distributed databases· A summary of research," in *Computer networks*, vol. 1, no. 2, North-Holland, Amsterdam, Sept. 1976.

DIJK71 DIJKSTRA, E. W. "Hierarchical ordering of sequential processes," *Acta Inf.* **1**, 2 (1971), 115–138.

ELLI77 ELLIS, C. A. "A robust algorithm for updating duplicate databases," in *Proc 2nd Berkeley Workshop Distributed Databases and Computer Networks*, May 1977.

ESWA76 ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. "The notions of consistency and predicate locks in a database system." *Commun. ACM* **19**, 11 (Nov. 1976), 624–633.

GARC78 GARCIA-MOLINA, H "Performance comparisons of two update algorithms for distributed databases," in *Proc. 3rd Berkeley Workshop Distributed Databases and Computer Networks*, Aug. 1978.

GARC79a GARCIA-MOLINA, H. "Performance of update algorithms for replicated data in a distributed database," Ph.D. dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif., June 1979.

GARC79b GARCIA-MOLINA, H. "A concurrency control mechanism for distributed data bases which use centralized locking controllers," in *Proc. 4th Berkeley Workshop Distributed Databases and Computer Networks*, Aug. 1979.

GARC79c GARCIA-MOLINA, H. "Centralized control update algorithms for fully redundant distributed databases," in *Proc. 1st Int. Conf. Distributed Computing Systems* (IEEE), New York, Oct. 1979, pp. 699–705.

GARD77 GARDARIN, G., AND LEBAUX, P. "Scheduling algorithms for avoiding inconsistency in large databases," in *Proc. 1977 Int. Conf. Very Large Data Bases* (IEEE), New York, pp. 501–516.

GELE78 GELEMBE, E., AND SEVCIK, K. "Analysis of update synchronization for multiple copy databases," in *Proc. 3rd Berkeley Workshop Distributed Databases and Computer Networks*, Aug. 1978.

GIFF79 GIFFORD, D. K. "Weighted voting for replicated data," in *Proc. 7th Symp. Operating Systems Principles*, Dec. 1979.

GRAY75 GRAY, J. N., LORIE, R. A., PUTZULO, G. R., AND TRAIGER, I. L. "Granularity of locks and degrees of consistency in a shared database," IBM Res. Rep. RJ1654, Sept. 1975.

GRAY78 GRAY, J. N. "Notes on database operating systems," in *Operating Systems: An Advanced Course*, vol. 60, *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1978, pp. 393–481.

HAMM80 HAMMER, M. M., AND SHIPMAN, D. W. "Reliability mechanisms for SDD-1: A system for distributed databases," *ACM Trans. Database Syst.* **5**, 4 (Dec. 1980), 431–466.

HEWI74 HEWITT, C. E. "Protection and synchronization in actor systems," Working Paper No. 83, M.I.T. Artificial Intelligence Lab., Cambridge, Mass., Nov. 1974.

HOAR74 HOARE, C. A. R. "Monitors. An operating system structuring concept," *Commun. ACM* **17**, 10 (Oct. 1974), 549–557.

HOLT72 HOLT, R. C. "Some deadlock properties of computer systems," *Comput. Surv.* **4**, 3 (Dec. 1972) 179–195.

KANE79 KANEKO, A., NISHIHARA, Y., TSURUOKA, K., AND HATTORI, M. "Logical clock synchronization method for duplicated database control," in *Proc. 1st Int. Conf. Distributed Computing Systems* (IEEE), New York, Oct. 1979, pp. 601–611.

KAWA79 KAWAZU, S., MINAMI, ITOH, S., AND TERANAKA, K. "Two-phase deadlock detection algorithm in distributed databases," in *Proc. 1979 Int. Conf. Very Large Data Bases* (IEEE), New York.

KING74 KING, P. F., AND COLLMEYER, A J. "Database sharing—an efficient method for supporting concurrent processes," in *Proc. 1974 Nat. Computer Conf.*, vol. 42, AFIPS Press, Arlington, Va., 1974.

KUNG79 KUNG, H. T., AND PAPADIMITRIOU, C. H. "An optimality theory of concurrency control for databases," in *Proc. 1979*

ACM-SIGMOD Int. Conf Management of Data, June 1979.

KUNG81 KUNG, H. T., AND ROBINSON, J. T. "On optimistic methods for concurrency control," *ACM Trans. Database Syst.* **6**, 2, (June 81), 213-226.

LAMP76 LAMPSON, B., AND STURGIS, H. "Crash recovery in a distributed data storage system," Tech. Rep., Computer Science Lab., Xerox Palo Alto Research Center, Palo Alto, Calif., 1976.

LAMP78 LAMPORT, L. "Time, clocks and ordering of events in a distributed system," *Commun. ACM* **21**, 7 (July 1978), 558-565.

LELA78 LELANN, G. "Algorithms for distributed data-sharing sytems which use tickets," in *Proc. 3rd Berkeley Workshop Distributed Databases and Computer Networks*, Aug. 1978.

LIN79 LIN, W. K. "Concurrency control in multiple copy distributed data base system," in *Proc 4th Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1979.

MENA79 MENASCE, D. A., AND MUNTZ, R. R. "Locking and deadlock detection in distributed databases," *IEEE Trans. Softw. Eng.* **SE-5**, 3 (May 1979), 195-202.

MENA80 MENASCE, D. A., POPEK, G. J., AND MUNTZ, R. R. "A locking protocol for resource coordination in distributed databases," *ACM Trans. Database Syst.* **5**, 2 (June 1980), 103-138.

MINO78 MINOURA, T. "Maximally concurrent transaction processing," in *Proc. 3rd Berkeley Workshop Distributed Databases and Computer Networks*, Aug. 1978.

MINO79 MINOURA, T. "A new concurrency control algorithm for distributed data base systems," in *Proc. 4th Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1979.

MONT78 MONTGOMERY, W. A. "Robust concurrency control for a distributed information system," Ph.D. dissertation, Lab. for Computer Science, M.I.T., Cambridge, Mass , Dec. 1978.

PAPA77 PAPADIMITRIOU, C. H., BERNSTEIN, P A , AND ROTHNIE, J. B. "Some computational problems related to database concurrency control," in *Proc. Conf. Theoretical Computer Science*, Waterloo, Ont., Canada, Aug. 1977.

PAPA79 PAPADIMITRIOU, C. H. "Serializability of concurrent updates," *J. ACM* **26**, 4 (Oct. 1979), 631-653.

RAHI79 RAHIMI, S K., AND FRANTS, W. R. "A posted update approach to concurrency control in distributed database systems," in *Proc. 1st Int. Conf. Distributed Computing Systems* (IEEE), New York, Oct. 1979, pp. 632-641.

RAMI79 RAMIREZ, R. J , AND SANTORO, N. "Distributed control of updates in multiple-copy data bases: A time optimal algorithm," in *Proc. 4th Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1979.

REED78 REED, D. P. "Naming and synchronization in a decentralized computer system, Ph.D. dissertation, Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., Sept., 1978.

REIS79a REIS, D. "The effect of concurrency control on database management system performance," Ph.D. dissertation, Computer Science Dept., Univ. California, Berkeley, April 1979.

REIS79b REIS, D. "The effects of concurrency control on the performance of a distributed database management system," in *Proc. 4th Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1979.

ROSE79 ROSEN, E. C. "The updating protocol of the ARPANET's new routing algorithm: A case study in maintaining identical copies of a changing distributed data base," in *Proc. 4th Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1979.

ROSE78 ROSENKRANTZ, D. J., STEARNS, R E., AND LEWIS, P. M. "System level concurrency control for distributed database systems," *ACM Trans. Database Syst.* **3**, 2 (June 1978), 178-198.

ROTH77 ROTHNIE, J. B., AND GOODMAN, N. "A survey of research and development in distributed databases systems," in *Proc 3rd Int. Conf. Very Large Data Bases* (IEEE), Tokyo, Japan, Oct. 1977.

SCHL78 SCHLAGETER, G. "Process synchronization in database systems." *ACM Trans. Database Syst.* **3**, 3 (Sept. 1978), 248-271.

SEQU79 SEQUIN, J., SARGEANT, G., AND WILNES, P. "A majority consensus algorithm for the consistency of duplicated and distributed information," in *Proc. 1st Int. Conf. Distributed Computing Systems* (IEEE), New York, Oct. 1979, pp. 617-624.

SHAP77a SHAPIRO, R. M., AND MILLSTEIN, R. E. "Reliability and fault recovery in distributed processing," in *Oceans '77 Conf Record*, vol II, Los Angeles, 1977.

SHAP77b SHAPIRO, R. M., AND MILLSTEIN, R. E. "NSW reliability plan," Massachusetts Tech. Rep. 7701-1411, Computer Associates, Wakefield, Mass., June 1977.

SILB80 SILBERSCHATZ, A., AND KEDEM, Z. "Consistency in hierarchical database systems," *J. ACM* **27**, 1 (Jan. 1980), 72-80.

STEA76 STEARNS, R. E., LEWIS, P. M. II, AND ROSENKRANTZ, D. J. "Concurrency controls for database systems," in *Proc. 17th Symp. Foundations Computer Science* (IEEE), 1976, pp. 19-32.

STEA81 STEARNS, R. E., AND ROSENKRANTZ, J. "Distributed database concurrency controls using fore-values," in *Proc 1981 SIGMOD Conf.* (ACM).

STON77 STONEBRAKER, M., AND NEUHOLD, E. "A distributed database version of INGRES," in *Proc. 2nd Berkeley Workshop Distributed Data Management and Computer Networks*, May 1977.

STON79 STONEBRAKER, M. "Concurrency control and consistency of multiple copies of data in distributed INGRES, *IEEE Trans. Softw. Eng.* **SE-5**, 3 (May 1979), 188–194.

THOM79 THOMAS, R. H. "A solution to the concurrency control problem for multiple copy databases," in *Proc. 1978 COMPCON Conf.* (IEEE), New York.

VERH78 VERHOFSTAD, J. S. M. "Recovery and crash resistance in a filing system," in *Proc. SIGMOD Int Conf. Management of Data* (ACM), New York, 1977, pp 158–167.

A Partial Index of References

1. *Certifiers*: BADA79, BAYE80, CASA79, KUNG81, PAPA79, THOM79
2. *Concurrency control theory*: BERN79b, BERN80c, CASA79, ESWA76, KUNG79, MINO78, PAPA77, PAPA79, SCHL78, SILB80, STEA76
3. *Performance*: BADA80, GARC78, GARC79a, GARC79b, GELE78, REIS79a, REIS79b, ROTH77
4. *Reliability*
   General: ALSB76a, ALSB76b, BELF76, BERN79a, HAMM80, LAMP76
   Two-phase commit: HAMM80, LAMP76
5. *Timestamp-ordered scheduling (T/O)*
   General: BADA78, BERN78a, BERN80a, BERN80b, BERN80d, LELA78, LIN79, RAMI79
   Thomas' Write Rule: THOM79
   Multiversion timestamp ordering: MONT78, REED78
   Timestamp and clock management: LAMP78, THOM79
6. *Two-phase locking (2PL)*
   General. BERN79b, BREI79, ESWA76, GARD77, GRAY75, GRAY78, PAPA79, SCHL78, SILB80, STEA81
   Distributed 2PL: MENA80, MINO79, ROSE78, STON79
   Primary copy 2PL: STON77, STON79
   Centralized 2PL: ALSB76a, ALSB76b, GARC79b, GARC79c
   Voting 2PL: GIFF79, SEQU79, THOM79
   Deadlock detection/prevention: GRAY78, KING74, KAWA79, ROSE78, STON79

Received April 1980; final revision accepted February 1981

Experience with Processes and Monitors in Mesa[1]

Butler W. Lampson
Xerox Palo Alto Research Center

David D. Redell
Xerox Business Systems

Abstract

The use of monitors for describing concurrency has been much discussed in the literature. When monitors are used in real systems of any size, however, a number of problems arise which have not been adequately dealt with: the semantics of nested monitor calls; the various ways of defining the meaning of WAIT; priority scheduling; handling of timeouts, aborts and other exceptional conditions; interactions with process creation and destruction; monitoring large numbers of small objects. These problems are addressed by the facilities described here for concurrent programming in Mesa. Experience with several substantial applications gives us some confidence in the validity of our solutions.

Key Words and Phrases: concurrency, condition variable, deadlock, module, monitor, operating system, process, synchronization, task

CR Categories: 4.32, 4.35, 5.24

1. Introduction

In early 1977 we began to design the concurrent programming facilities of Pilot, a new operating system for a personal computer [18]. Pilot is a fairly large program itself (24,000 lines of Mesa code). In addition, it must support a variety of quite large application programs, ranging from database management to inter-network message transmission, which are heavy users of concurrency; our experience with some of these applications is discussed later in the paper. We intended the new facilities to be used at least for the following purposes:

> *Local concurrent programming. An* individual application can be implemented as a tightly coupled group of synchronized processes to express the concurrency inherent in the application.

---

[1] This paper appeared in *Communications of the ACM* **23**, 2 (Feb. 1980), pp 105-117. An earlier version was presented at the 7th ACM Symposium on Operating Systems Principles, Pacific Grove, CA, Dec. 1979. This version was created from the published version by scanning and OCR; it may have errors.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Global resource sharing. Independent applications can run together on the same machine, cooperatively sharing the resources; in particular, their processes can share the processor.

Replacing interrupts. A request for software attention to a device can be handled directly by waking up an appropriate process, without going through a separate interrupt mechanism (for example, a forced branch).

Pilot is closely coupled to the Mesa language [17], which is used to write both Pilot itself and the applications programs it supports. Hence it was natural to design these facilities as part of Mesa; this makes them easier to use, and also allows the compiler to detect many kinds of errors in their use. The idea of integrating such facilities into a language is certainly not new; it goes back at least as far as PL/1 [1]. Furthermore, the invention of monitors by Dijkstra, Hoare, and Brinch Hansen [3, 5, 8] provided a very attractive framework for reliable concurrent programming. There followed a number of papers on the integration of concurrency into programming languages, and at least one implementation [4].

We therefore thought that our task would be an easy one: read the literature, compare the alternatives offered there, and pick the one most suitable for our needs. This expectation proved to be naive. Because of the large size and wide variety of our applications, we had to address a number of issues which were not clearly resolved in the published work on monitors. The most notable among these are listed below, with the sections in which they are discussed.

(a) *Program structure*. Mesa has facilities for organizing programs into modules which communicate through well-defined interfaces. Processes must fit into this scheme (see Section 3.1).

(b) *Creating processes*. A set of processes fixed at compile-time is unacceptable in such a general-purpose system (See Section 2). Existing proposals for varying the amount of concurrency were limited to concurrent elaboration of the statements in a block, in the style of Algol 68 (except for the rather complex mechanism in PL/1).

(c) *Creating monitors*. A fixed number of monitors is also unacceptable, since the number of synchronizers should be a function of the amount of data, but many of the details of existing proposals depended on a fixed association of a monitor with a block of the program text (see Section 3.2).

(d) *WAIT in a nested monitor call*. This issue had been (and has continued to be) the source of a considerable amount of confusion, which we had to resolve in an acceptable manner before we could proceed (see Section 3.1).

(e) *Exceptions*. A realistic system must have timeouts, and it must have a way to abort a process (see Section 4.1). Mesa has an UNWIND mechanism for abandoning part of a sequential computation in an orderly way, and this must interact properly with monitors (see Section 3.3).

(f) *Scheduling*. The precise semantics of waiting on a condition variable had been discussed [10] but not agreed upon, and the reasons for making any particular choice had not been articulated (see Section 4). No attention had been paid to the interaction between monitors and priority scheduling of processes (see Section 4.3).

(g) *Input-Output*. The details of fitting I/O devices into the framework of monitors and condition variables had not been fully worked out (see Section 4.2).

Some of these points have also been made by Keedy [12], who discusses the usefulness of monitors in a modern general-purpose mainframe operating system. The Modula language [21] addresses (b) and (g), but in a more limited context than ours.

Before settling on the monitor scheme described below, we considered other possibilities. We felt that our first task was to choose either shared memory (that is, monitors) or message passing as our basic interprocess communication paradigm.

Message passing has been used (without language support) in a number of operating systems; for a recent proposal to embed messages in a language, see [9]. An analysis of the differences between such schemes and those based on monitors was made by Lauer and Needham [14]. They conclude that, given certain mild restrictions on programming style, the two schemes are duals under the transformation

message ↔ process
process ↔ monitor
send/reply ↔ call/return

Since our work is based on a language whose main tool of program structuring is the procedure, it was considerably easier to use a monitor scheme than to devise a message-passing scheme properly integrated with the type system and control structures of the language.

Within the shared memory paradigm, we considered the possibility of adopting a simpler primitive synchronization facility than monitors. Assuming the absence of multiple processors, the simplest form of mutual exclusion appears to be a non-preemptive scheduler; if processes only yield the processor voluntarily, then mutual exclusion is insured between yield points. In its simplest form, this approach tends to produce very delicate programs, since the insertion of a yield in a random place can introduce a subtle bug in a previously correct program. This danger can be alleviated by the addition of a modest amount of “syntactic sugar” to delineate critical sections within which the processor must not be yielded (for example, pseudo monitors). This sugared form of non-preemptive scheduling can provide extremely efficient solutions to simple problems, but was nonetheless rejected for four reasons:

(1) While we were willing to accept an implementation that would not work on multiple processors, we did not want to embed this restriction in our basic semantics.

(2) A separate preemptive mechanism is needed anyway, since the processor must respond to time-critical events (for example, I/O interrupts) for which voluntary process switching is clearly too sluggish. With preemptive process scheduling, interrupts can be treated as ordinary process wakeups, which reduces the total amount of machinery needed and eliminates the awkward situations that tend to occur at the boundary between two scheduling regimes.

(3) The use of non-preemption as mutual exclusion restricts programming generality within critical sections; in particular, a procedure that happens to yield the processor cannot be called. In large systems where modularity is essential, such restrictions are intolerable.

(4) The Mesa concurrency facilities function in a virtual memory environment. The use of non-preemption as mutual exclusion forbids multiprogramming across page faults, since that would effectively insert preemptions at arbitrary points in the program.

For mutual exclusion with a preemptive scheduler, it is necessary to introduce explicit locks, and machinery that makes requesting processes wait when a lock is unavailable. We considered casting our locks as semaphores, but decided that, compared with monitors, they exert too little structuring discipline on concurrent programs. Semaphores do solve several different problems with a single mechanism (for example, mutual exclusion, producer/consumer) but we found similar economies in our implementation of monitors and condition variables (see Section 5.1).

We have not associated any protection mechanism with processes in Mesa, except what is implicit in the type system of the language. Since the system supports only one user, we feel that the considerable protection offered by the strong typing of the language is sufficient. This fact contributes substantially to the low cost of process operations.

2. Processes

Mesa casts the creation of a new process as a special procedure activation that executes concurrently with its caller. Mesa allows *any* procedure (except an internal procedure of a monitor; see Section 3.1) to be invoked in this way, at the caller's discretion. It is possible to later retrieve the results returned by the procedure. For example, a keyboard input routine might be invoked as a normal procedure by writing:

buffer ← *ReadLine*[*terminal*]

but since *ReadLine* is likely to wait for input, its caller might wish instead to compute concurrently:

p ← FORK *ReadLine*[*terminal*];
... <concurrent computation> ...
buffer ← JOIN *p*;

Here the types are

ReadLine: PROCEDURE [*Device*] RETURNS [*Line*];
p: PROCESS RETURNS [*Line*];

The rendezvous between the return from *ReadLine* that terminates the new process and the join in the old process is provided automatically. *ReadLine* is the *root* procedure of the new process.

This scheme has a number of important properties.

(h) It treats a process as a first class value in the language, which can be assigned to a variable or an array element, passed as a parameter, and in general treated exactly like any other value. A process value is like a pointer value or a procedure value that refers to a nested procedure, in that it can become a dangling reference if the process to which it refers goes away.

(i) The method for passing parameters to a new process and retrieving its results is exactly the same as the corresponding method for procedures, and is subject to the same strict type

checking. Just as PROCEDURE is a generator for a family of types (depending on the argument and result types), so PROCESS is a similar generator, slightly simpler since it depends only on result types.

(j) No special declaration is needed for a procedure that is invoked as a process. Because of the implementation of procedure calls and other global control transfers in Mesa [13], there is no extra execution cost for this generality.

(k) The cost of creating and destroying a process is moderate, and the cost in storage is only twice the minimum cost of a procedure instance. It is therefore feasible to program with a large number of processes, and to vary the number quite rapidly. As Lauer and Needham [14] point out, there are many synchronization problems that have straightforward solutions using monitors only when obtaining a new process is cheap.

Many patterns of process creation are possible. A common one is to create a *detached* process that never returns a result to its creator, but instead functions quite independently. When the root procedure *p* of a detached process returns, the process is destroyed without any fuss. The fact that no one intends to wait for a result from *p* can be expressed by executing:

Detach[*p*]

From the point of view of the caller, this is similar to freeing a dynamic variable—it is generally an error to make any further use of the current value of *p*, since the process, running asynchronously, may complete its work and be destroyed at any time. Of course the design of the program may be such that this cannot happen, and in this case the value of *p* can still be useful as a parameter to the *Abort* operation (see Section 4.1).

This remark illustrates a general point: Processes offer some new opportunities to create dangling references. A process variable itself is a kind of pointer, and must not be used after the process is destroyed. Furthermore, parameters passed by reference to a process are pointers, and if they happen to be local variables of a procedure, that procedure must not return until the process is destroyed. Like most implementation languages, Mesa does not provide any protection against dangling references, whether connected with processes or not.

The ordinary Mesa facility for exception handling uses the ordering established by procedure calls to control the processing of exceptions. Any block may have an attached exception handler. The block containing the statement that causes the exception is given the first chance to handle it, then its enclosing block, and so forth until a procedure body is reached. Then the caller of the procedure is given a chance in the same way. Since the root procedure of a process has no caller, it must be prepared to handle any exceptions that can be generated in the process, including exceptions generated by the procedure itself. If it fails to do so, the resulting error sends control to the debugger, where the identity of the procedure and the exception can easily be determined by a programmer. This is not much comfort, however, when a system is in operational use. The practical consequence is that while any procedure suitable for forking can also be called sequentially, the converse is not generally true.

3. Monitors

When several processes interact by sharing data, care must be taken to properly synchronize access to the data. The idea behind monitors is that a proper vehicle for this interaction is one that unifies

- the synchronization,
- the shared data,
- the body of code which performs the accesses.

The data is *protected* by a *monitor,* and can only be accessed within the body of a *monitor procedure.* There are two kinds of monitor procedures: *entry procedures,* which can be called from outside the monitor, and *internal procedures,* which can only be called from monitor procedures. Processes can only perform operations on the data by calling entry procedures. The monitor ensures that at most one process is executing a monitor procedure at a time; this process is said to be *in* the monitor. If a process is in the monitor, any other process that calls an entry procedure will be delayed. The monitor procedures are written textually next to each other, and next to the declaration of the protected data, so that a reader can conveniently survey all the references to the data.

As long as any order of calling the entry procedures produces meaningful results, no additional synchronization is needed among the processes sharing the monitor. If a random order is not acceptable, other provisions must be made in the program outside the monitor. For example, an unbounded buffer with *Put* and *Get* procedures imposes no constraints (of course a *Get* may have to wait, but this is taken care of within the monitor, as described in the next section). On the other hand, a tape unit with *Reserve, Read, Write,* and *Release* operations requires that each process execute a *Reserve* first and a *Release* last. A second process executing a *Reserve* will be delayed by the monitor, but another process doing a *Read* without a prior *Reserve* will produce chaos. Thus monitors do not solve all the problems of concurrent programming; they are intended, in part, as primitive building blocks for more complex scheduling policies. A discussion of such policies and how to implement them using monitors is beyond the scope of this paper.

3.1 Monitor modules

In Mesa the simplest monitor is an instance of a *module,* which is the basic unit of global program structuring. A Mesa module consists of a collection of procedures and their global data, and in sequential programming is used to implement a data abstraction. Such a module has PUBLIC procedures that constitute the external interface to the abstraction, and PRIVATE procedures that are internal to the implementation and cannot be called from outside the module; its data is normally entirely private. A MONITOR module differs only slightly. It has three kinds of procedures: *entry, internal* (private), and *external* (non-monitor procedures). The first two are the monitor procedures, and execute with the monitor lock held. For example, consider a simple storage allocator with two entry procedures, *Allocate* and *Free,* and an external procedure *Expand* that increases the size of a block.

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

A Mesa module is normally used to package a collection of related procedures and protect their private data from external access. In order to avoid introducing a new lexical structuring mechanism, we chose LO make the scope of a monitor identical to a module. Sometimes, however, procedures that belong in an abstraction do not need access to any shared data, and hence need not be entry procedures of the monitor; these must be distinguished somehow.

For example, two asynchronous processes clearly must not execute in the *Allocate* or *Free* procedures at the same time; hence, these must be entry procedures. On the other hand, it is unnecessary to hold the monitor lock during the copy in *Expand*, even though this procedure logically belongs in the storage allocator module; it is thus written as an external procedure. A more complex monitor might also have internal procedures, which are used to structure its computations, but which are inaccessible from outside the monitor. These do not acquire and release the lock on call and return, since they can only be called when the lock is already held.

If no suitable block is available, *Allocate* makes its caller wait on the *condition* variable *moreAvailable. Free* does a NOTIFY to this variable whenever a new block becomes available; this causes some process waiting on the variable to resume execution (see Section 4 for details). The WAIT releases the monitor lock, which is reacquired when the waiting process reenters the monitor. If a WAIT is done in an internal procedure, it still releases the lock. If, however, the monitor calls some other procedure which is outside the monitor module, the lock is not released, even if the other procedure is in (or calls) another monitor and ends up doing a WAIT. The same rule is adopted in Concurrent Pascal [4].

To understand the reasons for this, consider the form of a correctness argument for a program using a monitor. The basic idea is that the monitor maintains an *invariant* that is always true of its data, except when some process is executing in the monitor. Whenever control leaves the monitor, this invariant must be established. In return, whenever control enters the monitor the invariant can be assumed. Thus an entry procedure must establish the invariant before returning, and a monitor procedure must establish it before doing a WAIT. The invariant can be assumed at

the start of an entry procedure, and after each WAIT. Under these conditions, the monitor lock ensures that no one can enter the monitor when the invariant is false. Now, if the lock were to be released on a WAIT done in another monitor which happens to be called from this one, the invariant would have to be established before making the call which leads to the WAIT. Since in general there is no way to know whether a call outside the monitor will lead to a WAIT, the invariant would have to be established before every such call. The result would be to make calling such procedures hopelessly cumbersome.

An alternative solution is to allow an *outside block* to be written inside a monitor, with the following meaning: on entry to the block the lock is released (and hence the invariant must be established); within the block the protected data is inaccessible; on leaving the block the lock is reacquired. This scheme allows the state represented by the execution environment of the monitor to be maintained during the outside call, and imposes a minimal burden on the programmer: to establish the invariant before making the call. This mechanism would be easy to add to Mesa; we have left it out because we have not seen convincing examples in which it significantly simplifies the program.

If an entry procedure generates an exception in the usual way, the result will be a call on the exception handler from within the monitor, so that the lock will not be released. In particular, this means that the exception handler must carefully avoid invoking that same monitor, or a deadlock will result. To avoid this restriction, the entry procedure can restore the invariant and then execute

RETURN WITH ERROR[(arguments)]

which returns from the entry procedure, thus releasing the lock, and then generates the exception.

3.2 Monitors and deadlock

There are three patterns of pairwise deadlock that can occur using monitors. In practice, of course, deadlocks often involve more than two processes, in which case the actual patterns observed tend to be more complicated; conversely, it is also possible for a single process to deadlock with itself (for example, if an entry procedure is recursive).

The simplest form of deadlock takes place inside a single monitor when two processes do a WAIT, each expecting to be awakened by the other. This represents a localized bug in the monitor code and is usually easy to locate and correct.

A more subtle form of deadlock can occur if there is a cyclic calling pattern between two monitors. Thus if monitor *M* calls an entry procedure in *N*, and *N* calls one in M, each will wait for the other to release the monitor lock. This kind of deadlock is made neither more nor less serious by the monitor mechanism. It arises whenever such cyclic dependencies are allowed to occur in a program, and can be avoided in a number of ways. The simplest is to impose a partial ordering on resources such that all the resources simultaneously possessed by any process are totally ordered, and insist that if resource *r* precedes 5 in the ordering, then *r* cannot be acquired later than *5*. When the resources are monitors, this reduces to the simple rule that mutually recursive monitors must be avoided. Concurrent Pascal [4] makes this check at compile time; Mesa cannot do so because it has procedure variables.

A more serious problem arises if *M* calls *N*, and *N* then waits for a condition which can only occur when another process enters *N* through *M* and makes the condition true. In this situation, *N* will be unlocked, since the WAIT occurred there, but *M* will remain locked during the WAIT in *N*. This kind of two level data abstraction must be handled with some care. A straightforward solution using standard monitors is to break *M* into two parts: a monitor *M'* and an ordinary module *0* which implements the abstraction defined by *M*, and calls *M'* for access to the shared data. The call on *N* must be done from *0* rather than from within *M'*.

Monitors, like any other interprocess communication mechanism, are a *tool* for implementing synchronization constraints chosen by the programmer. It is unreasonable to blame the tool when poorly chosen constraints lead to deadlock. What is crucial, however, is that the tool make the program structure as understandable as possible, while not restricting the programmer too much in his choice of constraints (for example, by forcing a monitor lock to be held much longer than necessary). To some extent, these two goals tend to conflict; the Mesa concurrency facilities attempt to strike a reasonable balance and provide an environment in which the conscientious programmer can avoid deadlock reasonably easily. Our experience in this area is reported in Section 6.

3.3 Monitored objects

Often we wish to have a collection of shared data objects, each one representing an instance of some abstract object such as a file, a storage volume, a virtual circuit, or a database view, and we wish to add objects to the collection and delete them dynamically. In a sequential program this is done with standard techniques for allocating and freeing storage. In a concurrent program, however, provision must also be made for serializing access to each object. The straightforward way is to use a single monitor for accessing all instances of the object, and we recommend this approach whenever possible. If the objects function independently of each other for the most part, however, the single monitor drastically reduces the maximum concurrency that can be obtained. In this case, what we want is to give each object its own monitor; all these monitors will share the same code, since all the instances of the abstract object share the same code, but each object will have its own lock.

One way to achieve this result is to make multiple instances of the monitor module. Mesa makes this quite easy, and it is the next recommended approach. However, the data associated with a module instance includes information that the Mesa system uses to support program linking and code swapping, and there is some cost in duplicating this information. Furthermore, module instances are allocated by the system; hence the program cannot exercise the fme control over allocation strategies which is possible for ordinary Mesa data objects. We have therefore introduced a new type constructor called a *monitored record*, which is exactly like an ordinary record, except that it includes a monitor lock and is intended to be used as the protected data of a monitor.

In writing the code for such a monitor, the programmer must specify how to access the monitored record, which might be embedded in some larger data structure passed as a parameter to the entry procedures. This is done with a LOCKS clause which is written at the beginning of the module:

MONITOR LOCKS *file*↑ USING *file*: POINTER TO *FileData*;

if the *FileData* is the protected data. An arbitrary expression can appear in the LOCKS clause; for instance, LOCKS *file.buffers*[*currentPage*] might be appropriate if the protected data is one of the buffers in an array which is part of the *file*. Every entry procedure of this monitor, and every internal procedure that does a WAIT, must have access to a *file*, so that it can acquire and release the lock upon entry or around a WAIT. This can be accomplished in two ways: the *file* may be a global variable of the module, or it may be a parameter to *every* such procedure. In the latter case, we have effectively created a separate monitor for each object, without limiting the program's freedom to arrange access paths and storage allocation as it likes.

Unfortunately, the type system of Mesa is not strong enough to make this construction completely safe. If the value of *file* is changed within an entry procedure, for example, chaos will result, since the return from this procedure will release not the lock which was acquired during the call, but some other lock instead. In this example we can insist that *file* be read-only, but with another level of indirection aliasing can occur and such a restriction cannot be enforced. In practice this lack of safety has not been a problem.

3.4 Abandoning a computation

Suppose that a procedure P_1 has called another procedure P_2, which in turn has called P_3 and so forth until the current procedure is P_n. If P_n generates an exception which is eventually handled by P_1 (because P_2 ... P_n do not provide handlers), Mesa allows the exception handler in P_1 to abandon the portion of the computation being done in P_2 ... P_n and continue execution in P_1. When this happens, a distinguished exception called UNWIND is first generated, and each of P_2 ... P_n is given a chance to handle it and do any necessary cleanup before its activation is destroyed.

This feature of Mesa is not part of the concurrency facilities, but it does interact with those facilities in the following way. If one of the procedures being abandoned, say P_i, is an entry procedure, then the invariant must be restored and the monitor lock released before P_i is destroyed. Thus if the logic of the program allows an UNWIND, the programmer must supply a suitable handler in P_i to restore the invariant; Mesa will automatically supply the code to release the lock. If the programmer fails to supply an UNWIND handler for an entry procedure, the lock is *not* automatically released, but remains set; the cause of the resulting deadlock is not hard to find.

4. Condition variables

In this section we discuss the precise semantics of WAIT and other details associated with condition variables. Hoare's definition of monitors [8] requires that a process waiting on a condition variable must run immediately when another process *signals* that variable, and that the signaling process in turn runs as soon as the waiter leaves the monitor. This definition allows the waiter to assume the truth of some predicate stronger than the monitor invariant (which the signaler must of course establish), but it requires several additional process switches whenever a process continues after a WAIT. It also requires that the signaling mechanism be perfectly reliable.

Mesa takes a different view: When one process establishes a condition for which some other process may be waiting, it *notifies* the corresponding condition variable. A NOTIFY is regarded as a *hint* to a waiting process; it causes execution of some process waiting on the condition to resume at some convenient future time. When the waiting process resumes, it will reacquire the

monitor lock. There is no guarantee that some other process will not enter the monitor before the waiting process. Hence nothing more than the monitor invariant may be assumed after a WAIT, and the waiter must reevaluate the situation each time it resumes. The proper pattern of code for waiting is therefore:

WHILE NOT <OK to proceed> DO WAIT *c* ENDLOOP.

This arrangement results in an extra evaluation of the <OK to proceed> predicate after a wait, compared to Hoare's monitors, in which the code is:

IF NOT <OK to proceed> THEN WAIT *c*.

In return, however, there are no extra process switches, and indeed no constraints at all on when the waiting process must run after a NOTIFY. In fact, it is perfectly all right to run the waiting process even if there is no NOTIFY, although this is presumably pointless if a NOTIFY is done whenever an interesting change is made to the protected data.

It is possible that such a laissez-faire attitude to scheduling monitor accesses will lead to unfairness and even starvation. We do not think this is a legitimate cause for concern, since in a properly designed system there should typically be no processes waiting for a monitor lock. As Hoare, Brinch Hansen, Keedy, and others have pointed out, the low level scheduling mechanism provided by monitor locks should not be used to implement high level scheduling decisions within a system (for example, about which process should get a printer next). High level scheduling should be done by taking account of the specific characteristics of the resource being scheduled (for example, whether the right kind of paper is in the printer). Such a scheduler will delay its client processes on condition variables after recording information about their requirements, make its decisions based on this information, and notify the proper conditions. In such a design the data protected by a monitor is never a bottleneck.

The verification rules for Mesa monitors are thus extremely simple: The monitor invariant must be established just before a return from an entry procedure or a WAIT, and it may be assumed at the start of an entry procedure and just after a WAIT. Since awakened waiters do not run immediately, the predicate established before a NOTIFY cannot be assumed after the corresponding WAIT, but since the waiter tests explicitly for <OK to proceed>, verification is actually made simpler and more localized.

Another consequence of Mesa's treatment of NOTIFY as a hint is that many applications do not trouble to determine whether the exact condition needed by a waiter has been established. Instead, they choose a very cheap predicate which implies the exact condition (for example, some change has occurred), and NOTIFY a *covering* condition variable. Any waiting process is then responsible for determining whether the exact condition holds; if not, it simply waits again. For example, a process may need to wait until a particular object in a set changes state. A single condition covers the entire set, and a process changing any of the objects broadcasts to this condition (see Section 4.1). The information about exactly which objects are currently of interest is implicit in the states of the waiting processes, rather than having to be represented explicitly in a shared data structure. This is an attractive way to decouple the detailed design of two processes: it is feasible because the cost of waking up a process is small.

4.1 Alternatives to NOTIFY

With this rule it is easy to add three additional ways to resume a waiting process:

Timeout. Associated with a condition variable is a timeout interval *t.* A process which has been waiting for time *t* will resume regardless of whether the condition has been notified. Presumably in most cases it will check the time and take some recovery action before waiting again. The original design for timeouts raised an exception if the timeout occurred; it was changed because many users simply wanted to retry on a timeout, and objected to the cost and coding complexity of handling the exception. This decision could certainly go either way.

Abort. A process may be aborted at any time by executing *Abort*[*p*]. The effect is that the next time the process waits, or if it is waiting now, it will resume immediately and the *Aborted* exception will occur. This mechanism allows one process to gently prod another, generally to suggest that it should clean up and terminate. The aborted process is, however, free to do arbitrary computations, or indeed to ignore the abort entirely.

Broadcast. Instead of doing a NOTIFY to a condition, a process may do a BROADCAST, which causes *all* the processes waiting on the condition to resume, instead of simply one of them. Since a NOTIFY is just a hint, it is always correct to use BROADCAST. It is better to use NOTIFY if there will typically be several processes waiting on the condition, and it is known that any waiting process can respond properly. On the other hand, there are times when a BROADCAST is correct and a NOTIFY is not; the alert reader may have noticed a problem with the example program in Section 3.1, which can be solved by replacing the NOTIFY with a BROADCAST.

None of these mechanisms affects the proof rule for monitors at all. Each provides a way to attract the attention of a waiting process at an appropriate time.

Note that there is no way to stop a runaway process. This reflects the fact that Mesa processes are cooperative. Many aspects of the design would not be appropriate in a competitive environment such as a general-purpose timesharing system.

4.2 Naked NOTIFY

Communication with input/output devices is handled by monitors and condition variables much like communication among processes. There is typically a shared data structure, whose details are determined by the hardware, for passing commands to the device and returning status information. Since it is not possible for the device to wait on a monitor lock, the update operations on this structure must be designed so that the single word atomic read and write operations provided by the memory are sufficient to make them atomic. When the device needs attention, it can NOTIFY a condition variable to wake up a waiting process (that is, the interrupt handler); since the device does not actually acquire the monitor lock, its NOTIFY is called a *naked* NOTIFY. The device finds the address of the condition variable in a ftxed memory location.

There is one complication associated with a naked NOTIFY: Since the notification is not protected by a monitor lock, there can be a race. It is possible for a process to be in the monitor, find the <OK to proceed> predicate to be FALSE (that is, the device does not need attention), and be about to do a WAIT, when the device updates the shared data and does its NOTIFY. The WAIT will then be done and the NOTIFY from the device will be lost. With ordinary processes, this cannot happen,

since the monitor lock ensures that one process cannot be testing the predicate and preparing to WAIT, while another is changing the value of <OK to proceed> and doing the NOTIFY. The problem is avoided by providing the familiar wakeup-waiting switch [19] in a condition variable, thus turning it into a binary semaphore [8]. This switch is needed only for condition variables that are notified by devices.

We briefly considered a design in which devices would wait on and acquire the monitor lock, exactly like ordinary Mesa processes; this design is attractive because it avoids both the anomalies just discussed. However, there is a serious problem with any kind of mutual exclusion between two processes which run on processors of substantially different speeds: The faster process may have to wait for the slower one. The worst-case response time of the faster process therefore cannot be less than the time the slower one needs to finish its critical section. Although one can get higher throughput from the faster processor than from the slower one, one cannot get better worst-case real time performance. We consider this a fundamental deficiency.

It therefore seemed best to avoid any mutual exclusion (except for that provided by the atomic memory read and write operations) between Mesa code and device hardware and microcode. Their relationship is easily cast into a producer-consumer form, and this can be implemented, using linked lists or arrays, with only the memory's mutual exclusion. Only a small amount of Mesa code must handle device data structures without the protection of a monitor. Clearly a change of models must occur at some point between a disk head and an application program; we see no good reason why it should not happen within Mesa code, although it should certainly be tightly encapsulated.

4. Priorities

In some applications it is desirable to use a priority scheduling discipline for allocating the processor(s) to processes which are not waiting. Unless care is taken, the ordering implied by the assignment of priorities can be subverted by monitors. Suppose there are three priority levels (3 highest, 1 lowest), and three processes P_1, P_2, and P_3, one running at each level. Let P_1 and P_3 communicate using a monitor *M*. Now consider the following sequence of events:

1. P_1 enters *M*.
2. P_1 is preempted by P_2.
3. P_2 is preempted by P_3.
4. P_3 tries to enter the monitor, and waits for the lock.
5. P_2 runs again, and can effectively prevent P_3 from running, contrary to the purpose of the priorities.

A simple way to avoid this situation is to associate with each monitor the priority of the highest priority process which ever enters that monitor. Then whenever a process enters a monitor, its priority is temporarily increased to the monitor's priority. Modula solves the problem in an even simpler way—interrupts are disabled on entry to *M*, thus effectively giving the process the highest possible priority, as well as supplying the monitor lock for *M*. This approach fails if a page fault can occur while executing in *M*.

The mechanism is not free, and whether or not it is needed depends on the application. For instance, if only processes with adjacent priorities share a monitor, the problem described above

cannot occur. Even if this is not the case, the problem may occur rarely, and absolute enforcement of the priority scheduling may not be important.

5. Implementation

The implementation of processes and monitors is split more or less equally among the Mesa compiler, the runtime package, and the underlying machine. The compiler recognizes the various syntactic constructs and generates appropriate code, including implicit calls on built-in (that is, known to the compiler) support procedures. The runtime implements the less heavily used operations, such as process creation and destruction. The machine directly implements the more heavily used features, such as process scheduling and monitor entry/exit.

Note that it was primarily frequency of use, rather than cleanliness of abstraction, that motivated our division of labor between processor and software. Nonetheless, the split did turn out to be a fairly clean layering, in which the birth and death of processes are implemented on top of monitors and process scheduling.

5.1 The processor

The existence of a process is normally represented only by its stack of procedure activation records or *frames*, plus a small (10-byte) description called a *ProcessState.* Frames are allocated from a *frame heap* by a microcoded allocator. They come in a range of sizes that differ by 20 percent to 30 percent; there is a separate free list for each size up to a few hundred bytes (about 15 sizes). Allocating and freeing frames are thus very fast, except when more frames of a given size are needed. Because all frames come from the heap, there is no need to preplan the stack space needed by a process. When a frame of a given size is needed but not available, there is a *frame fault*, and the fault handler allocates more frames in virtual memory. Resident procedures have a private frame heap that is replenished by seizing real memory from the virtual memory manager.

The *ProcessState*s are kept in a fixed table known to the processor; the size of this table determines the maximum number of processes. At any given time, a *ProcessState* is on exactly one *queue.* There are four kinds of queues:

Ready queue. There is one ready queue, containing all processes that are ready to run.

Monitor lock queue. When a process attempts to enter a locked monitor, it is moved from the ready queue to a queue associated with the monitor lock.

Condition variable queue. When a process executes a WAIT, it is moved from the ready queue to a queue associated with the condition variable.

Fault queue. A fault can make a process temporarily unable to run; such a process is moved from the ready queue to a fault queue, and a fault handling process is notified.

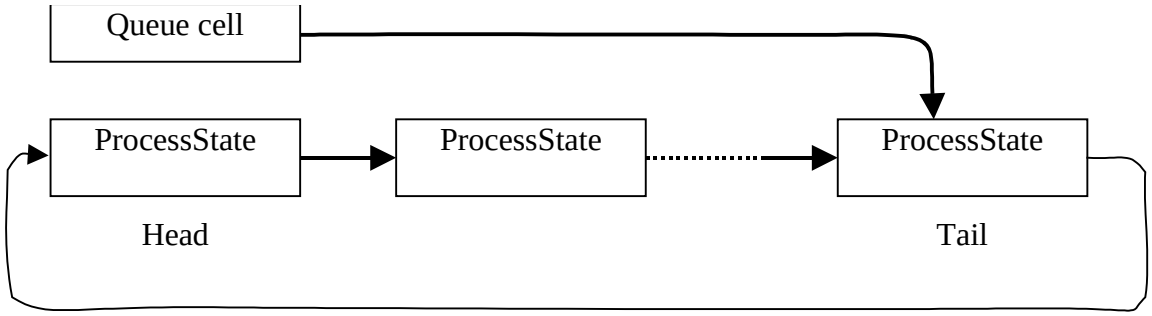


Figure 1: A process queue

Queues are kept sorted by process priority. The implementation of queues is a simple one way circular list, with the queue cell pointing to the *tail* of the queue (see Figure 1). This compact structure allows rapid access to both the head and the tail of the queue. Insertion at the tail and removal at the head are quick and easy; more general insertion and deletion involve scanning some fraction of the queue. The queues are usually short enough that this is not a problem. Only the ready queue grows to a substantial size during normal operation, and its patterns of insertions and deletions are such that queue scanning overhead is small.

The queue cell of the ready queue is kept in a fixed location known to the processor, whose fundamental task is to always execute the next instruction of the highest priority ready process. To this end, a check is made before each instruction, and a process switch is done if necessary. In particular, this is the mechanism by which interrupts are serviced. The machine thus implements a simple priority scheduler, which is preemptive between priorities and FIFO within a given priority.

Queues other than the ready list are passed to the processor by software as operands of instructions, or through a trap vector in the case of fault queues. The queue cells are passed by reference, since in general they must be updated (that is, the identity of the tail may change.) Monitor locks and condition variables are implemented as small records containing their associated queue cells plus a small amount of extra information: in a monitor lock, the actual lock; in a condition variable, the timeout interval and the wakeup-waiting switch.

At a fixed interval (about 20 times per second) the processor scans the table of *ProcessState*s and notifies any waiting processes whose timeout intervals have expired. This special NOTIFY is tricky because the processor does not know the location of the condition variables on which such processes are waiting, and hence cannot update the queue cells. This problem is solved by leaving the queue cells out of date, but marking the processes in such a way that the next normal usage of the queue cells will notice the situation and update them appropriately.

There is no provision for time-slicing in the current implementation, but it could easily be added, since it has no effect on the semantics of processes.

5.2 The runtime support package

The *Process* module of the Mesa runtime package does creation and deletion of processes. This module is written (in Mesa) as a monitor, using the underlying synchronization machinery of the processor to coordinate the implementation of FORK and JOIN as the built-in entry procedures *Process.Fork* and *Process.Join*, respectively. The unused *ProcessState*s are treated as essentially normal processes which are all waiting on a condition variable called *rebirth*. A call of *Process.Fork* performs appropriate "brain surgery" on the first process in the queue and then notifies *rebirth* to bring the process to life: *Process.Join* synchronizes with the dying process and retrieves the results. The (implicitly invoked) procedure *Process.End* synchronizes the dying process with the joining process and then commits suicide by waiting on *rebirth.* An explicit call on *Process.Detach* marks the process so that when it later calls *Process.End,* it will simply destroy itself immediately.

The operations *Process.Abort* and *Process.Yield* are provided to allow special handling of processes that wait too long and compute too long, respectively. Both adjust the states of the appropriate queues, using the machine's standard queueing mechanisms. Utility routines are also provided by the runtime for such operations as setting a condition variable timeout and setting a process priority.

5.3 The compiler

The compiler recognizes the syntactic constructs for processes and monitors and emits the appropriate code (for example, a MONITORENTRY instruction at the start of each entry procedure, an implicit call of *Process.Fork* for each FORK). The compiler also performs special static checks to help avoid certain frequently encountered errors. For example, use of WAIT in an external procedure is flagged as an error, as is a direct call from an external procedure to an internal one. Because of the power of the underlying Mesa control structure primitives, and the care with which concurrency was integrated into the language, the introduction of processes and monitors into Mesa resulted in remarkably little upheaval inside the compiler.

5.4 Performance

Mesa's concurrent programming facilities allow the intrinsic parallelism of application programs to be represented naturally; the hope is that well structured programs with high global efficiency will result. At the same time, these facilities have nontrivial local costs in storage and/or execution time when compared with similar sequential constructs; it is important to minimize these costs, so that the facilities can be applied to a finer grain of concurrency. This section summarizes the costs of processes and monitors relative to other basic Mesa constructs, such as simple statements, procedures, and modules. Of course, the relative efficiency of an arbitrary concurrent program and an equivalent sequential one cannot be determined from these numbers alone; the intent is simply to provide an indication of the relative costs of various local constructs.

Storage costs fall naturally into data and program storage (both of which reside in swappable virtual memory unless otherwise indicated). The minimum cost for the existence of a Mesa module is 8 bytes of data and 2 bytes of code. Changing the module to a monitor adds 2 bytes of data and 2 bytes of code. The prime component of a module is a set of procedures, each of which

requires a minimum of an 8-byte activation record and 2 bytes of code. Changing a normal procedure to a monitor entry procedure leaves the size of the activation record unchanged, and adds 8 bytes of code. All of these costs are small compared with the program and data storage actually needed by typical modules and procedures. The other cost specific to monitors is space for condition variables; each condition variable occupies 4 bytes of data storage, while WAIT and NOTIFY require 12 bytes and 3 bytes of code, respectively.

The data storage overhead for a process is 10 bytes of resident storage for its *ProcessState*, plus the swappable storage for its stack of procedure activation records. The process itself contains no extra code, but the code for the FORK and JOIN which create and delete it together occupy 13 bytes, as compared with 3 bytes for a normal procedure call and return. The FORK/JOIN sequence also uses 2 data bytes to store the process value. In summary:

	Space (*bytes*)	
Construct	*data*	*code*
module	8	2
procedure	8	2
call + return	-	3
monitor	10	4
entry procedure	8	10
FORK+JOIN	2	13
process	10	0
condition variable	4	
WAIT	-	12
NOTIFY	-	3

For measuring execution times we define a unit called a *tick*: the time required to execute a simple instruction (for example, on a "one MIP" machine, one tick would be one microsecond). A tick is arbitrarily set at one-fourth of the time needed to execute the simple statement "$a \leftarrow b + c$" (that is, two loads, an add, and a store). One interesting number against which to compare the concurrency facilities is the cost of a normal procedure call (and its associated return), which takes 30 ticks if there are no arguments or results.

The cost of calling and returning from a monitor entry procedure is 50 ticks, about 70 percent more than an ordinary call and return. In practice, the percentage increase is somewhat lower, since typical procedures pass arguments and return results, at a cost of 24 ticks per item. A process switch takes 60 ticks; this includes the queue manipulations and all the state saving and restoring. The speed of WAIT and NOTIFY depends somewhat on the number and priorities of the processes involved, but representative figures are 15 ticks for a WAIT and 6 ticks for a NOTIFY. Finally, the minimum cost of a FORK/ JOIN pair is 1,100 ticks, or about 38 times that of a procedure call. To summarize:

Construct	*Time* (*ticks*)
simple instruction	1
call + return	30
monitor call + return	50
process switch	60
WAIT	15
NOTIFY, no one waiting	4
NOTIFY, process waiting	9
FORK+JOIN	1,100

On the basis of these performance figures, we feel that our implementation has met our efficiency goals, with the possible exception of FORK and JOIN. The decision to implement these two language constructs in software rather than in the underlying machine is the main reason for their somewhat lackluster performance. Nevertheless, we still regard this decision as a sound one, since these two facilities are considerably more complex than the basic synchronization mechanism, and are used much less frequently (especially JOIN, since the detached processes discussed in Section 2 have turned out to be quite popular).

6. Applications

In this section we describe the way in which processes and monitors are used by three substantial Mesa programs: an operating system, a calendar system using replicated databases, and an internetwork gateway.

6.1 Pilot: A general-purpose operating system

Pilot is a Mesa-based operating system [18] which runs on a large personal computer. It was designed jointly with the new language features and makes heavy use of them. Pilot has several autonomous processes of its own, and can be called by any number of client processes of any priority, in a fully asynchronous manner. Exploiting this potential concurrency requires extensive use of monitors within Pilot; the roughly 75 program modules contain nearly 40 separate monitors.

The Pilot implementation includes about 15 dedicated processes (the exact number depends on the hardware configuration); most of these are event handlers for three classes of events:

I/O interrupts. Naked notifies as discussed in Section 4.2.

Process faults. Page faults and other such events, signaled via fault queues as discussed in Section 5.1. Both client code and the higher levels of Pilot, including some of the dedicated processes, can cause such faults.

Internal exceptions. Missing entries in resident databases, for example, cause an appropriate high level “helper” process to wake up and retrieve the needed data from secondary storage.

There are also a few “daemon” processes, which awaken periodically and perform housekeeping chores (for example, swap out unreferenced pages). Essentially all of Pilot’s internal processes

and monitors are created at system initialization time (in particular, a suitable complement of interrupt handler processes is created to match the actual hardware configuration, which is determined by interrogating the hardware). The running system makes no use of dynamic process and monitor creation, largely because much of Pilot is involved in implementing facilities such as virtual memory which are themselves used by the dynamic creation software.

The internal structure of Pilot is fairly complicated, but careful placement of monitors and dedicated processes succeeded in limiting the number of bugs which caused deadlock; over the life of the system, somewhere between one and two dozen distinct deadlocks have been discovered, all of which have been fixed relatively easily without any global disruption of the system's structure.

At least two areas have caused annoying problems in the development of Pilot:

1. *The lack of mutual exclusion in the handling of interrupts*. As in more conventional interrupt systems, subtle bugs have occurred due to timing races between I/O devices and their handlers. To some extent, the illusion of mutual exclusion provided by the casting of interrupt code as a monitor may have contributed to this, although we feel that the resultant economy of mechanism still justifies this choice.

2. *The interaction of the concurrency and exception facilities*. Aside from the general problems of exception handling in a concurrent environment, we have experienced some difficulties due to the specific interactions of Mesa signals with processes and monitors (see Sections 3.1 and 3.4). In particular, the reasonable and consistent handling of signals (including UNWINDS) in entry procedures represents a considerable increase in the mental overhead involved in designing a new monitor or understanding an existing one.

6.2 Violet: A distributed calendar system

The Violet system [6, 7] is a distributed database manager which supports replicated data files, and provides a display interface to a distributed calendar system. It is constructed according to the hierarchy of abstractions shown in Figure 2. Each level builds on the next lower one by calling procedures supplied by it. In addition, two of the levels explicitly deal with more than one process. Of course, as any level with multiple processes calls lower levels, it is possible for multiple processes to be executing procedures in those levels as well.

The user interface level has three processes: *Display*, *Keyboard*, and *DataChanges*. The *Display* process is responsible for keeping the display of the database consistent with the views specified by the user and with changes occurring in the database itself. The other processes notify it when changes occur, and it calls on lower levels to read information for updating the display. *Display* never calls update operations in any lower level. The other two processes respond to changes initiated either by the user (*Keyboard*) or by the database (*DataChanges*). The latter process is FORKed from the *Transactions* module when data being looked at by Violet changes, and disappears when it has reported the changes to *Display*.

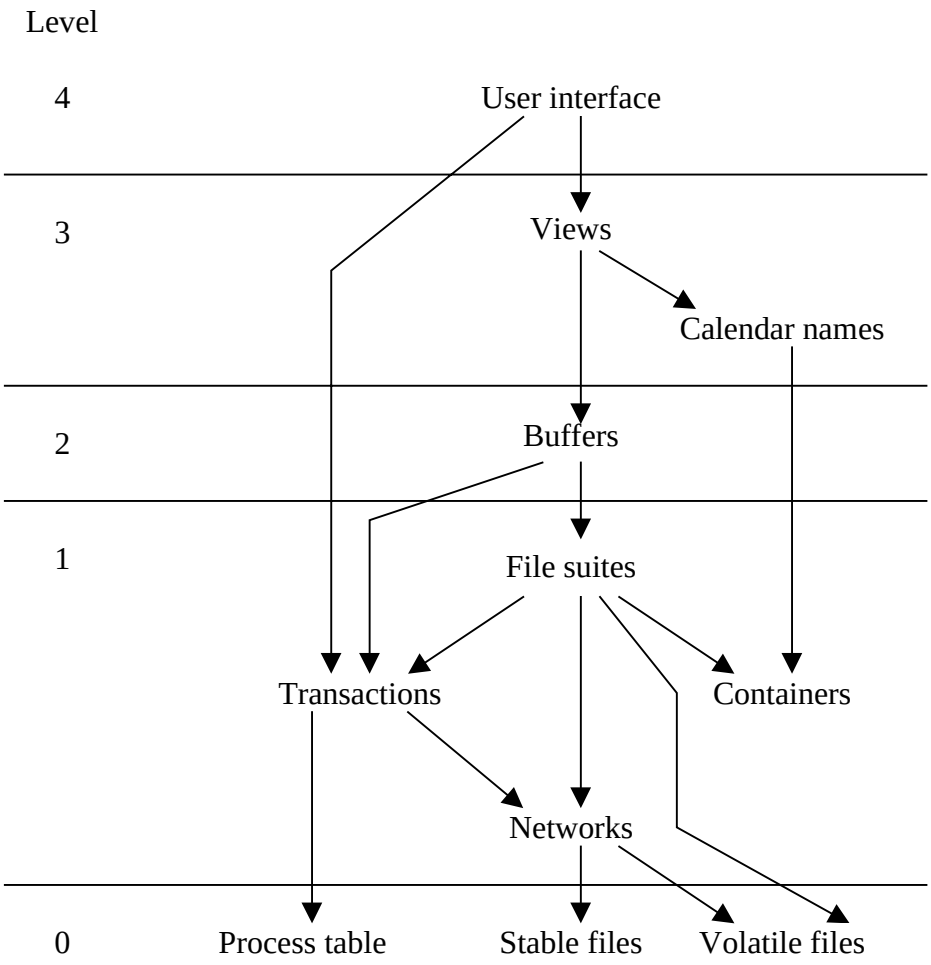


Figure 2: The internal structure of Violet

A more complex constellation of processes exists in *FileSuites*, which constructs a single *replicated file* from a set of *representative* files, each containing data from some version of the replicated file. The representatives are stored in a transactional file system [11], so that each one is updated atomically, and each carries a version number. For each *FileSuite* being accessed, there is a monitor that keeps track of the known representatives and their version numbers. The replicated file is considered to be updated when all the representatives in a *write quorum* have been updated; the latest version can be found by examining a *read quorum*. Provided the sum of the read quorum and the write quorum is as large as the total set of representatives, the replicated file behaves like a conventional file.

When the file suite is created, it FORKs and detaches an *inquiry* process for each representative. This process tries to read the representative's version number, and if successful, reports the number to the monitor associated with the file suite and notifies the condition *CrowdLarger*. Any process trying to read from the suite must collect a read quorum. If there are not enough representatives present yet, it waits on *CrowdLarger*. The inquiry processes expire after their work is done.

When the client wants to update the *FileSuite*, it must collect a write quorum of representatives containing the current version, again waiting on *CrowdLarger* if one is not yet present. It then FORKS an *update* process for each representative in the quorum, and each tries to write its file. After FORKing the update processes, the client JOINS each one in turn, and hence does not proceed until all have completed. Because all processes run within the same transaction, the underlying transactional file system guarantees that either all the representatives in the quorum will be written, or none of them.

It is possible that a write quorum is not currently accessible, but a read quorum is. In this case the writing client FORKS a *copy* process for each representative which is accessible but is not up to date. This process copies the current file suite contents (obtained from the read quorum) into the representative, which is now eligible to join the write quorum.

Thus as many as three processes may be created for each representative in each replicated file. In the normal situation when the state of enough representatives is known, however, all these processes have done their work and vanished; only one monitor call is required to collect a quorum. This potentially complex structure is held together by a single monitor containing an array of representative states and a single condition variable.

6.3 Gateway: An internetwork forwarder

Another substantial application program that has been implemented in Mesa using the process and monitor facilities is an internetwork gateway for packet networks [2]. The gateway is attached to two or more networks and serves as the connection point between them, passing packets across network boundaries as required. To perform this task efficiently requires rather heavy use of concurrency.

At the lowest level, the gateway contains a set of device drivers, one per device, typically consisting of a high priority interrupt process, and a monitor for synchronizing with the device and with non-interrupt-level software. Aside from the drivers for standard devices (disk, keyboard, etc.) a gateway contains two or more drivers for Ethernet local broadcast networks [16] and/or common carrier lines. Each Ethernet driver has two processes, an interrupt process and a background process for autonomous handling of timeouts and other infrequent events. The driver for common carrier lines is similar, but has a third process which makes a collection of lines resemble a single Ethernet by iteratively simulating a broadcast. The other network drivers have much the same structure; all drivers provide the same standard network interface to higher level software.

The next level of software provides packet routing and dispatching functions. The *dispatcher* consists of a monitor and a dedicated process. The monitor synchronizes interactions between the drivers and the dispatcher process. The dispatcher process is normally waiting for the completion of a packet transfer (input or output); when one occurs, the interrupt process handles the interrupt, notifies the dispatcher, and immediately returns to await the next interrupt. For example, on input the interrupt process notifies the dispatcher, which dispatches the newly arrived packet to the appropriate *socket* for further processing by invoking a procedure associated with the socket.

The *router* contains a monitor that keeps a *routing* table mapping network names to addresses of other gateway machines. This defines the next “hop” in the path to each accessible remote

network. The router also contains a dedicated housekeeping process that maintains the table by exchanging special packets with other gateways. A packet is transmitted rather differently than it is received. The process wishing to transmit to a remote socket calls into the router monitor to consult the routing table, and then the same process calls directly into the appropriate network driver monitor to initiate the output operation. Such asymmetry between input and output is particularly characteristic of packet communication, but is also typical of much other I/O software.

The primary operation of the gateway is now easy to describe: When the arrival of a packet has been processed up through the level of the dispatcher, and it is discovered that the packet is addressed to a remote socket, the dispatcher forwards it by doing a normal transmission; that is, consulting the routing table and calling back down to the driver to initiate output. Thus, although the gateway contains a substantial number of asynchronous processes, the most critical path (forwarding a message) involves only a single switch between a pair of processes.

Conclusion

The integration of processes and monitors into the Mesa language was a somewhat more substantial task than one might have anticipated, given the flexibility of Mesa's control structures and the amount of published work on monitors. This was largely because Mesa is designed for the construction of large, serious programs, and processes and monitors had to be refined sufficiently to fit into this context. The task has been accomplished, however, yielding a set of language features of sufficient power that they serve as the only software concurrency mechanism on our personal computer, handling situations ranging from input/output interrupts to cooperative resource sharing among unrelated application programs.

Received June 1979; accepted September 1979: revised November 1979

References

1. *American National Standard Programming Language PL/1*. X3.53, American Nat. Standards Inst., New York, 1976.

2. Boggs, D.R. et al. Pup: An internetwork architecture. *IEEE Trans. on Communications* **28**, 4 (April 1980).

3. Brinch Hansen, P. *Operating System Principles*. Prentice-Hall, July 1973.

4. Brinch Hansen. P. The programming language Concurrent Pascal. *IEEE Trans. on Software Engineering* **1**,2 (June 1975), 199-207.

5. Dijkstra, E.W. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*, Academic Press, 1972.

6. Gifford, D.K. Weighted voting for replicated data. *Operating Systems Review* **13**, 5 (Dec.1979), l50-l62.

7. Gifford. D.K. Violet, an experimental decentralized system. Integrated Office Systems Workshop, IRIA, Rocquencourt, France, Nov. 1979 (also available as CSL report 79-12, Xerox Research Center, Palo Alto, Calif.).

8. Hoare, C.A.R. Monitors: An operating system structuring concept. *Comm. ACM* **17**, 10 (Oct.1974), 549-557.

9. Hoare, C.A.R. Communicating sequential processes. *Comm. ACM* **21**, 8 (Aug.1978), 666-677.

10. Howard, J.H. Signaling in monitors. *Second Int. Conf. on Software Engineering*, San Francisco, Oct.1976, 47-52.

11. Israel, J.E., Mitchell, J.G., and Sturgis, H.E. Separating data from function in a distributed file system. *Second Int. Symposium on Operating Systems*, IRIA, Rocquencourt, France, Oct. 1978.

12. Keedy, J.J. On structuring operating systems with monitors. *Australian Computer J*. **10**, 1 (Feb.1978), 23-27 (reprinted in *Operating Systems Review* **13**, 1 (Jan.1979), 5-9).

13. Lampson, B.W., Mitchell, J.G., and Satterthwaite, E.H. On the transfer of control between contexts. *Lecture Notes in Computer Science* **19**, Springer, 1974, 181-203.

14. Lauer. H.E., and Needham. R.M. On the duality of operating system structures. *Second Int. Symposium on Operating Systems*, IRIA, Rocquencourt, France, Oct. 1978 (reprinted in *Operating Systems Review* **13**,*2* (April 1979), 3-19).

15. Lister, AM., and Maynard. K.J. An implementation of monitors. *Software—Practice and Experience* **6**,3 (July 1976), 377-386.

16. Metcalfe. R.M., and Boggs, D.G. Ethernet: Packet switching for local computer networks. *Comm. ACM* **19**, *7* (July 1976), 395-403.

17. Mitchell. J.G., Maybury. W., and Sweet, R. *Mesa Language Manual*. Xerox Research Center, Palo Alto, Calif., 1979.

18. Redell, D., et al. Pilot: An operating system for a personal computer. *Comm. ACM* **23**,*2* (Feb.1980).

19. Saltzer, J.H. *Traffic Control in a Multiplexed Computer System*. MAC-TR-30, MIT, July 1966.

20. Saxena, A.R., and Bredt, T.H. A structured specification of a hierarchical operating system. *SIGPLAN Notices* **10**, 6 (June 1975), 310-318.

21. Wirth, N. Modula: A language for modular multi-programming. *Software—Practice and Experience* **7**, 1 (Jan.1977), 3-36.

Capriccio: Scalable Threads for Internet Services

Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer
Computer Science Division
University of California, Berkeley
{jrvb,jcondit,zf,necula,brewer}@cs.berkeley.edu

ABSTRACT

This paper presents Capriccio, a scalable thread package for use with high-concurrency servers. While recent work has advocated event-based systems, we believe that thread-based systems can provide a simpler programming model that achieves equivalent or superior performance.

By implementing Capriccio as a user-level thread package, we have decoupled the thread package implementation from the underlying operating system. As a result, we can take advantage of cooperative threading, new asynchronous I/O mechanisms, and compiler support. Using this approach, we are able to provide three key features: (1) scalability to 100,000 threads, (2) efficient stack management, and (3) resource-aware scheduling.

We introduce *linked stack management*, which minimizes the amount of wasted stack space by providing safe, small, and non-contiguous stacks that can grow or shrink at run time. A compiler analysis makes our stack implementation efficient and sound. We also present *resource-aware scheduling*, which allows thread scheduling and admission control to adapt to the system's current resource usage. This technique uses a *blocking graph* that is automatically derived from the application to describe the flow of control between blocking points in a cooperative thread package. We have applied our techniques to the Apache 2.0.44 web server, demonstrating that we can achieve high performance and scalability despite using a simple threaded programming model.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*threads*

General Terms

Algorithms, Design, Performance

Keywords

user-level threads, linked stack management, dynamic stack growth, resource-aware scheduling, blocking graph

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.
Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

1. INTRODUCTION

Today's Internet services have ever-increasing scalability demands. Modern servers must be capable of handling tens or hundreds of thousands of simultaneous connections without significant performance degradation. Current commodity hardware is capable of meeting these demands, but software has lagged behind. In particular, there is a pressing need for a programming model that allows programmers to design efficient and robust servers with ease.

Thread packages provide a natural abstraction for high-concurrency programming, but in recent years, they have been supplanted by event-based systems such as SEDA [41]. These event-based systems handle requests using a pipeline of stages. Each request is represented by an event, and each stage is implemented as an event handler. These systems allow precise control over batch processing, state management, and admission control; in addition, they provide benefits such as atomicity within each event handler.

Unfortunately, event-based programming has a number of drawbacks when compared to threaded programming [39]. Event systems hide the control flow through an application, making it difficult to understand cause and effect relationships when examining source code and when debugging. For instance, many event systems invoke a method in another module by sending a "call" event and then waiting for a "return" event in response. In order to understand the application, the programmer must mentally match these call/return pairs, even when they are in different parts of the code. Furthermore, creating these call/return pairs often requires the programmer to manually save and restore live state. This process, referred to as "stack ripping" [1], is a major burden for programmers who wish to use event systems.

In this paper, we advocate a different solution: instead of switching to an event-based model to achieve high concurrency, we should fix the thread-based model. We believe that a modern thread package will be able to provide the same benefits as an event system while also offering a better programming model for Internet services. Specifically, our goals for our revised thread package are:

- Support for existing thread APIs.
- Scalability to hundreds of thousands of threads.
- Flexibility to address application-specific needs.

In meeting these goals, we have made it possible for programmers to write high-performance Internet servers using the intuitive one-thread-per-connection programming style.

Indeed, our thread package can improve performance of existing threaded applications with little to no modification of the application itself.

1.1 Thread Design Principles

In the process of "fixing" threads for use in server applications, we found that a user-level approach is essential. While user-level threads and kernel threads are both useful, they solve fundamentally different problems. Kernel threads are primarily useful for enabling true concurrency via multiple devices, disk requests, or CPUs. User-level threads are really logical threads that should provide a clean programming model with useful invariants and semantics.

To date, we do not strongly advocate any *particular* semantics for threads; rather, we argue that any clean semantics for threads requires *decoupling* the threads of the programming model (logical threads) from those of the underlying kernel. Decoupling the programming model from the kernel is important for two reasons. First, there is substantial variation in interfaces and semantics among modern kernels, despite the existence of the POSIX standard. Second, kernel threads and asynchronous I/O interfaces are areas of active research [22, 23]. The range of semantics and the rate of evolution both require decoupling: logical threads can hide both OS variation and kernel evolution.

In our case, this decoupling has provided a number of advantages. We have been able to integrate compiler support into our thread package, and we have taken advantage of several new kernel features. Thus, we have been able to increase performance, improve scalability, and address application-specific needs, all without changing application code.

1.2 Capriccio

This paper discusses our new thread package, Capriccio. This thread package achieves our goals with the help of three key features:

First, we improved the scalability of basic thread operations. We accomplished this task by using user-level threads with cooperative scheduling, by taking advantage of a new asynchronous I/O interface, and by engineering our runtime system so that all thread operations are $O(1)$.

Second, we introduced *linked stacks*, a mechanism for dynamic stack growth that solves the problem of stack allocation for large numbers of threads. Traditional thread systems preallocate large chunks of memory for each thread's stack, which severely limits scalability. Capriccio uses a combination of compile-time analysis and run-time checks to limit the amount of wasted stack space in an efficient and application-specific manner.

Finally, we designed a *resource-aware scheduler*, which extracts information about the flow of control within a program in order to make scheduling decisions based on predicted resource usage. This scheduling technique takes advantage of compiler support and cooperative threading to address application-specific needs without requiring the programmer to modify the original program.

The remainder of this paper discusses each of these three features in detail. Then, we present an overall experimental evaluation of our thread package. Finally, we discuss future directions for user-level thread packages with integrated compiler support.

2. THREAD DESIGN AND SCALABILITY

Capriccio is a fast, user-level thread package that supports the POSIX API for thread management and synchronization. In this section, we discuss the overall design of our thread package, and we demonstrate that it satisfies our scalability goals.

2.1 User-Level Threads

One of the first issues we explored when designing Capriccio was whether to employ user-level threads or kernel threads. User-level threads have some important advantages for both performance and flexibility. Unfortunately, they also complicate preemption and can interact badly with the kernel scheduler. Ultimately, we decided that the advantages of user-level threads are significant enough to warrant the additional engineering required to circumvent their drawbacks.

2.1.1 Flexibility

User-level threads provide a tremendous amount of flexibility for system designers by creating a level of indirection between applications and the kernel. This abstraction helps to decouple the two, and it allows faster innovation on both sides. For example, Capriccio is capable of taking advantage of the new asynchronous I/O mechanisms the development-series Linux kernel, which allows us to provide performance improvements without changing application code.

The use of user-level threads also increases the flexibility of the thread scheduler. Kernel-level thread scheduling must be general enough to provide a reasonable level of quality for all applications. Thus, kernel threads cannot tailor the scheduling algorithm to fit a specific application. Fortunately, user-level threads do not suffer from this limitation. Instead, the user-level thread scheduler can be built along with the application.

User-level threads are extremely lightweight, which allows programmers to use a tremendous number of threads without worrying about threading overhead. The benchmarks in Section 2.3 show that Capriccio can scale to $100,000$ threads; thus, Capriccio makes it possible to write highly concurrent applications (which are often written with messy, event-driven code) in a simple threaded style.

2.1.2 Performance

User-level threads can greatly reduce the overhead of thread synchronization. In the simplest case of cooperative scheduling on a single CPU, synchronization is nearly free, since neither user threads nor the thread scheduler can be interrupted while in a critical section.[1] In the future, we believe that flexible user-level scheduling and compile-time analysis will allow us to offer similar advantages on a multi-CPU machine.

Even in the case of preemptive threading, user-level threads offer an advantage in that they do not require kernel crossings for mutex acquisition or release. By comparison, kernel-level mutual exclusion requires a kernel crossing for every synchronization operation. While this situation can be improved for uncontended locks,[2] highly contended mutexes still require kernel crossings.

[1] Poorly designed signal handling code can reintroduce these problems, but this problem can easily be avoided.

[2] The `futexes` in recent Linux kernels allow operations on uncontended mutexes to occur entirely in user space.

Finally, memory management is more efficient with user-level threads. Kernel threads require data structures that eat up valuable kernel address space, decreasing the space available for I/O buffers, file descriptors, and other resources.

2.1.3 Disadvantages

User-level threading is not without its drawbacks, however. In order to retain control of the processor when a user-level thread executes a blocking I/O call, a user-level threading package overrides these blocking calls and replaces them internally with non-blocking equivalents. The semantics of these non-blocking I/O mechanisms generally require an increased number of kernel crossings when compared to the blocking equivalents. For example, the most efficient non-blocking network I/O primitive in Linux (`epoll`) involves first polling sockets for I/O readiness and then performing the actual I/O call. These second I/O calls are identical to those performed in the blocking case; the poll calls are additional overhead. Non-blocking disk I/O mechanisms are often similar in that they employ separate system calls to submit requests and retrieve responses.[3]

In addition, user-level thread packages must introduce a wrapper layer that translates blocking I/O mechanisms to non-blocking I/O ones, and this layer is another source of overhead. At best, this layer can be a very thin shim, which simply adds a few extra function calls. However, for quick operations such as in-cache reads that are easily satisfied by the kernel, this overhead can become important.

Finally, user-level threading can make it more difficult to take advantage of multiple processors. The performance advantage of lightweight synchronization is diminished when multiple processors are allowed, since synchronization is no longer "for free". Additionally, as discussed by Anderson et al. in their work on scheduler activations, purely user-level synchronization mechanisms are ineffective in the face of true concurrency and may lead to starvation [2].

Ultimately, we believe the benefits of user-level threading far outweigh these disadvantages. As the benchmarks in Section 2.3 show, the additional overhead incurred does not seem to be a problem in practice. In addition, we are working on ways to overcome the difficulties with multiple processors; we will discuss this issue further in Section 7.

2.2 Implementation

We have implemented Capriccio as a user-level threading library for Linux. Capriccio implements the POSIX threading API, which allows it to run most applications without modification.

Context Switches. Capriccio is built on top of Edgar Toernig's coroutine library [35]. This library provides extremely fast context switches for the common case in which threads voluntarily yield, either explicitly or through making a blocking I/O call. We are currently designing signal-based code that allows for preemption of long-running user threads, but Capriccio does not provide this feature yet.

I/O. Capriccio intercepts blocking I/O calls at the library level by overriding the system call stub functions in GNU libc. This approach works flawlessly for statically linked applications and for dynamically linked applications that use GNU libc versions 2.2 and earlier. However, GNU libc version 2.3 bypasses the system call stubs for many of its internal routines (such as `printf`), which causes problems for dynamically linked applications. We are working to allow Capriccio to function as a libc add-on in order to provide better integration with the latest versions of GNU libc.

Internally, Capriccio uses the latest Linux asynchronous I/O mechanisms—`epoll` for pollable file descriptors (e.g., sockets, pipes, and fifos) and Linux AIO for disk. If these mechanisms are not available, Capriccio falls back on the standard Unix `poll()` call for pollable descriptors and a pool of kernel threads for disk I/O. Users can select among the available I/O mechanisms by setting appropriate environment variables prior to starting their application.

Scheduling. Capriccio's main scheduling loop looks very much like an event-driven application, alternately running application threads and checking for new I/O completions. Note, though, that the scheduler hides this event-driven behavior from the programmer, who still uses the standard thread-based abstraction. Capriccio has a modular scheduling mechanism that allows the user to easily select between different schedulers at run time. This approach has also made it simple for us to develop several different schedulers, including a novel scheduler based on thread resource utilization. We discuss this feature in detail in Section 4.

Synchronization. Capriccio takes advantage of cooperative scheduling to improve synchronization. At present, Capriccio supports cooperative threading on single-CPU machines, in which case inter-thread synchronization primitives require only simple checks of a boolean locked/unlocked flag. For cases in which multiple kernel threads are involved, Capriccio employs either spin locks or optimistic concurrency control primitives, depending on which mechanism best fits the situation.

Efficiency. In developing Capriccio, we have taken great care to choose efficient algorithms and data structures. Consequently, all but one of Capriccio's thread management functions has a bounded worst-case running time, independent of the number of threads. The sole exception is the sleep queue, which currently uses a naive linked list implementation. While the literature contains a number of good algorithms for efficient sleep queues, our current implementation has not caused problems yet, so we have focused our development efforts on other aspects of the system.

2.3 Threading Microbenchmarks

We ran a number of microbenchmarks to validate Capriccio's design and implementation. Our test platform was an SMP with two 2.4 GHz Xeon processors, 1 GB of memory, two 10K RPM SCSI Ultra II hard drives, and 3 Gigabit Ethernet interfaces. The operating system was Linux 2.5.70, which includes support for `epoll`, asynchronous disk I/O, and lightweight system calls (`vsyscall`). We ran our benchmarks on three thread packages: Capriccio, LinuxThreads (the standard Linux kernel thread package), and NPTL version 0.53 (the new Native POSIX Threads for Linux

[3] Although there are non-blocking I/O mechanisms (such as POSIX AIO's `lio_listio()` and Linux's new `io_submit()`) that allow the submission of multiple I/O requests with a single system call, there are other issues that make this feature difficult to use. For example, implementations of POSIX AIO often suffer from performance problems. Additionally, use of batching creates a trade-off between system call overhead and I/O latency, which is difficult to manage.

	Capriccio	Capriccio_notrace	LinuxThreads	NPTL
Thread creation	21.5	21.5	37.9	17.7
Thread context switch	0.56	0.24	0.71	0.65
Uncontended mutex lock	0.04	0.04	0.14	0.15

Table 1: Latencies (in μs) of thread primitives for different thread packages.

package). We built all applications with gcc 3.3 and linked against GNU libc 2.3. We recompiled LinuxThreads to use the new lightweight system call feature of latest Linux kernels to ensure a fair comparison with NPTL, which uses this feature.

2.4 Thread Primitives

Table 1 compares average times of several thread primitives for Capriccio, LinuxThreads, and NPTL. In the test labeled `Capriccio_notrace`, we disabled statistics collection and dynamic stack backtracing (used for the scheduler discussed in Section 4) to show their impact on performance. Thread creation time is dominated by stack allocation time and is quite expensive for all four thread packages. Thread context switches, however, are significantly faster in Capriccio, even with the stack tracing and statistics collection overhead. We believe that reduced kernel crossings and our simpler scheduling policy both contributed to this result. Synchronization primitives are also much faster in Capriccio (by a factor of 4 for uncontended mutex locking) because no kernel crossings are involved.

2.5 Thread Scalability

To measure the overall efficiency and scalability of scheduling and synchronization in different thread packages, we ran a simple producer-consumer microbenchmark on the three packages. Producers put empty messages into a shared buffer, and consumers "process" each message by looping for a random amount of time. Synchronization is implemented using condition variables and mutexes. Equal numbers of producers and consumers are created for each test. Each test is run for 10 seconds and repeated 5 times. Average throughput and standard deviations are shown in Figure 1. Capriccio outperforms NPTL and LinuxThreads in terms of both raw performance and scalability. Throughput of LinuxThreads begins to degrade quickly after only 20 threads are created, and NPTL's throughput degrades after 100. NPTL shows unstable behavior with more than 64 threads, which persists across two NPTL versions (0.53 and 0.56) and several 2.5 series kernels we tested. Capriccio scales to 32K producers and consumers (64K threads total). We attribute the drop of throughput between 100 threads and 1000 to increased cache footprint.

2.6 I/O Performance

Figure 2 shows the network performance of Capriccio and other thread packages under load. In this test, we measured the throughput of concurrently passing a number of tokens (12 bytes each) among a fixed number of pipes. The number of concurrent tokens is one quarter of the number of pipes if there are less than 128 pipes; otherwise, there are exactly 128 tokens. The benchmark thus simulates the effect of slow client links—that is, a large number of mostly-idle pipes. This scenario is typical for Internet servers, and traditional threading systems often perform poorly in such tests. Two functionally equivalent benchmark programs are used to obtain the results: a threaded version is used for Capriccio, LinuxThreads, and NPTL, and a non-blocking I/O version is used for `poll` and `epoll`. Five million tokens are passed for each test and each test is run five times.

The figure shows that Capriccio scales smoothly to 64K threads and incurs less than 10% overhead when compared to `epoll` with more than 256 pipes. To our knowledge, `epoll` is the best non-blocking I/O mechanism available on Linux; hence, its performance should reflect that of the best event-based servers, which all rely on such a mechanism. Capriccio performs consistently better than Poll, LinuxThreads, and NPTL with more than 256 threads and is more than twice as fast as both LinuxThreads and NPTL when more than 1000 threads are created.

However, when concurrency is low (< 100 pipes), Capriccio is slower than its competitors because it issues more system calls. In particular, it calls `epoll_wait()` to obtain file descriptor readiness events to wake up threads blocking for I/O. It performs these calls periodically, transferring as many events as possible on each call. However, when concurrency is low, the number of runnable threads occasionally reaches zero, forcing Capriccio to issue more `epoll_wait()` calls. In the worst case, Capriccio is 37% slower than NPTL when there are only 2 concurrent tokens (and 8 threads). Fortunately, this overhead is amortized quickly when concurrency increases; more scalable scheduling allows Capriccio to outperform LinuxThreads and NPTL at high concurrency.

Since Capriccio uses asynchronous I/O primitives, Capriccio can benefit from the kernel's disk head scheduling algorithm just as much as kernel threads can. Figure 3 shows a microbenchmark in which a number of threads perform random 4 KB reads from a 1 GB file. The test program bypasses the kernel buffer cache by using `O_DIRECT` when opening the file. Each test is run for 10 seconds and averages of 5 runs are shown. Throughput of all three thread libraries increases steadily with the concurrency level until it levels off when concurrency reaches about 100. In contrast, utilization of the kernel's head scheduling algorithm in event-based systems that use blocking disk I/O (e.g., SEDA) is limited by the number of kernel threads used, which is often made deliberately small to reduce kernel scheduling overhead. Even worse, other process-based applications that use non-blocking I/O (either `poll()`, `select()`, `/dev/poll`, or `epoll`) cannot benefit from the kernel's head scheduling at all if they do not explicitly use asynchronous I/O. Unfortunately, most programs do not use asynchronous I/O because it significantly increases programming complexity and compromises portability.

Figure 4 shows disk I/O performance of the three thread libraries when using the OS buffer cache. In this test, we measure the throughput achieved when 200 threads read continuously 4K blocks from the file system with a specified buffer cache miss rate. The cache miss rate is fixed by reading an appropriate portion of data from a small file opened normally (hence all cache hits) and by reading the

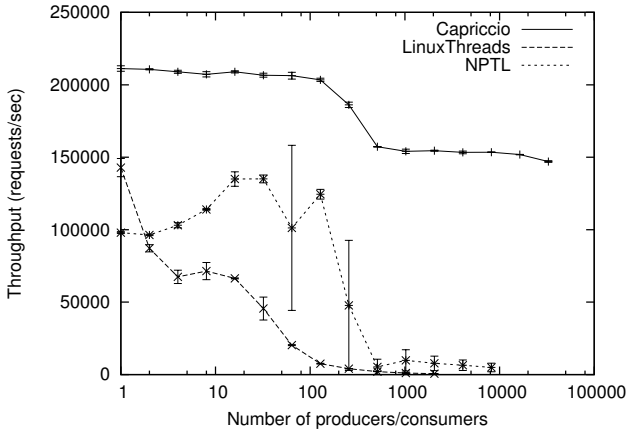


Figure 1: `Producer-Consumer` - scheduling and synchronization performance.

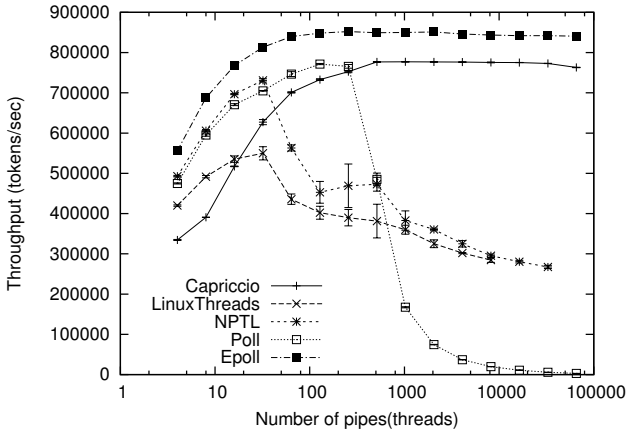


Figure 2: `Pipetest` - network scalability test.

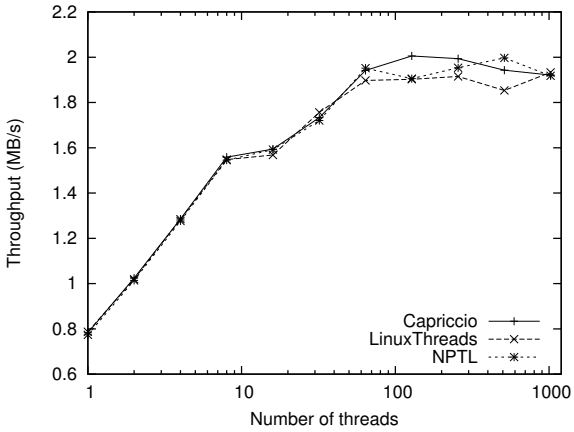


Figure 3: Benefits of disk head scheduling.

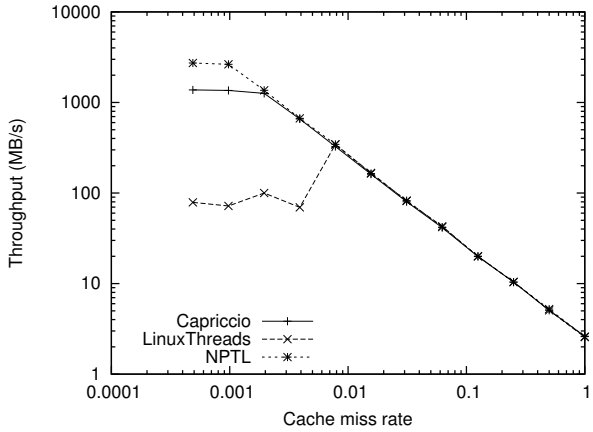


Figure 4: Disk I/O performance with buffer cache.

remaining data from a file opened with `O_DIRECT`. For a higher miss rate, the test is disk-bound; thus, Capriccio's performance is identical to that of NPTL and LinuxThreads. However, when the miss rate is very low, the program is CPU-bound, so throughput is limited by per-transfer overhead. Here, Capriccio's maximum throughput is about 50% of NPTL's, which means Capriccio's overhead is twice that of NPTL. The source of this overhead is the asynchronous I/O interface (Linux AIO) used by Capriccio, which incurs the same amount of overhead for cache-hitting operations and for ones that reach the disk: for each I/O request, a completion event needs to be constructed, queued, and delivered to user-level through a separate system call. However, this shortcoming is relatively easy to fix: by returning the result immediately for requests that do not need to wait, we can eliminate most (if not all) of this overhead. We leave this modification as future work. Finally, LinuxThreads' performance degrades significantly at a very low miss rate. We believe this degradation is a result of a bug either in the kernel or in the library, since the processor is mostly idle during the test.

3. LINKED STACK MANAGEMENT

Thread packages usually attempt to provide the programmer with the abstraction of an unbounded call stack for each thread. In reality, the stack size is bounded, but the bounds are chosen conservatively so that there is plenty of space for normal program execution. For example, LinuxThreads allocates two megabytes per stack by default; with such a conservative allocation scheme, we consume 1 GB of virtual memory for stack space with just 500 threads. Fortunately, most threads consume only a few kilobytes of stack space at any given time, although they might go through stages when they use considerably more. This observation suggests that we can significantly reduce the size of virtual memory dedicated to stacks if we adopt a dynamic stack allocation policy wherein stack space is allocated to threads on demand in relatively small increments and is deallocated when the thread requires less stack space. In the rest of this section, we discuss a compiler feature that allows us to provide such a mechanism while preserving the programming abstraction of unbounded stacks.

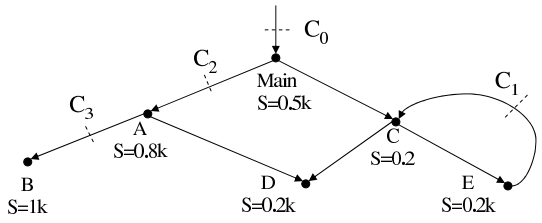


Figure 5: An example of a call graph annotated with stack frame sizes. The edges marked with C_i (i=$0, \ldots, 3$) are the checkpoints.

3.1 Compiler Analysis and Linked Stacks

Our approach uses a compiler analysis to limit the amount of stack space that must be preallocated. We perform a whole-program analysis based on a *weighted call graph*.[4] Each function in the program is represented by a node in this call graph, weighted by the maximum amount of stack space that a single stack frame for that function will consume. An edge between node A and node B indicates that function A calls function B directly. Thus, paths between nodes in this graph correspond to sequences of stack frames that may appear on the stack at run time. The length of a path is the sum of the weights of all nodes in this path; that is, it is the total size of the corresponding sequence of stack frames. An example of such a graph is shown in Figure 5.

Using this call graph, we wish to place a reasonable bound on the amount of stack space that will be consumed by each thread. If there are no recursive functions in our program, there will be no cycles in the call graph, and thus we can easily bound the maximum stack size for the program at compile time by finding the longest path starting from each thread's entry point. However, most real-world programs make use of recursion, which means that we cannot compute a bound on the stack size at compile time. And even in the absence of recursion, the static computation of stack size might be too conservative. For example, consider the call graph in Figure 5. Ignoring the cycle in the graph, the maximum stack size is 2.3 KB on the path `Main`–`A`–`B`. However, the path `Main`–`C`–`D` has a smaller stack size of only 0.9 KB. If the first path is only used during initialization and the second path is used through the program's execution, then allocating 2.3 KB to each thread would be wasteful. For these reasons, it is important to be able to grow and shrink the stack size on demand.

In order to implement dynamically-sized stacks, our call graph analysis identifies call sites at which we must insert *checkpoints*. A checkpoint is a small piece of code that determines whether there is enough stack space left to reach the next checkpoint without causing stack overflow. If not enough space remains, a new *stack chunk* is allocated, and the stack pointer is adjusted to point to this new chunk. When the function call returns, the stack chunk is unlinked and returned to a free list.

This scheme results in non-contiguous stacks, but because the stack chunks are switched right before the actual arguments for a function call are pushed, the code for the callee need not be changed. And because the caller's frame pointer is stored on the callee's stack frame, debuggers can follow the backtrace of a program.[5] The code for a checkpoint is written in C, with a small amount of inline assembly for reading and setting of the stack pointer; this code is inserted using a source-to-source transformation of the program prior to compilation. Mutual exclusion for accessing the free stack chunk list is ensured by our cooperative threading approach.

3.2 Placing Checkpoints

During our program analysis, we must determine where to place checkpoints. A simple solution is to insert checkpoints at every call site; however, this approach is prohibitively expensive. A less restrictive approach is to ensure that at each checkpoint, we have a bound on the stack space that may be consumed before we reach the next checkpoint (or a leaf in the call graph).

To satisfy this requirement, we must ensure that there is at least one checkpoint in every cycle within the call graph (recall that the edges in the call graph correspond to call sites). To find the appropriate points to insert checkpoints, we perform a depth-first search on the call graph, which identifies back edges—that is, edges that connect a node to one of its ancestors in the call graph [25]. All cycles in the graph must contain a back edge, so we add checkpoints at all call sites identified as back edges in order to ensure that any path from a function to a checkpoint has bounded length. In Figure 5, the checkpoint C_0 allocates the first stack chunk, and the checkpoint C_1 is inserted on the back edge `E`–`C`.

Even after we break all cycles, the bounds on stack size may be too large. Thus, we add additional checkpoints to the graph to ensure that all paths between checkpoints are within a desired bound, which is given as a compile-time parameter. To insert these new checkpoints, we process the call graph once more, this time determining the longest path from each node to the next checkpoint or leaf. When performing this analysis, we consider a restricted call graph that does not contain any back edges, since these edges already have checkpoints. This restricted graph has no cycles, so we can process the nodes bottom-up; thus, when processing node n, we will have already determined the longest path for each of n's successors. So, for each successor s of node n, we take the longest path for s and add n. If this new path's length exceeds the specified path limit parameter, we add a checkpoint to the edge between n and s, which effectively reduces the longest path of s to zero. The result of this algorithm is a set of edges where checkpoints should be added along with reasonable bounds on the maximum path length from each node. For the example in Figure 5, with a limit of 1 KB, this algorithm places the additional checkpoints C_2 and C_3. Without the checkpoint C_2, the stack frames of `Main` and `A` would use more than 1 KB.

Figure 6 shows four instances in the lifetime of the thread whose call graph is shown in Figure 5. In Figure 6(a), the function B is executing, with three stack chunks allocated at checkpoints C_0, C_2, and C_3. Notice that 0.5 KB is wasted in the first stack chunk, and 0.2 KB is wasted in the second

[4] We use the CIL toolkit [26] for this purpose, which allows efficient whole-program analysis of real-world applications like the Apache web server.

[5] This scheme does not work when the `omit-frame-pointer` is enabled in `gcc`. It is possible to support this optimization by using more expensive checkpoint operations such as copying the arguments from the caller's frame to the callee's frame.

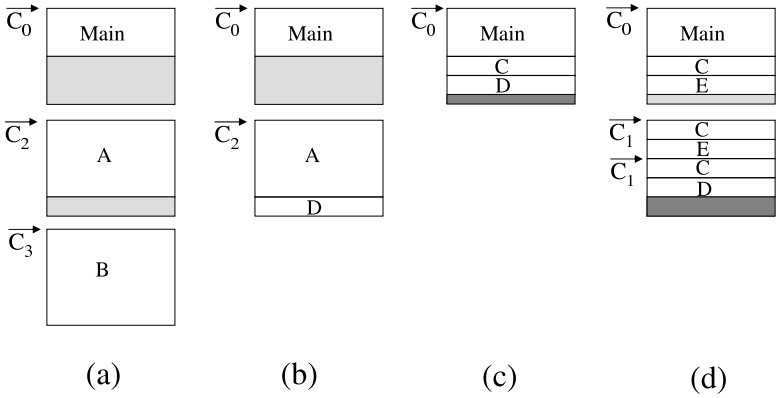


Figure 6: Examples of dynamic allocation and deallocation of stack chunks.

chunk. In Figure 6(b), function A has called D, and only two stack chunks were necessary. Finally, in Figure 6(d) we see an instance with recursion. A new stack chunk is allocated when E calls C (at checkpoint C_1). However, the second time around, the code at checkpoint C_1 decides that there is enough space remaining in the current stack chunk to reach either a leaf function (D) or the next checkpoint (C_1).

3.3 Dealing with Special Cases

Function pointers present an additional challenge to our algorithm, because we do not know at compile time exactly which function may be called through a given function pointer. To improve the results of our analysis, though, we want to determine as precisely as possible the set of functions that might be called at a function pointer call site. Currently, we categorize function pointers by number and type of arguments, but in the future, we plan to use a more sophisticated pointer analysis.

Calls to external functions also cause problems, since it is more difficult to bound the stack space used by precompiled libraries. We provide two solutions to this problem. First, we allow the programmer to annotate external library functions with trusted stack bounds. Alternatively, we allow larger stack chunks to be linked for external functions; as long as threads don't block frequently within these functions, we can reuse a small number of large stack chunks throughout the application. For the C standard library, we use annotations to deal with functions that block or functions that are frequently called; these annotations were derived by analyzing library code.

3.4 Tuning the Algorithm

Our algorithm causes stack space to be wasted in two places. First, some stack space is wasted when a new stack chunk is linked; we call this space *internal wasted space*. Second, stack space at the bottom of the current chunk is considered unused; this space is called *external wasted space*. In Figure 6, internal wasted space is shown in light gray, whereas external wasted space is shown in dark gray.

The user is allowed to tune two parameters that adjust the trade-offs in terms of wasted space and execution speed. First, the user can adjust *MaxPath*, which specifies the maximum desired path length in the algorithm we have just described. This parameter affects the trade-off between execution time and internal wasted space; larger path lengths require fewer checkpoints but more stack linking. Second, the user can adjust *MinChunk*, the minimum stack chunk size. This parameter affects the trade-off between stack linking and external wasted space; larger chunks result in more external wasted space but less frequent stack linking, which in turn results in less internal wasted space and a smaller execution time overhead. Overall, these parameters provide a useful mechanism allowing the user (or the compiler) to optimize memory usage.

3.5 Memory Benefits

Our linked stack technique has a number of advantages in terms of memory performance. In general, these benefits are achieved by divorcing thread implementation from kernel mechanisms, thus improving our ability to tune individual application memory usage. Compiler techniques make this application-specific tuning practical.

First, our technique makes preallocation of large stacks unnecessary, which in turn reduces virtual memory pressure when running large numbers of threads. Our analysis achieves this goal without the use of guard pages, which would contribute unnecessary kernel crossings and virtual memory waste.

Second, using linked stacks can improve paging behavior significantly. Linked stack chunks are reused in LIFO order, which allows stack chunks to be shared between threads, reducing the size of the application's working set. Also, we can allocate stack chunks that are smaller than a single page, thus reducing the overall amount of memory waste.

To demonstrate the benefit of our approach with respect to paging, we created a microbenchmark in which each thread repeatedly calls a function `bigstack()`, which touches all pages of a 1 MB buffer on the stack. Threads yield between calls to `bigstack()`. Our compiler analysis inserts a checkpoint at these calls, and the checkpoint causes a large stack chunk to be linked only for the duration of the call. Since `bigstack()` does not yield, all threads share a single 1 MB stack chunk; without our stack analysis, we would have to give each thread its own individual 1 MB stack.

We ran this microbenchmark with 800 threads, each of which calls `bigstack()` 10 times. We recorded execution time for five runs of the test and averaged the results. When each thread has its own individual stack, the benchmark takes 3.33 seconds, 1.07 seconds of which are at user level. When using our stack analysis, the benchmark takes 1.04

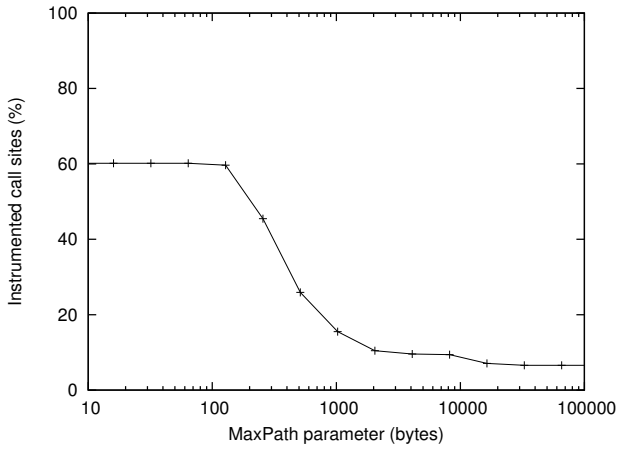


Figure 7: Number of Apache 2.0.44 call sites instrumented as a function of the *MaxPath* parameter.

seconds, with 1.00 seconds at user level. All standard deviations were within 0.02 seconds. The fact that total exeuction time decreases by a factor of three while user-level execution time remains roughly the same suggests that sharing a single stack via our linked stack mechanism drastically reduces the cost of paging. When running this test with 1,000 threads, the version without our stack analysis starts thrashing; with the stack analysis, though, the running time scales linearly up to 100,000 threads.

3.6 Case Study: Apache 2.0.44

We applied this analysis to the Apache 2.0.44 web server. We set the *MaxPath* parameter to 2 KB; this choice was made by examining the number of call sites instrumented for various parameter values. The results, shown in Figure 7, indicate that 2 KB or 4 KB is a reasonable choice, since larger parameter values make little difference in the overall amount of instrumentation. We set the *MinChunk* parameter to 4 KB based on profiling information. By adding profiling counters to checkpoints, we determined that increasing the chunk size to 4 KB reduced the number of stack links and unlinks significantly, but further increases yielded no additional benefit. We expect that this tuning methodology can be automated as long as the programmer supplies a reasonable profiling workload.

Using these parameters, we studied the behavior of Apache during execution of a workload consisting of static web pages based on the SPECweb99 benchmark suite. We used the threaded client program from the SEDA work [41] with 1000 simulated clients, a 10ms request delay, and a total file workload of 32 MB. The server ran 200 threads, using standard Unix `poll()` for network I/O and blocking for disk I/O. The total virtual memory footprint for Apache was approximately 22 MB, with a resident set size of approximately 10 MB. During this test, most functions could be executed entirely within the initial 4 KB chunk; when necessary, though, threads linked a 16 KB chunk in order to call a function that has an 8 KB buffer on its stack. Over five runs of this benchmark, the maximum number of 16 KB chunks needed at any given time had a mean of 66 (standard deviation 4.4). Thus, we required just under 8 MB of stack space overall: 800 KB for the initial stacks, 1 MB for larger chunks, and 6 MB for three 2 MB chunks used to run external functions. However, we believe that additional 16 KB chunks will be needed when using high-performance I/O mechanisms; we are still in the process of studying the impact of these features on stack usage. And while using an average of 66 16 KB buffers rather than one for each of the 200 threads is clearly a win, the addition of internal and external wasted space makes it difficult to directly compare our stack utilization with that of unmodified Apache. Nevertheless, this example shows that we are capable of running unmodified applications with a small amount of stack space without fear of stack overflow. Indeed, it is important to note that we provide safety in addition to efficiency; even though the unmodified version of Apache could run this workload with a single, contiguous 20 KB stack, this setting may not be safe for other workloads or for different configurations of Apache.

We observed the program's behavior at each call site crossed during the execution of this benchmark. The results were extremely consistent across five repetitions of the benchmark; thus, the numbers below represent the entire range of results over all five repetitions. At 0.1% of call sites, checkpoints caused a new stack chunk to be linked, at a cost of 27 instructions. At 0.4–0.5% of call sites, a large stack chunk was linked unconditionally in order to handle an external function, costing 20 instructions. At 10% of call sites, a checkpoint determined that a new chunk was not required, which cost 6 instructions. The remaining 89% of call sites were unaffected. Assuming all instructions are roughly equal in cost, the result is a 71–73% slowdown when considering function calls alone. Since call instructions make up only 5% of the program's instructions, the overall slowdown is approximately 3% to 4%.

4. RESOURCE-AWARE SCHEDULING

One of the advantages claimed for event systems is that their scheduling can easily adapt to the application's needs. Event-based applications are broken into distinct event handlers, and computation for a particular task proceeds as that task is passed from handler to handler. This architecture provides two pieces of information that are useful for scheduling. First, the current handler for a task provides information about the task's location in the processing chain. This information can be used to give priority to tasks that are closer to completion, hence reducing load on the system. Second, the lengths of the handlers' task queues can be used to determine which stages are bottlenecks and can indicate when the server is overloaded.

Capriccio provides similar application-specific scheduling for thread-based applications. Since Capriccio uses a cooperative threading model, we can view an application as a sequence of stages, where the stages are separated by blocking points. In this sense, Capriccio's scheduler is quite similar to an event-based system's scheduler. Our methods are more powerful, however, in that they deduce the stages automatically and have direct knowledge of the resources used by each stage, thus enabling finer-grained dynamic scheduling decisions. In particular, we use this automated scheduling to provide admission control and to improve response time.

Our approach allows Capriccio to provide sophisticated, application-specific scheduling without requiring the programmer to use complex or brittle tuning APIs. Thus, we

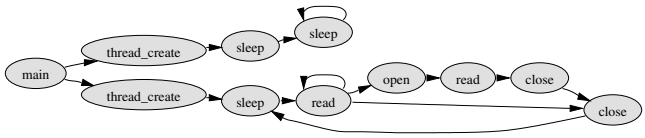


Figure 8: An example blocking graph. This graph was generated from a run of Knot, our test web server.

can improve performance and scalability without compromising the simplicity of the threaded programming model.

4.1 Blocking Graph

The key abstraction we use for scheduling is the *blocking graph*, which contains information about the places in the program where threads block. Each node is a location in the program that blocked, and an edge exists between two nodes if they were consecutive blocking points. The "location" in the program is not merely the value of the program counter, but rather the call chain that was used to reach the blocking point. This path-based approach allows us to differentiate blocking points in a more useful way than the program counter alone would allow, since otherwise there tend to be very few such points (e.g., the `read` and `write` system calls). Figure 8 shows the blocking graph for Knot, a simple thread-based web server. Each thread walks this graph independently, and every blocked thread is located at one of these nodes.

Capriccio generates this graph at run time by observing the transitions between blocking points. The key idea behind this approach is that Capriccio can *learn* the behavior of the application dynamically and then use that information to improve scheduling and admission control. This technique works in part because we are targeting long-running programs such as Internet servers, so it is acceptable to spend time learning in order to make improved decisions later on.

To make use of this graph when scheduling threads, we must annotate the edges and nodes with information about thread behavior. The first annotation we introduce is the average running time for each edge. When a thread blocks, we know which edge was just traversed, since we know the previous node. We measure the time it took to traverse the edge using the cycle counter, and we update an exponentially weighted average for that edge.

We keep a similar weighted average for each node, which we update every time a thread traverses one of its outgoing edges. Each node's average is essentially a weighted average of the edge values, since the number of updates is proportional to the number of times each outgoing edge is taken. The node value thus tells us how long the next edge will take *on average*.

Finally, we annotate the changes in resource usage. Currently, we define resources as memory, stack space, and sockets, and we track them individually. As with CPU time, there are weighted averages for both edges and nodes. Given that a blocked thread is located at a particular node, these annotations allows us to estimate whether running this thread will increase or decrease the thread's usage of each resource. This estimate is the basis for resource-aware scheduling: once we know that a resource is scarce, we promote nodes (and thus threads) that release that resource and demote nodes that acquire that resource.

4.2 Resource-Aware Scheduling

Most existing event systems prioritize event handlers statically. SEDA uses information such as event handler queue lengths to dynamically tune the system. Capriccio goes one step further by introducing the notion of resource-aware scheduling. In this section, we show how to use the blocking graph to perform resource-aware scheduling that is both transparent and application-specific.

Our strategy for resource-aware scheduling has three parts:

1. Keep track of resource utilization levels and decide dynamically if each resource is at its limit.
2. Annotate each node with the resources used on its outgoing edges so we can predict the impact on each resource should we schedule threads from that node.
3. Dynamically prioritize nodes (and thus threads) for scheduling based on information from the first two parts.

For each resource, we increase utilization until it reaches maximum capacity (so long as we don't overload another resource), and then we throttle back by scheduling nodes that release that resource. When resource usage is low, we want to preferentially schedule nodes that consume that resource, under the assumption that doing so will increase throughput. More importantly, when a resource is overbooked, we preferentially schedule nodes that release the resource to avoid thrashing.

This combination, when used with some hysteresis, tends to keep the system at full throttle without the risk of thrashing. Additionally, resource-aware scheduling provides a natural, workload-sensitive form of admission control, since tasks near completion tend to release resources, whereas new tasks allocate them. This strategy is completely adaptive, in that the scheduler responds to changes resource consumption due to both the type of work being done and offered load. The speed of adaptation is controlled by the parameters of the exponentially weighted averages in our blocking graph annotations.

Our implementation of resource-aware scheduling is quite straightforward. We maintain separate run queues for each node in the blocking graph. We periodically determine the relative priorities of each node based on our prediction of their subsequent resource needs and the overall resource utilization of the system. Once the priorities are known, we select a nodes by stride scheduling, and then we select threads within nodes by dequeuing from the nodes' run queues. Both of these operations are $O(1)$.

A key underlying assumption of our resource-aware scheduler is that resource usage is likely to be similar for many tasks at a blocking point. Fortunately, this assumption seems to hold in practice. With Apache, for example, there is almost no variation in resource utilization along the edges of the blocking graph.

4.2.1 Resources

The resources we currently track are CPU, memory, and file descriptors. We track memory usage by providing our own version of the `malloc()` family. We detect the resource limit for memory by watching page fault activity.

For file descriptors, we track the `open()` and `close()` calls. This technique allows us to detect an increase in open file descriptors, which we view as a resource. Currently, we set the resource limit by estimating the number of open connections at which response time jumps up.

We can also track virtual memory usage and number of threads, but we do not do so at present. VM is tracked the same way as physical memory, but the limit is reached when we reach some absolute threshold for total VM allocated (e.g., 90% of the full address space).

4.2.2 Pitfalls

We encountered some interesting pitfalls when implementing Capriccio's resource-aware scheduler. First, determining the maximum capacity of a particular resource can be tricky. The utilization level at which thrashing occurs often depends on the workload. For example, the disk subsystem can sustain far more requests per second if the requests are sequential instead of random. Additionally, resources can interact, as when the VM system trades spare disk bandwidth to free physical memory. The most effective solution we have found is to watch for early signs of thrashing (such as high page fault rates) and to use these signs to indicate maximum capacity.

Unfortunately, thrashing is not always an easy thing to detect, since it is characterized by a decrease in productive work and an increase in system overhead. While we can measure overhead, productivity is inherently an application-specific notion. At present, we attempt to guess at throughput, using measures like the number of threads created and destroyed and the number of files opened and closed. Although this approach seems sufficient for applications such as Apache, more complicated applications might benefit from a threading API that allows them to explicitly inform the runtime system about their current productivity.

Application-specific resources also present some challenges. For example, application-level memory management hides resource allocation and deallocation from the runtime system. Additionally, applications may define other logical resources such as locks. Once again, providing an API through which the application can inform the runtime system about its logical resources may be a reasonable solution. For simple cases like memory allocators, it may also be possible to achieve this goal with the help of the compiler.

4.3 Yield Profiling

One problem that arises with cooperative scheduling is that threads may not yield the processor, which can lead to unfairness or even starvation. These problems are mitigated to some extent by the fact that all of the threads are part of the same application and are therefore mutually trusting. Nonetheless, failure to yield is still a performance problem that matters.

Because we annotate the graph dynamically with the running time for each edge, it is trivial to find those edges that failed to yield: their running times are typically orders of magnitude larger than the average edge. Our implementation allows the system operator to see the full blocking graph including edge time frequencies and resources used, by sending a `USR2` signal to the running server process.

This tool is very valuable when porting legacy applications to Capriccio. For example, in porting Apache, we found many places that did not yield sufficiently often. This result is not surprising, since Apache expects to run with preemptive threads. For example, it turns out that the `close()` call, which closes a socket, can sometimes take 5ms even though the documentation insists that it returns immediately when nonblocking I/O is selected. To fix this problem, we insert additional yields in our system call library, before and after the actual call to `close()`. While this solution does not fix the problem in general, it does allow us to break the long edge into smaller pieces. A better solution (which we have not yet implemented) is to use multiple kernel threads for running user-level threads. This approach would allow the use of multiple processors, and it would hide latencies from occasional uncontrollable blocking operations such as `close()` calls or page fault handling.

5. EVALUATION

The microbenchmarks presented in Section 2.3 show that Capriccio has good I/O performance and excellent scalability. In this section, we evaluate Capriccio's performance more generally under a realistic web server workload. Real-world web workloads involve large numbers of potentially slow clients, which provide good tests of both Capriccio's scalability and scheduling. We discuss the overhead of Capriccio's resource-aware scheduler in this context, and then we discuss how this scheduler can achieve automatic admission control.

5.1 Web Server Performance

The server machine for our web benchmarks is a 4x500 MHz Pentium server with 2GB memory and a Intel e1000 Gigabit Ethernet card. The operating system is stock Linux 2.4.20. Unfortunately, we found that the development-series Linux kernel used in the microbenchmarks discussed earlier became unstable when placed under heavy load. Hence, this experiment does not take advantage of `epoll` or Linux AIO. Similarly, we were not able to compare Capriccio against NPTL for this workload. We leave these additional experiments for future work.

We generated client load with up to 16 similarly configured machines across a Gigabit switched network. Both Capriccio and Haboob perform non-blocking network I/O with the standard UNIX `poll()` system call and use a thread pool for disk I/O. Apache 2.0.44 (configured to use POSIX threads) uses a combination of spin-polling on individual file descriptors and standard blocking I/O calls.

The workload for this test consisted of requests for 3.2 GB of static file data with various file sizes. The request frequencies for each size and for each file were designed to match those of the SPECweb99 benchmark. The clients for this test repeatedly connect to the server and issue a series of five requests, separated by 20ms pauses. For each client load level we ran the test for 4 minutes and based our measurements on the middle two minutes.

We used the client program from the SEDA work [41] because this program was simpler to set up on our client machines and because it allowed us to disable the dynamic content tests, thus preventing external CGI programs from competing with the web server for resources.

We limited the cache sizes of Haboob and Knot to 200 MB in order to force a good deal of disk activity. We used a minimal configuration for Apache, disabling all dynamic modules and access permission checking. Hence, it performed essentially the same tasks as Haboob and Knot.

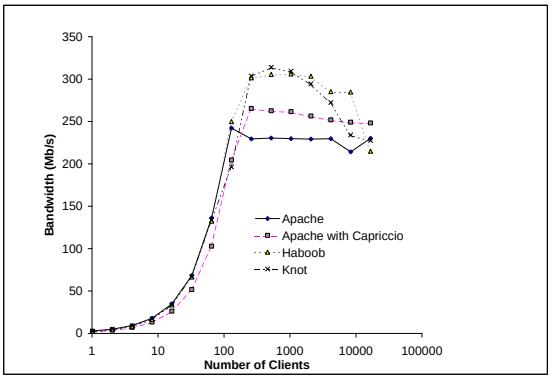


Figure 9: Web server bandwidth versus the number of simultaneous clients.

The performance results, shown in Figure 9 were quite encouraging. Apache's performance improved nearly 15% when run under Capriccio. Additionally, Knot's performance matched that of the event-based Haboob web server. While we do not have specific data on the variance of these results, it was quite small for load levels. There was more variation with more than 1024 clients, but the general trends were repeatable between runs.

Particularly remarkable is Knot's simplicity. Knot consists of 1290 lines of C code, written in a straightforward threaded style. Knot was very easy to write (it took one of us 3 days to create), and it is easy to understand. We consider this experience to be strong evidence for the simplicity of the threaded approach to writing highly concurrent applications.

5.2 Blocking Graph Statistics

Maintaining information about the resources used at each blocking point requires both determining where the program is when it blocks and performing some amount of computation to save and aggregate resource utilization figures.

Table 2 quantifies this overhead for Apache and Knot, for the workload described above. The top two lines show the average number of application cycles that each application spent going from one blocking point to the next. The bottom two lines show the number of cycles that Capriccio spends internally in order to maintain information used by the resource-aware scheduler. All cycle counts are the average number of cycles per blocking-graph edge during normal processing (i.e., under load and after the memory cache and branch predictors have warmed up).

It is important to note that these cycle counts include *only* the time spent in the application itself. Kernel time spent on I/O processing is not included. Since Internet applications are I/O intensive, much of their work actually takes place in the kernel. Hence, the performance impact of this overhead is lower than Table 2 would suggest.

The overhead of gathering and maintaining statistics is relatively small—less than 2% for edges in Apache. Moreover, these statistics tend to remain fairly steady in the workloads we have tested, so they can be sampled relatively infrequently. We have found a sampling ratio of 1/20 to be quite sufficient to maintain an accurate view of the system. This reduces the aggregate overhead to a mere 0.1%.

	Item	Cycles	Enabled
Apps	Apache	32697	n/a
	Knot	6868	n/a
System	stack trace	2447	Always for dynamic BG
	edge statistics	673	During sampling periods

Table 2: Average per-edge cycle counts for applications on Capriccio.

The overhead from stack traces is significantly higher, amounting to roughly 8% of the execution time for Apache and 36% for Knot. Additionally, since stack traces are essential for determining the location in the program, they must always be enabled.

The overhead from stack tracing illustrates how compiler integration could help to improve Capriccio's performance. The overhead to maintain location information in a statically generated blocking graph is essentially zero. Another more dynamic technique would be to maintain a global variable that holds a fingerprint of the current stack. This fingerprint can be updated at each function call by XOR'ing a unique function ID at each function's entry and exit point; these extra instructions can easily be inserted by the compiler. This fingerprint is not as accurate as a true stack trace, but it should be accurate enough to generate the same blocking graph that we currently use.

5.3 Resource-Aware Admission Control

To test our resource-aware admission control algorithms, we created a simple consumer-producer application. Producer threads loop, adding memory to a global pool and randomly touching pages to force them to stay in memory (or to cause VM faults for pages that have been swapped out). Consumer threads loop, removing memory from the global pool and freeing it.

This benchmark tests a number of system resources. First, if the producers allocate memory too quickly, the program may run out of virtual address space. Additionally, if page touching proceeds too quickly, the machine will thrash as the virtual memory system sends pages to and from disk. The goal, then, is to maximize the task throughput (measured by number of producer loops per second) while also making the best use of both memory and disk resources.

At run time, the test application is parameterized by the number of consumers and producers. Running under LinuxThreads, if there are more producers than consumers (and often when there are fewer) the system quickly starts to thrash. Under Capriccio, however, the resource-aware scheduler quickly detects the overload conditions and limits the number of producer threads from running. Thus, applications can reach a steady state near the knee of the performance curve.

6. RELATED WORK

Programming Models for High Concurrency

There has been a long-standing debate in the research community about the best programming model for high-concurrency; this debate has often focused on threads and events in particular. Ousterhout [28] enumerated a number

of potential advantages for events. Similarly, recent work on scalable servers advocates the use of events. Examples include Internet servers such as Flash [29] and Harvest [10] and server infrastructures like SEDA [41] and Ninja [38].

In the tradition of the duality argument developed by Lauer and Needham [21], we have previously argued that any apparent advantages of events are simply artifacts of poor thread implementations [39]. Hence, we believe past arguments in favor of events are better viewed as arguments for application-specific optimization and the need for efficient thread runtimes. Both of these arguments are major motivations for Capriccio. Moreover, the blocking graph used by Capriccio's scheduler was directly inspired by SEDA's stages and explicit queues.

In previous work [39], we also presented a number of reasons that threads should be preferred over events for highly concurrent programming. This paper provides additional evidence for that claim by demonstrating Capriccio's performance, scalability, and ability to perform application-specific optimization.

Adya et al. [1] pointed out that the debate between event-driven and threaded programming can actually be split into two debates: one between preemptive and cooperative task management, and one between automatic and manual stack management. They coin the term "stack ripping" to describe the process of manually saving and restoring live state across blocking points, and they identify this process as the primary drawback to manual stack management. The authors also point out the advantages of the cooperative threading approach.

Many authors have attempted to improve threading performance by transforming threaded code to event based code. For example, Adya et al. [1] automate the process of "stack-ripping" in event-driven systems, allowing code to be written in a more thread-like style. In some sense, though, all thread packages perform this same translation at run time, by mapping blocking operations into non-blocking state machines underneath. Ultimately, we believe there is no advantage to a static transformation from threaded code to event-driven code, because a well-tuned thread runtime can perform just as well as an event-based one. Our performance tests with Capriccio corroborate this claim.

User-Level Threads

There have been many user-level thread packages, but they differ from Capriccio in their goals and techniques. To the best of our knowledge, Capriccio is unique in its use of the blocking graph to provide resource-aware scheduling and in its use of compile-time analysis to effect application-specific optimizations. Additionally, we are not aware of any language-independent threading library that uses linked stack frames, though we discuss some language-dependent ones below.

Filaments [31] and NT's Fibers are two high-performance user-level thread packages. Both use cooperative scheduling, but they are not targeted at large numbers of blocking threads. Minimal Context-Switching Threads [19] is a high-performance thread package specialized for web caches that includes fast disk libraries and memory management. The performance optimizations employed by these packages would be useful for Capriccio as well; these are complementary to our work.

The State Threads package [37] is a lightweight cooperative threading system that shares Capriccio's goal of simplifying the programming model for network servers. Unlike Capriccio, the State Threads library does not provide a POSIX threading interface, so applications must be rewritten to use it. Additionally, State Threads use either `select` or `poll` instead of the more scalable Linux `epoll`, and they use blocking disk I/O. These factors limit the scalability of State Threads for network-intensive workloads, and they restrict its concurrency for disk-intensive workloads. There are patches available to allow Apache to use State Threads [36], resulting in a performance increase. These patches include a number of other improvements to Apache, however, so it is impossible to tell how much of the improvement came from State Threads. Unfortunately, these patches are no longer maintained and do not compile cleanly, so we were unable to run direct comparisons against Capriccio.

Scheduler activations [2] solve the problem of blocking I/O and unexpected blocking/preemption of user-level threads by adding kernel support for notifying the user-level scheduler of these events. This approach ensures clean integration of the thread library and the operating system; however, the large amount of kernel changes involved seem to have precluded wide adoption. Another potential problem with this approach is that there will be one scheduler activation for each outstanding I/O operation, which can number in the tens of thousands for Internet servers. This result is contrary to the original goal of reducing the number of kernel threads needed. This problem apparently stems from the fact that scheduler activations are developed primarily for high performance computing environments, where disk and fast network I/O are dominant. Nevertheless, scheduler activations can be a viable approach to dealing with page faults and preemptions in Capriccio. Employing scheduler activations would also allow the user-level scheduler to influence the kernel's decision about which kernel thread to preempt. This scheme can be used to solve difficult problems like *priority inversion* and the *convoy phenomenon* [6].

Support for user-level preemption and M:N threading (i.e., running M user-level threads on top of N kernel threads) is tricky. Techniques such as optimistic concurrency control and Cilk's work-stealing [7] can be used effectively to manage thread and scheduler data structures. Cordina presents a nice description of these and other techniques in the context of Linux [12]. We expect to employ many of these techniques in Capriccio when we add support for M:N threading.

Kernel Threads

The NPTL project for Linux has made great strides toward improving the efficiency of Linux kernel threads. These advances include a number of kernel-level improvements such as better data structures, lower memory overhead, and the use of $O(1)$ thread management operations. NPTL is quite new and is still under active development. Hence, we expect that some of the performance degradation we found with higher numbers of threads may be resolved as the developers find bugs and create faster algorithms.

Application-Specific Optimization

Performance optimization through application-specific control of system resources is an important theme in OS research. Mach [24] allowed applications to specify their own VM paging scheme, which improved performance for applications that knew about their upcoming memory needs and disk access patterns. UNET [40] did similar things

for network I/O, improving flexibility and reducing overhead without compromising safety. The SPIN operating system [5] and the VINO operating system [32] provide user customization by allowing application code to be moved into the kernel. The Exokernel [15] took the opposite approach and moved most of the OS to user level. All of these systems allow application-specific optimization of nearly all aspects of the system.

These techniques require programmers to tailor their application to manage resources for itself; this type of tuning is often difficult and brittle. Additionally, they tie programs to nonstandard APIs, reducing their portability. Capriccio takes a new approach to application-specific optimization by enabling automatic compiler-directed and feedback-based tuning of the thread package. We believe that this approach will make these techniques more practical and will allow a wider range of applications to benefit from them.

Asynchronous I/O

A number of authors propose improved kernel interfaces that could have an important impact on user-level threading. Asynchronous I/O primitives such as Linux's `epoll` [23], disk AIO [20] and FreeBSD's kqueue interface [22] are central to creating a scalable user-level thread package. Capriccio takes advantage of these interfaces and would benefit from improvements such as reducing the number of kernel crossings.

Stack Management

There are a number of related approaches to the problem of preallocating large stacks. Some functional languages, such as Standard ML of New Jersey [3], do not use a call stack at all; rather, they allocate all activation records on the heap. This approach is reasonable in the context of a language that uses a garbage collector and that supports higher-order functions and first-class continuations [4]. However, these features are not provided by the C programming language, which means that many of the arguments in favor of heap-allocated activation records do not apply in our case. Furthermore, we do not wish to incur the overhead associated with adding a garbage collector to our system; previous work has shown that Java's general-purpose garbage collector is inappropriate for high-performance systems [33].

A number of other systems have used lists of small stack chunks in place of contiguous stacks. Bobrow and Wegbreit describe a technique that uses a single stack for multiple environments, effectively dividing the stack into substacks [8]; however, they do not analyze the program to attempt to reduce the amount of run-time checks required. Olden, which is a language and runtime system for parallelizing programs, used a simplified version of Bobrow and Wegbreit's technique called "spaghetti stacks" [9]. In this technique, activation records for different threads are interleaved on a single stack; however, dead activation records in the middle of the stack cannot be reclaimed if live activation records still exist further down the stack, which can allow the amount of wasted stack space to grow without bound. More recently, the Lazy Threads project introduced stacklets, which are linked stack chunks for use in compiling parallel languages [18]. This mechanism provides run-time stack overflow checks, and it uses a compiler analysis to eliminate checks when stack usage can be bounded; however, this analysis that does not handle recursion as Capriccio does, and it does not provide tuning parameters. Cheng and Blelloch also used fixed-size stacklets to provide bounds on processing time in a parallel, real-time garbage collector [11].

Draves et al. [14] show how to reduce stack waste for kernel threads by using continuations. In this case, they have eliminated stacks entirely by allowing kernel threads to package their state in a continuation. In some sense, this approach is similar to the event-driven model, where programmers use "stack ripping" [1] to package live state before unwinding the stack. In the Internet servers we are considering, though, this approach is impractical, because the relatively large amount of state that must be saved and restored makes this process tedious and error-prone.

Resource-Aware Scheduling

Others have previously suggested techniques that are similar to our resource-aware scheduler. Douceur and Bolosky [13] describe a system that monitors the progress of running applications (as indicated by the application through a special API) and suspends low-priority processes when it detects thrashing. Their technique is deliberately unaware of specific resources and hence cannot be used with as much selectivity as ours.

Fowler et al. [16] propose a technique that is closer to ours, in that they directly examine low-level statistics provided by the operating system or through hardware performance counters. They show how this approach can be used at the application level to achieve adaptive admission control, and they suggest that the kernel scheduler might use this information as well. Their technique views applications as monolithic, however, so it is unclear how the kernel scheduler could do anything other than suspend resource intensive processes, as in [13]. Our blocking graph provides the additional information we believe the scheduler needs in order to make truly intelligent decisions about resources.

7. FUTURE WORK

We are in the process of extending Capriccio to work with multi-CPU machines. The fundamental challenge provided by multiple CPUs is that we can no longer rely on the cooperative threading model to provide atomicity. However, we believe that information produced by the compiler can assist the scheduler in making decisions that guarantee atomicity of certain blocks of code at the application level.

There are a number of aspects of Capriccio's implementation we would like to explore. We believe we could dramatically reduce kernel crossings under heavy network load with a batching interface for asynchronous network I/O. We also expect there are many ways to improve our resource-aware scheduler, such as tracking the variance in the resource usage of blocking graph nodes and improving our detection of thrashing.

There are several ways in which our stack analysis can be improved. As mentioned earlier, we use a conservative approximation of the call graph in the presence of function pointers or other language features that require indirect calls (e.g., higher-order functions, virtual method dispatch, and exceptions). Improvements to this approximation could substantially improve our results. In particular, we plan to adapt the dataflow analysis of CCured [27] in order to disambiguate many of the function pointer call sites. When compiling other languages, we could start with similarly conservative call graphs and then employ existing control flow analyses (e.g., the 0CFA analyses [34] for functional

and object-oriented languages languages, or virtual function resolution analyses [30] for object-oriented languages).

In addition, we plan to produce profiling tools that can assist the programmer and the compiler in tuning Capriccio's stack parameters to the application's needs. In particular, we can record information about internal and external wasted space, and we can gather statistics about which function calls cause new stack chunks to be linked. By observing this information for a range of parameter values, we can automate parameter tuning. We can also suggest potential optimizations to the programmer by indicating which functions are most often responsible for increasing stack size and stack waste.

In general, we believe that compiler technology will play an important role in the evolution of the techniques described in this paper. For example, we are in the process of devising a compiler analysis that is capable of generating a blocking graph at compile time; these results will improve the efficiency of the runtime system (since no backtraces are required to generate the graph), and they will allow us to get atomicity for free by guaranteeing statically that certain critical sections do not contain blocking points. In addition, we plan to investigate strategies for inserting blocking points into the code at compile time in order to enforce fairness.

Compile-time analysis can also reduce the occurrence of bugs by warning the programmer about data races. Although static detection of race conditions is challenging, there has been recent progress due to compiler improvements and tractable whole-program analyses. In nesC [17], a language for networked sensors, there is support for atomic sections, and the compiler understands the concurrency model. It uses a mixture of I/O completions and run-to-completion threads, and the compiler uses a variation of a call graph that is similar to our blocking graph. The compiler ensures that atomic sections reside within one edge on that graph; in particular, calls within an atomic section cannot yield or block (even indirectly). This kind of support would be extremely powerful for authoring servers. Finally, we expect that atomic sections will also enable better scheduling and even deadlock detection.

8. CONCLUSIONS

The Capriccio thread package provides empirical evidence that fixing thread packages is a viable solution to the problem of building scalable, high-concurrency Internet servers. Our experience with writing such programs suggests that the threaded programming model is a more useful abstraction than the event-based model for writing, maintaining, and debugging these servers. By decoupling the thread implementation from the operating system itself, we can take advantage of new I/O mechanisms and compiler support. As a result, we can use techniques such as linked stacks and resource-aware scheduling, which allow us to achieve significant scalability and performance improvements when compared to existing thread-based or event-based systems.

As this technology matures, we expect even more of these techniques to be integrated with compiler technology. By writing programs in threaded style, programmers provide the compiler with more information about the high-level structure of the tasks that the server must perform. Using this information, we hope the compiler can expose even more opportunities for both static and dynamic performance tuning.

9. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 Usenix ATC*, June 2002.

[2] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[3] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, 1991.

[4] A. W. Appel and Z. Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, Jan 1996.

[5] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN - an extensible microkernel for application-specific operating system services. In *ACM SIGOPS European Workshop*, pages 68–71, 1994.

[6] M. W. Blasgen, J. Gray, M. F. Mitoma, and T. G. Price. The convoy phenomenon. *Operating Systems Review*, 13(2):20–25, 1979.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[8] D. G. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. *Communications of the ACM*, 16(10):591–603, Oct 1973.

[9] M. C. Carlisle, A. Rogers, J. Reppy, and L. Hendren. Early experiences with Olden. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing (LNCS)*, 1993.

[10] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.

[11] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, 2001.

[12] J. Cordina. Fast multithreading on shared memory multiprocessors. Technical report, University of Malta, June 2000.

[13] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Symposium on Operating Systems Principles*, pages 247–260, 1999.

[14] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the13th ACM Symposium on Operating Systems Principle*, pages 122–136. Association for Computing Machinery SIGOPS, 1991.

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for

application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[16] R. Fowler, A. Cox, S. Elnikety, and W. Zwaenepoel. Using performance reflection in systems software. In *Proceedings of the 2003 HotOS Workshop*, May 2003.

[17] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[18] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads, Stacklets, and Synchronizers: Enabling primitives for compiling parallel languages. In *Third Workshop on Langauges, Compilers, and Run-Time Systems for Scalable Computers*, 1995.

[19] T. Hun. Minimal Context Thread 0.7 manual. http://www.aranetwork.com/docs/mct-manual.pdf, 2002.

[20] B. LaHaise. Linux AIO home page. http://www.kvack.org/ blah/aio/.

[21] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Second Inernational Symposium on Operating Systems, IR1A*, October 1978.

[22] J. Lemon. Kqueue: A generic and scalable event notification facility. In *USENIX Technical conference*, 2001.

[23] D. Libenzi. Linux epoll patch. http://www.xmailserver.org/ linux-patches/nio-improve.html.

[24] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. Technical Report TR-90-09-05, University of Washington, 1990.

[25] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.

[26] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, 2304:213–229, 2002.

[27] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *The 29th Annual ACM Symposium on Principles of Programming Languages*, pages 128–139. ACM, Jan. 2002.

[28] J. K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[29] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[30] H. D. Pande and B. G. Ryder. Data-flow-based virtual function resolution. *Lecture Notes in Computer Science*, 1145:238–254, 1996.

[31] W. Pang and S. D. Goodwin. An algorithm for solving constraint-satisfaction problems.

[32] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[33] M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein. Java support for data-intensive systems: Experiences building the Telegraph dataflow system. *SIGMOD Record*, 30(4):103–114, 2001.

[34] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, May 1991.

[35] E. Toernig. Coroutine library source. http://www.goron.de/˜froese/coro/.

[36] Unknown. Accellerating Apache project. http://aap.sourceforge.net/.

[37] Unknown. State threads for Internet applications. http://state-threads.sourceforge.net/docs/st.html.

[38] J. R. von Behren, E. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, , and D. Culler. Ninja: A framework for network services. In *Proceedings of the 2002 Usenix Annual Technical Conference*, June 2002.

[39] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 2003 HotOS Workshop*, May 2003.

[40] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, Decemeber 1995.

[41] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

Concurrency Control Performance Modeling: Alternatives and Implications

RAKESH AGRAWAL
AT&T Bell Laboratories
MICHAEL J. CAREY and MIRON LIVNY
University of Wisconsin

A number of recent studies have examined the performance of concurrency control algorithms for database management systems. The results reported to date, rather than being definitive, have tended to be contradictory. In this paper, rather than presenting "yet another algorithm performance study," we critically investigate the assumptions made in the models used in past studies and their implications. We employ a fairly complete model of a database environment for studying the relative performance of three different approaches to the concurrency control problem under a variety of modeling assumptions. The three approaches studied represent different extremes in how transaction conflicts are dealt with, and the assumptions addressed pertain to the nature of the database system's resources, how transaction restarts are modeled, and the amount of information available to the concurrency control algorithm about transactions' reference strings. We show that differences in the underlying assumptions explain the seemingly contradictory performance results. We also address the question of how realistic the various assumptions are for actual database systems.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*transaction processing*; D.4.8 [**Operating Systems**]: Performance—*simulation, modeling and prediction*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Concurrency control

1. INTRODUCTION

Research in the area of concurrency control for database systems has led to the development of many concurrency control algorithms. Most of these algorithms are based on one of three basic mechanisms: *locking* [23, 31, 32, 44, 48], *timestamps* [8, 36, 52], and *optimistic* concurrency control (also called commit-time validation or certification) [5, 16, 17, 27]. Bernstein and Goodman [9, 10] survey many of

A preliminary version of this paper appeared as "Models for Studying Concurrency Control Performance: Alternatives and Implications," in *Proceedings of the International Conference on Management of Data* (Austin, Tx., May 28–30, 1985).
M. J. Carey and M. Livny were partially supported by the Wisconsin Alumni Research Foundation under National Science Foundation grant DCR-8402818 and an IBM Faculty Development Award.
Authors' addresses: R. Agrawal, AT&T Bell Laboratories, Murray Hill, NJ 07974; M. J. Carey and M. Livny, Computer Sciences Department, University of Wisconsin, Madison, WI 53706.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1987 ACM 0362-5915/87/1200-0609 $01.50

the algorithms that have been developed and describe how new algorithms may be created by combining the three basic mechanisms.

Given the ever-growing number of available concurrency control algorithms, considerable research has recently been devoted to evaluating the performance of concurrency control algorithms. The behavior of locking has been investigated using both simulation [6, 28, 29, 39–41, 47] and analytical models [22, 24, 26, 35, 37, 50, 51, 53]. A qualitative study that discussed performance issues for a number of distributed locking and timestamp algorithms was presented in [7], and an empirical comparison of several concurrency control schemes was given in [34]. Recently, the performance of different concurrency control mechanisms has been compared in a number of studies. The performance of locking was compared with the performance of basic timestamp ordering in [21] and with basic and multiversion timestamp ordering in [30]. The performance of several alternatives for handling deadlock in locking algorithms was studied in [6]. Results of experiments comparing locking to the optimistic method appeared in [42 and 43], and the performance of several variants of locking, basic timestamp ordering, and the optimistic method was compared in [12 and 15]. Finally, the performance of several integrated concurrency control and recovery algorithms was evaluated in [1 and 2].

These performance studies are informative, but the results that have emerged, instead of being definitive, have been very contradictory. For example, studies by Carey and Stonebraker [15] and Agrawal and DeWitt [2] suggest that an algorithm that uses blocking instead of restarts is preferable from a performance viewpoint, but studies by Tay [50, 51] and Balter et al. [6] suggest that restarts lead to better performance than blocking. Optimistic methods outperformed locking in [20], whereas the opposite results were reported in [2 and 15]. In this paper, rather than presenting "yet another algorithm performance study," we examine the reasons for these apparent contradictions, addressing the models used in past studies and their implications.

The research that led to the development of the many currently available concurrency control algorithms was guided by the notion of *serializability* as the correctness criteria for general-purpose concurrency control algorithms [11, 19, 33]. Transactions are typically viewed as sequences of read and write requests, and the interleaved sequence of read and write requests for a concurrent execution of transactions is called the execution *log*. Proving algorithm correctness then amounts to proving that any log that can be generated using a particular concurrency control algorithm is equivalent to some serial log (i.e., one in which all requests from each individual transaction are adjacent in the log). Algorithm correctness work has therefore been guided by the existence of this widely accepted standard approach based on logs and serializability. Algorithm performance work has not been so fortunate—no analogous standard performance model has been available to guide the work in this area. As we will see shortly, the result is that nearly every study has been based on its own unique set of assumptions regarding database system resources, transaction behavior, and other such issues.

In this paper, we begin by establishing a performance evaluation framework based on a fairly complete model of a database management system. Our model

captures the main elements of a database environment, including both *users* (i.e., terminals, the source of transactions) and *physical resources* for storing and processing the data (i.e., disks and CPUs), in addition to the characteristics of the workload and the database. On the basis of this framework, we then show that differences in assumptions explain the apparently contradictory performance results from previous studies. We examine the effects of alternative assumptions, and we briefly address the question of which alternatives seem most reasonable for use in studying the performance of database management systems.

In particular, we critically examine the common assumption of *infinite resources.* A number of studies (e.g., [20, 29, 30, 50, 51]) compare concurrency control algorithms under the assumption that transactions progress at a rate independent of the number of active transactions. In other words, they proceed in *parallel* rather than in an interleaved manner. This is only really possible in a system with enough resources so that transactions *never* have to wait before receiving CPU or I/O service—hence our choice of the phrase "infinite resources." We will investigate this assumption by performing studies with truly infinite resources, with multiple CPU-I/O devices, and with transactions that think while holding locks. The infinite resource case represents an "ideal" system, the multiple CPU-I/O device case models a class of multiprocessor database machines, and having transactions think while executing models an interactive workload.

In addition to these resource-related assumptions, we examine two modeling assumptions related to transaction behavior that have varied from study to study. In each case, we investigate how alternative assumptions affect the performance results. One of the additional assumptions that we address is the *fake restart* assumption, in which it is assumed that a restarted transaction is replaced by a new, independent transaction, rather than running the same transaction over again. This assumption is nearly always used in analytical models in order to make the modeling of restarts tractable. Another assumption that we examine has to do with *write-lock acquisition.* A number of studies that distinguish between read and write locks assume that read locks are set on read-only items and that write locks are set on the items to be updated when they are first read. In reality, however, transactions often acquire a read lock on an item, then examine the item, and only then request that the read lock be upgraded to a write lock—because a transaction must usually examine an item before deciding whether or not to update it [B. Lindsay, personal communication, 1984].

We examine three concurrency control algorithms in this study, two locking algorithms and an optimistic algorithm, which represent extremes as to when and how they detect and resolve conflicts. Section 2 describes our choice of concurrency control algorithms. We use a simulator based on a closed queuing model of a single-site database system for our performance studies. The structure and characteristics of our model are described in Section 3. Section 4 discusses the performance metrics and statistical methods used for the experiments, and it also discusses how a number of our parameter values were chosen. Section 5 presents the resource-related performance experiments and results. Section 6 presents the results of our examination of the other modeling assumptions

described above. Finally, in Section 7 we summarize the main conclusions of this study.

2. CONCURRENCY CONTROL STRATEGIES

A transaction T is a sequence of actions $\{a_1, a_2, \ldots, a_n\}$, where a_i is either read or write. Given a concurrent execution of transactions, action a_i of transaction T_i and action a_j of T_j *conflict* if they access the same object and either (1) a_i is read and a_j is write, or (2) a_i is write and a_j is read or write. The various concurrency control algorithms basically differ in the time when they *detect conflicts* and the way that they *resolve conflicts* [9]. For this study we have chosen to examine the following three concurrency control algorithms that represent extremes in conflict detection and resolution:

Blocking. Transactions set read locks on objects that they read, and these locks are later upgraded to write locks for objects that they also write. If a lock request is denied, the requesting transaction is blocked. A waits-for graph of transactions is maintained [23], and deadlock detection is performed each time a transaction blocks.[1] If a deadlock is discovered, the youngest transaction in the deadlock cycle is chosen as the victim and restarted. Dynamic two-phase locking [23] is an example of this strategy.

Immediate-Restart. As in the case of blocking, transactions read-lock the objects that they read, and they later upgrade these locks to write locks for objects that they also write. However, if a lock request is denied, the requesting transaction is aborted and restarted after a restart delay. The delay period, which should be on the order of the expected response time of a transaction, prevents the same conflict from occurring repeatedly. A concurrency control strategy similar to this one was considered in [50 and 51].

Optimistic. Transactions are allowed to execute unhindered and are validated only after they have reached their commit points. A transaction is restarted at its commit point if it finds that any object that it read has been written by another transaction that committed during its lifetime. The optimistic method proposed by Kung and Robinson [27] is based on this strategy.

These algorithms represent two extremes with respect to when conflicts are detected. The blocking and immediate-restart algorithms are based on dynamic locking, so conflicts are detected as they occur. The optimistic algorithm, on the other hand, does not detect conflicts until transaction-commit time. The three algorithms also represent two different extremes with respect to conflict resolution. The blocking algorithm blocks transactions to resolve conflicts, restarting them only when necessary because of a deadlock. The immediate-restart and optimistic algorithms always use restarts to resolve conflicts.

One final note in regard to the three algorithms: In the immediate-restart algorithm, a restarted transaction must be delayed for some time to allow the conflicting transaction to complete; otherwise, the same lock conflict will occur repeatedly. For the optimistic algorithm, it is unnecessary to delay the restarted

[1] Blocking's performance results would change very little if periodic deadlock detection were assumed instead [4].

transaction, since any detected conflict is with an already committed transaction. A restart delay is also unnecessary for the blocking algorithm, since the same deadlock cannot arise repeatedly.

3. PERFORMANCE MODEL

There are three main parts of a concurrency control performance model: a database system model, a user model, and a transaction model. The *database system model* captures the relevant characteristics of the system's hardware and software, including the physical resources (CPUs and disks) and their associated schedulers, the characteristics of the database (e.g., its size and granularity), the load control mechanism for controlling the number of active transactions, and the concurrency control algorithm itself. The *user model* captures the arrival process for users, assuming either an open system or else a closed system with terminals. Also included in the user model is the nature of users' transactions, since they may either be batch-style (noninteractive) or interactive in their behavior. Finally, the *transaction model* captures the reference behavior and processing requirements of the transactions in the workload. A transaction can be thought of as being described via two characteristic strings. There is a logical reference string, which contains concurrency control level read and write requests, and a physical reference string, which contains requests for accesses to physical items on disk and the associated CPU processing time for each item accessed. These strings are typically described in a probabilistic manner for a class of transactions. In addition, if there is more than one class of transactions in the workload, the transaction model must specify the mix of transaction classes. It is our view that a concurrency control performance model that fails to include any of these three major parts is in some sense incomplete. Although our model description is not broken down in exactly this way (it is more transaction-flow oriented), it should become clear to the reader that our model includes all three parts.

Central to our simulation model for studying concurrency control algorithm performance is the closed queuing model of a single-site database system shown in Figure 1. This model is an extended version of the model used in [12 and 15], which in turn had its origins in the model of [39, 40, and 41]. There are a fixed number of terminals from which transactions originate. There is a limit to the number of transactions allowed to be active at any time in the system, the multiprogramming level *mpl*. A transaction is considered active if it is either receiving service or waiting for service inside the database system. When a new transaction originates, if the system already has a full set of active transactions, it enters the *ready queue* where it waits for a currently active transaction to complete or abort. (Transactions in the ready queue are not considered active.) The transaction then enters the *cc queue* (concurrency control queue) and makes the first of its concurrency control requests. If the concurrency control request is granted, the transaction proceeds to the *object queue* and accesses its first object. If more than one object is to be accessed prior to the next concurrency control request, the transaction will cycle through this queue several times. When the next concurrency control request is required, the transaction reenters the concurrency control queue and makes the request. It is assumed for modeling

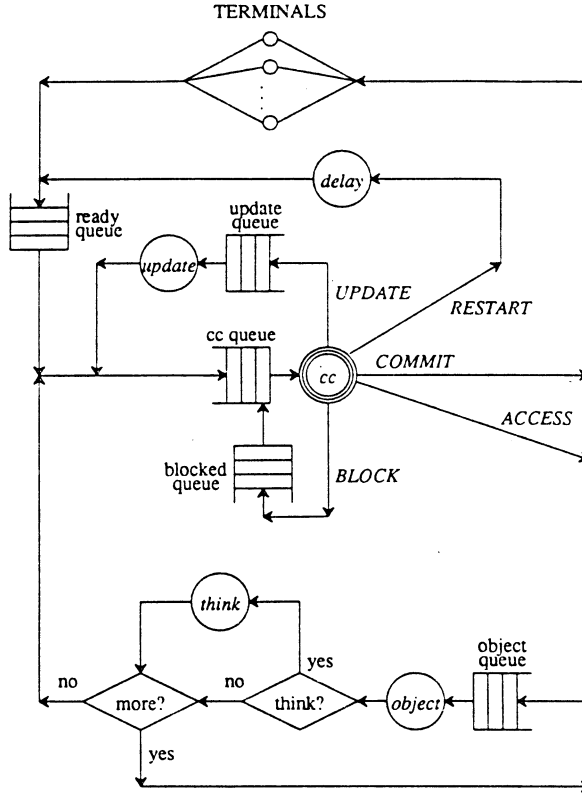


Fig. 1. Logical queuing model.

convenience that a transaction performs all of its reads before performing any writes. In one of the performance studies later in the paper, we examine the performance of concurrency control algorithms under interactive workloads. The think path in the model provides an optional random delay that follows object accesses for this purpose. More will be said about modeling interactive transactions shortly.

If the result of a concurrency control request is that the transaction must block, it enters the *blocked queue* until it is once again able to proceed. If a request leads to a decision to restart the transaction, it goes to the back of the ready queue, possibly after a randomly determined restart delay period of mean *restart_delay* (as in the immediate-restart algorithm). It then begins making all of the *same* concurrency control requests and object accesses over again.[2] Eventually the transaction may complete and the concurrency control algorithm may choose to commit the transaction. If the transaction is read-only, it is finished. If it has written one or more objects during its execution, however, it first enters the *update queue* and writes its deferred updates into the database. Deferred updates are assumed here because our modeling framework is intended to support *any* concurrency control algorithm—all algorithms operate correctly with

[2] The simulator maintains backup copies of transaction read and write sets.

deferred updates, but not all algorithms work with recovery schemes that allow in-place updates prior to transaction-commit time. (Examples of recovery schemes that use some form of deferred updates to minimize the cost of backing out aborted transactions include the Commercial INGRES recovery mechanism [45], the database cache of Elhard and Bayer [18], and the POSTGRES recovery mechanism [49]. Results on the performance of such recovery schemes and their alternatives may be found in [2 and 38]; a study of deferred versus in-place updates and the associated cost of backing out restarted transactions in the presence of limited buffer space is included in [2].)

To further illustrate the flow of transactions in the model, we briefly describe how the locking algorithms and the optimistic algorithm are modeled. For locking, each concurrency control request corresponds to a lock request for an object, and these requests alternate with object accesses. Locks are released together at end-of-transaction (after the deferred updates have been performed). Wait queues for locks and a waits-for graph are maintained by an algorithm-specific portion of the simulator. For optimistic concurrency control, the first concurrency control request is granted immediately (i.e., it is a "no-op"); all object accesses are then performed with no intervening concurrency control requests. Only after the last object access is finished does a transaction return to the concurrency control queue in the optimistic case, at which time its validation test is performed (followed, if successful, by its deferred updates).

Underlying the logical model of Figure 1 are two physical resources, the CPU and the I/O (i.e., disk) resources. Associated with the concurrency control, object access, and deferred update services in Figure 1 are some use of one or both of these two resources. The amounts of CPU and I/O time per logical service are specified as model parameters. The physical queuing model is depicted in Figure 2, and Table I summarizes the associated model parameters. As shown, the physical model is a collection of terminals, multiple CPU servers, and multiple I/O servers. The delay paths for the think and restart delays are also reflected in the physical queuing model. Model parameters specify the number of CPU servers, the number of I/O servers, and the number of terminals for the model. When a transaction needs CPU service, it is assigned a free CPU server; otherwise the transaction waits until one becomes free. Thus, the CPU servers may be thought of as being a pool of servers, all identical and serving one global CPU queue. Requests in the CPU queue are serviced FCFS (first-come, first-served), except that concurrency control requests have priority over all other service requests. Our I/O model is a probabilistic model of a database that is spread out across all of the disks. There is a queue associated with each of the I/O servers. When a transaction needs service, it chooses a disk (at random, with all disks being equally likely) and waits in an I/O queue associated with the selected disk. The service discipline for the I/O queues is also FCFS.

The parameters *obj_io* and *obj_cpu* are the amounts of I/O and CPU time associated with reading or writing an object. Reading an object takes resources equal to *obj_io* followed by *obj_cpu*. Writing an object takes resources equal to *obj_cpu* at the time of the write request and *obj_io* at deferred update time, since it is assumed that transactions maintain deferred update lists in buffers in main memory. These parameters represent constant service time requirements rather than stochastic ones for simplicity. The *ext_think_time* parameter is the mean

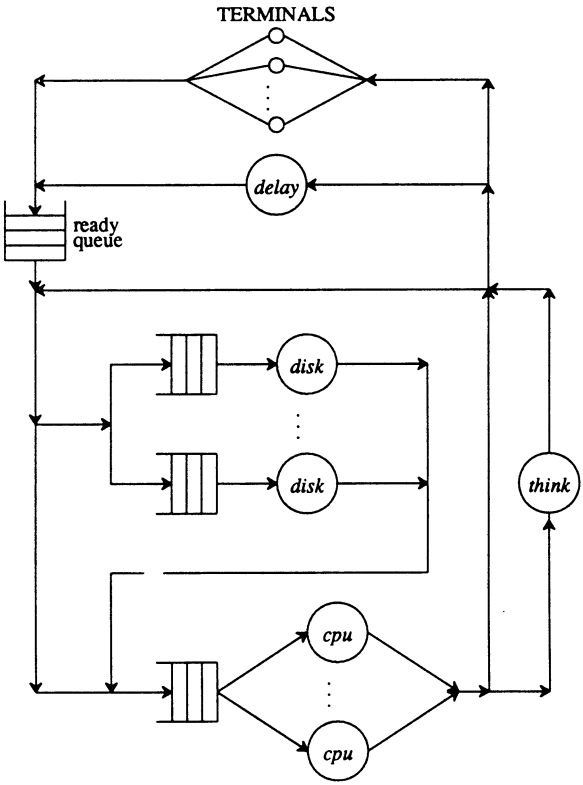


Fig. 2. Physical queuing model.

Table I. Model Parameters

Parameter	Meaning
db_size	Number of objects in database
tran_size	Mean size of transaction
max_size	Size of largest transaction
min_size	Size of smallest transaction
write_prob	Pr(write X | read X)
int_think_time	Mean intratransaction think time
restart_delay	Mean transaction restart delay
num_terms	Number of terminals
mpl	Multiprogramming level
ext_think_time	Mean time between transactions
obj_io	I/O time for accessing an object
obj_cpu	CPU time for accessing an object
num_cpus	Number of cpus
num_disks	Number of disks

time delay between the completion of a transaction and the initiation of a new transaction from a terminal, and the *int_think_time* parameter is the mean intratransaction think time (if any). We assume that both of these think times are exponentially distributed. To model interactive workloads, transactions can

be made to undergo a thinking period between finishing their reads and starting their writes.

A transaction is modeled according to the number of objects that it reads and writes. The parameter *tran_size* is the average number of objects read by a transaction, the mean of a uniform distribution between *min_size* and *max_size* (inclusive). These objects are randomly chosen (without replacement) from among all of the objects in the database. The probability that an object read by a transaction will also be written is determined by the parameter *write_prob*. The size of the database is assumed to be *db_size*.

The reader may have noted the absence of explicit concurrency control cost parameters. We assume for the purpose of this study that the cost of performing concurrency control operations is negligible compared to the cost of accessing objects. It has been shown elsewhere that the concurrency control request processing costs for algorithms based on locking and optimistic methods are roughly comparable [13], the main difference being the times at which these costs are incurred; it has also been argued that deadlock detection should not add significantly to the overhead of locking in a centralized database system [3, 13]. Thus, our negligible concurrency control cost assumption should not bias our results. Other sources of overhead that we do not consider in this study are the degradation of operating system and/or database system performance under high multiprogramming levels because of effects such as increased instruction path lengths due to large control structures (e.g., process tables), page thrashing due to memory limitations, increases in context switching, contention for high-traffic semaphores, and so forth. Database operating system issues such as context switching overhead and semaphore contention have been previously addressed by others [25].

4. GENERAL EXPERIMENT INFORMATION

The remainder of this paper presents results from a number of experiments designed to investigate the alternative modeling assumptions discussed in Section 1. Section 5 will describe studies with infinite resources, a small number of resources, various intermediate resource levels, and interactive transactions. Section 6 will describe studies of alternative modeling assumptions regarding restarted transactions and write-lock acquisition. First, however, this section of the paper will discuss the performance metrics and statistical methods used in the rest of the paper, and it will also explain how we chose many of the parameter settings used in the experiments reported here.

4.1 Performance Metrics

The primary performance metric used throughout the paper is the transaction throughput rate, which is the number of transactions completed per second. We employed a modified form of the batch means method [46] for the statistical data analyses of our throughput results, and each simulation was run for 20 batches with a large batch time to produce sufficiently tight 90 percent confidence intervals.[3] The actual batch time varied from experiment to experiment, but our

[3] More information on the details of the modified batch means method may be found in [12].

throughput confidence intervals were typically in the range of plus or minus a few percentage points of the mean value, more than sufficient for our purposes. We omit confidence interval information from our graphs for clarity, but we discuss only the statistically significant performance differences when summarizing our results. Response times, expressed in seconds, are also given in some cases. These are measured as the difference between when a terminal first submits a new transaction and when the transaction returns to the terminal following its successful completion, including any time spent waiting in the ready queue, time spent before (and while) being restarted, and so forth. The response time variance is also examined in a few cases.

Several additional performance-related metrics are used in analyzing the results of our experiments. In analyzing concurrency control activity for the three algorithms, two conflict-related metrics are employed. The first metric is the *blocking ratio*, which gives the average number of times that a transaction has to block per commit (computed as the ratio of the number of transaction-blocking events to the number of transaction commits). The other conflict-related metric is the *restart ratio*, which gives the average number of times that a transaction has to restart per commit (computed similarly). The last set of metrics used in our analysis are the total and useful disk utilizations. The total utilization for a disk gives the fraction of time during which it is busy. The useful utilization indicates the fraction of disk time used to do work that actually completed, excluding the fraction of time used for work that was later undone because of restarts. Note that since disks are uniformly selected when a request arrives, all of the disks have the same utilization over the range of our long simulation times. Disk utilization is used instead of CPU utilization because the disks turn out to be the bottleneck resource with our parameter settings (discussed next).

4.2 Parameter Settings

Table II gives the values of the simulation parameters that all of our experiments have in common (except where otherwise noted). Parameters that vary from experiment to experiment are not listed in Table II and will instead be given with the description of the relevant experiments.

The number of terminals is set to 200 in this study. The multiprogramming level, which limits the number of active transactions, is varied from 5 transactions up to the total number of terminals. This range was chosen for several reasons. First, it includes values that we consider to be reasonable for actual database systems. Second, varying the multiprogramming level in this way provides a wide range of operating conditions with respect to both data contention (conflict probabilities) and resource contention (waiting for CPUs and disks). The object processing costs were chosen on the basis of our notion of roughly what realistic values might be. In varying the number of CPUs and disks, an issue not addressed in Table II, we decided to use 1 CPU and 2 disks as 1 *resource unit*, and then vary the number of resource units assumed. In cases in which we have 1 CPU, we have 2 disks, and in cases in which we have N CPUs, we have $2N$ disks. This balance of CPUs and disks makes the utilization of these resources about equal (i.e., balanced) with our parameter values, as opposed to being either strongly CPU bound or strongly I/O bound; in particular, the system is just slightly I/O

Table II. Simulation Parameter Settings

Parameter	Value
db_size	1,000 pages
tran_size	8-page readset
max_size	12-page readset (maximum)
min_size	4-page readset (minimum)
write_prob	0.25
restart_delay	zero or adaptive (see text)
num_terms	200 terminals
mpl	5, 10, 25, 50, 75, 100, and 200
ext_think_time	1 second
obj_io	35 milliseconds
obj_cpu	15 milliseconds

bound. As for the *restart_delay* parameter, we mentioned in Section 2 that only the immediate-restart algorithm demands a restart delay. Thus, the delay is set to zero for the other two algorithms. For the immediate-restart algorithm we use an exponential delay with a mean equal to the running average of the transaction response time—that is, the duration of the delay is *adaptive*, depending on the observed average response time. We chose to employ an adaptive delay after performing a sensitivity analysis that showed us that the performance of the immediate-restart algorithm is sensitive to the restart delay time, particularly in the infinite resource case. Our preliminary experiments indicated that a delay of about one transaction time is best, and that throughput begins to drop off rapidly when the delay exceeds more than a few transaction times.

In choosing parameter values for our experiments, we wanted to choose database and transaction sizes[4] that would jointly yield a region of operation that would allow the interesting performance effects to be observed without requiring impossibly long simulation times. A preliminary experiment was conducted with an average transaction size of 8 reads and 2 writes, as shown in Table II, but with a database size of 10,000 pages. Due to the large database size and the relatively small transaction size, there were few conflicts in this experiment. The throughput results for a system with infinite resources and a system with limited resources (1 resource unit, meaning 1 CPU and 2 disks) are shown in Figures 3 and 4, respectively. The performance of the three concurrency control strategies was close in both cases, confirming the results reported in [1, 2, 12, and 15] and elsewhere. If conflicts are rare, it makes little difference which concurrency control algorithm is used. In both cases, blocking outperformed the other two algorithms by a small amount. Note also that the throughput curves reach a plateau at a multiprogramming level of 75 in Figure 3 (the infinite resource case). This is due to the fact that with 200 terminals, a 1 second think time, and an expected execution time of 500 milliseconds, increasing the allowed number of active transactions beyond 75 has no effect—all available transactions are already active, and the rest are in the think state. A plateau is reached earlier in Figure 4 (the limited resource case) because the resources are already saturated

[4] These sizes are expressed in pages, as we equate objects and pages in this study.

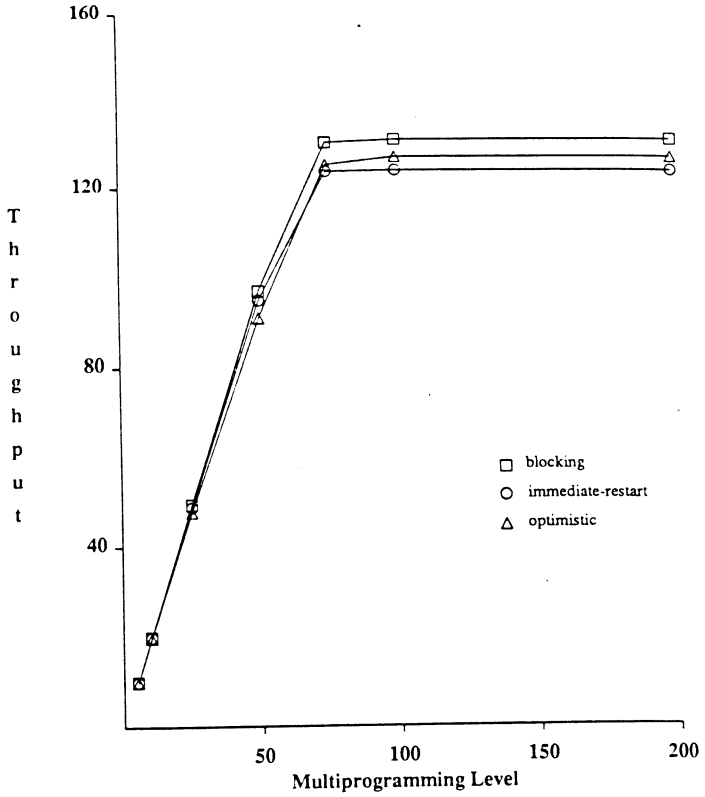


Fig. 3. Throughput (∞ resources).

with 25 concurrently active transactions. Since we are interested in investigating *differences* in concurrency control strategies, we decreased the database size to 1000 objects, as shown in Table II, to create a situation in which conflicts are more frequent. The remainder of our experiments were performed using this smaller database size.

Before closing this section, we should mention that there are a number of parameters that we could have varied but did not. For example, we could have varied the size of transactions, their distribution of sizes, or the granularity of the database; we could have varied the write probability for transactions; we could have investigated workloads containing several classes of transactions, and so forth. We also could have varied our notion of a resource unit, examining systems with less balanced resource utilization characteristics. For the purposes of this study, such variations were not of interest. Our goal was to see how certain basic assumptions affect the results of a concurrency control performance study, not to investigate exactly how performance varies with all possible sets of parameter values. Also, we have reported on the results of experiments with variations such as these elsewhere [12, 14, 15]. Our experience with variations of the first type is that for the most part they are just different ways of varying the probability of conflicts. Results regarding the relative performance of concurrency

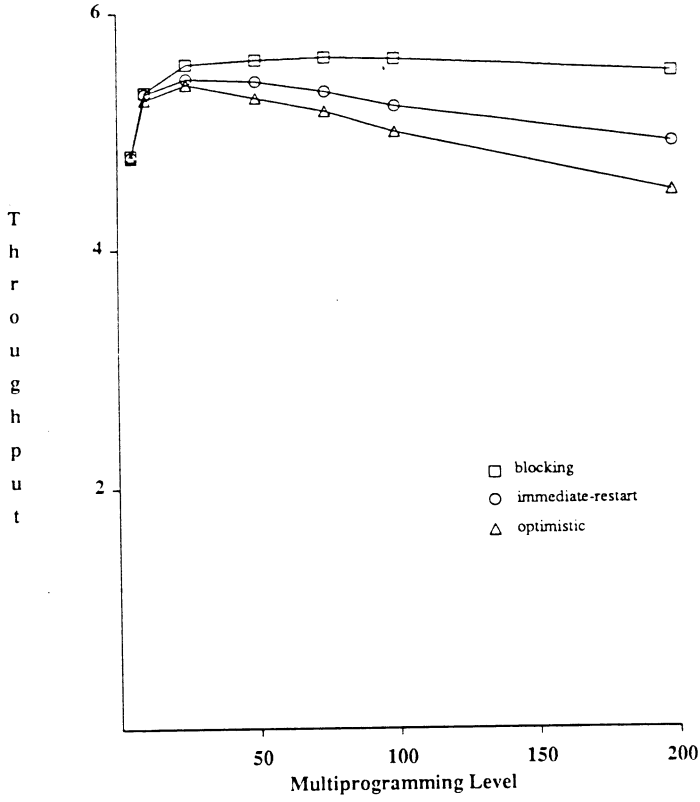


Fig. 4. Throughput (1 resource unit).

control algorithms appear to be insensitive to the particular manner in which this probability is varied, as long as it is indeed varied. (The only significant exception to this statement applies to workloads containing a wide range of transaction sizes or types, where algorithms sometimes display performance biases against a particular transaction class—for example, very large transactions can starve under the immediate-restart and optimistic algorithms, particularly with a workload containing a mix of short and long transactions—but we have studied this effect elsewhere as well [14].) Variations of the second type have been found to be of little or no interest in determining relative algorithm performance, since the primary factor determining the shape of the performance curves for an algorithm is the utilization of the bottleneck resource and not the fact that the bottleneck is the CPU or disk subsystem.

5. RESOURCE-RELATED ASSUMPTIONS

We performed several simulation experiments to study the implications of different resource-related assumptions on the performance of the three concurrency control algorithms described in Section 3. We investigated the performance of the three algorithms under the infinite resources assumption, with limited resources, and with varying numbers of CPUs and disks. To vary the number of

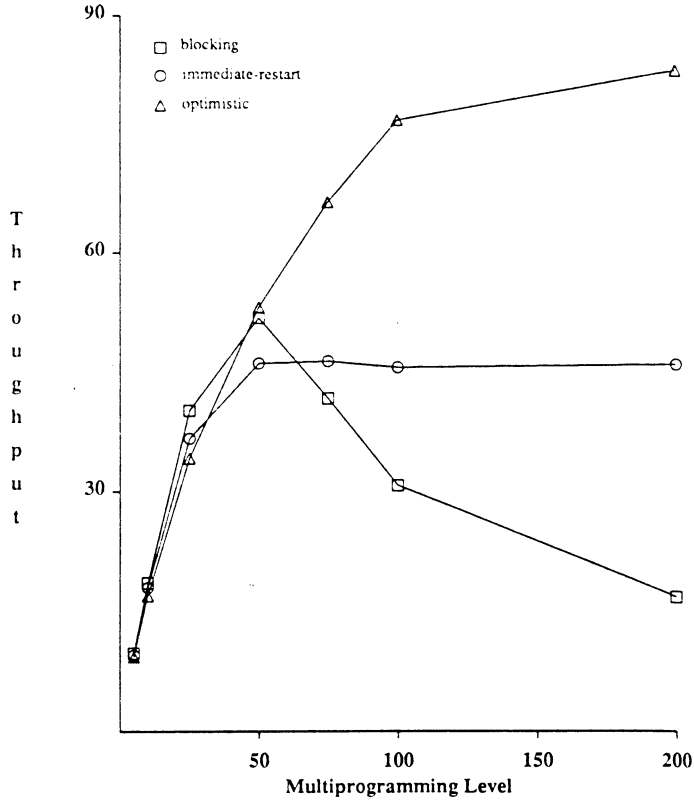


Fig. 5. Throughput (∞ resources).

CPUs and disks, we varied the number of resource units employed (each of which consists of 1 CPU and 2 disks). We also examined the case of an alternative workload type, an interactive workload.

5.1 Experiment 1: Infinite Resources

The first resource-related experiment examined the performance characteristics of the three strategies for a variety of multiprogramming levels, assuming infinite resources. With infinite resources, the throughput should be a nondecreasing function of multiprogramming level in the absence of data contention.[5] However, for a given size database, the probability of conflicts increases as the multiprogramming level increases. For blocking, the increased conflict probability manifests itself in the form of more blocking due to denial of lock requests and an increased number of restarts due to deadlocks. For the restart-oriented strategies, the higher probability of conflicts results in a larger number of restarts.

Figure 5 shows the throughput results for Experiment 1. Blocking starts thrashing as the multiprogramming level is increased beyond a certain level,

[5] We do not attempt to model system overhead phenomena such as the effects of increased operating system and database system control structure sizes on system path lengths.

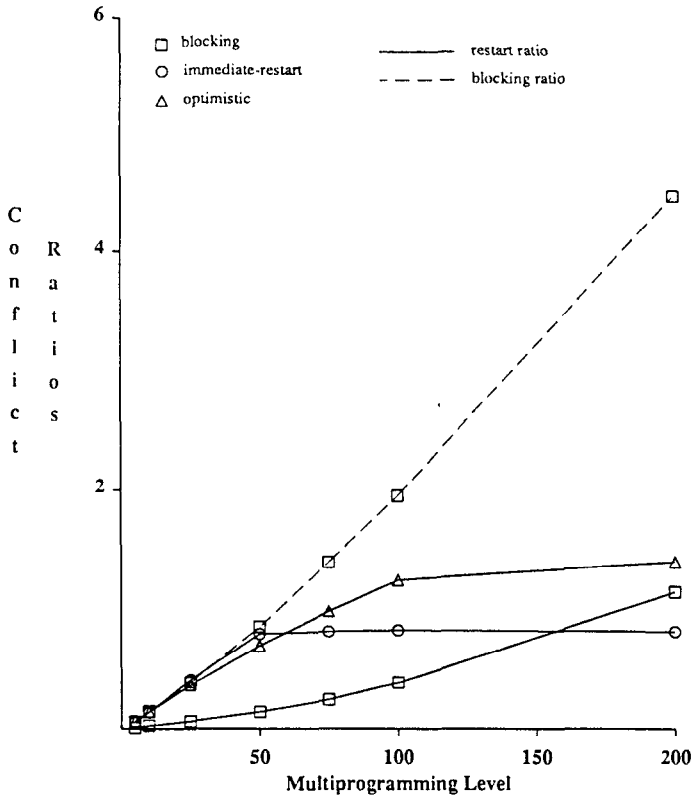


Fig. 6. Conflict ratios (∞ resources).

whereas the throughput keeps increasing for the optimistic algorithm. These results agree with predictions in [20] that were based on similar assumptions. Figure 6 shows the blocking and restart ratios for the three concurrency control algorithms. Note that the thrashing in blocking is due to the large increase in the number of times that a transaction is blocked, which reduces the number of transactions available to run and make forward progress, rather than to an increase in the number of restarts. This result is in agreement with the assertion in [6, 50 and 51] that under low resource contention and a high level of multiprogramming, blocking may start thrashing before restarts do. Although the restart ratio for the optimistic algorithm increases quickly with an increase in the multiprogramming level, new transactions start executing in place of the restarted ones, keeping the effective multiprogramming level high and thus entailing an increase in throughput.

Unlike the other two algorithms, the throughput of the immediate-restart algorithm reaches a plateau. This happens for the following reason: When a transaction is restarted in the immediate-restart strategy, a restart delay is invoked to allow the conflicting transaction to complete before the restarted transaction is placed back in the ready queue. As described in Section 4, the duration of the delay is *adaptive*, equal to the running average of the response

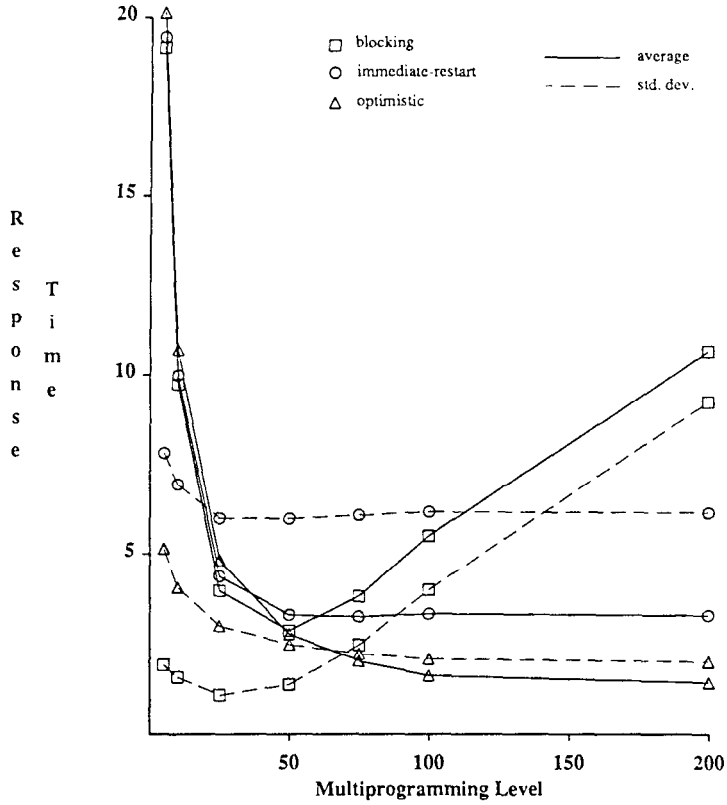


Fig. 7. Response time (∞ resources).

time. Because of this adaptive delay, the immediate-restart algorithm reaches a point beyond which all of the transactions that are not active are either in a restart delay state or else in a terminal thinking state (where a terminal is pausing between the completion of one transaction and submitting a new transaction). This point is reached when the number of active transactions in the system is such that a new transaction is basically sure to conflict with an active transaction and is therefore sure to be quickly restarted and then delayed. Such delays increase the average response time for transactions, which increases their average restart delay time; this has the effect of reducing the number of transactions competing for active status and in turn reduces the probability of conflicts. In other words, the adaptive restart delay creates a negative feedback loop (in the control system sense). Once the plateau is reached, there are simply no transactions waiting in the ready queue, and increasing the multiprogramming level is a "no-op" beyond this point. (Increasing the *allowed* number of active transactions cannot increase the *actual* number if none are waiting anyway.)

Figure 7 shows the mean response time (solid lines) and the standard deviation of response time (dotted lines) for each of the three algorithms. The response times are basically what one would expect, given the throughput results plus the fact that we have employed a closed queuing model. This figure does illustrate

one interesting phenomenon that occurred in nearly all of the experiments reported in this paper: The standard deviation of the response time is much smaller for blocking than for the immediate-restart algorithm over most of the multiprogramming levels explored, and it is also smaller than that of the optimistic algorithm for the lower multiprogramming levels (i.e., until blocking's performance begins to degrade significantly because of thrashing). The immediate-restart algorithm has a large response-time variance due to its restart delay. When a transaction has to be restarted because of a lock conflict during its execution, its response time is increased by a randomly chosen restart delay period with a mean of one entire response time, and in addition the transaction must be run all over again. Thus, a restart leads to a large response time increase for the restarted transaction. The optimistic algorithm restarts transactions at the end of their execution and requires restarted transactions to be run again from the beginning, but it does not add a restart delay to the time required to complete a transaction. The blocking algorithm restarts transactions much less often than the other algorithms for most multiprogramming levels, and it restarts them during their execution (rather than at the end) and without imposing a restart delay. Because of this, and because lock waiting times tend to be quite a bit smaller than the additional response time added by a restart, blocking has the lowest response time variance until it starts to thrash significantly. A high variance in response time is undesirable from a user's standpoint.

5.2 Experiment 2: Resource-Limited Situation

In Experiment 2 we analyzed the impact of limited resources on the performance characteristics of the three concurrency control algorithms. A database system with one resource unit (one CPU and two disks) was assumed for this experiment. The throughput results are presented in Figure 8.

Observe that for all three algorithms, the throughput curves indicate thrashing—as the multiprogramming level is increased, the throughput first increases, then reaches a peak, and then finally either decreases or remains roughly constant. In a system with limited CPU and I/O resources, the achievable throughput may be constrained by one or more of the following factors: It may be that not enough transactions are available to keep the system resources busy. Alternatively, it may be that enough transactions are available, but because of data contention, the "useful" number of transactions is less than what is required to keep the resources "usefully" busy. That is, transactions that are blocked due to lock conflicts are not useful. Similarly, the use of resources to process transactions that are later restarted is not useful. Finally, it may be that enough useful, nonconflicting transactions are available, but that the available resources are already saturated.

As the multiprogramming level was increased, the throughput first increased for all three concurrency control algorithms since there were not enough transactions to keep the resources utilized at low levels of multiprogramming. Figure 9 shows the total (solid lines) and useful (dotted lines) disk utilizations for this experiment. As one would expect, there is a direct correlation between the useful utilization curves of Figure 9 and the throughput curves of Figure 8. For blocking, the throughput peaks at mpl = 25, where the disks are being

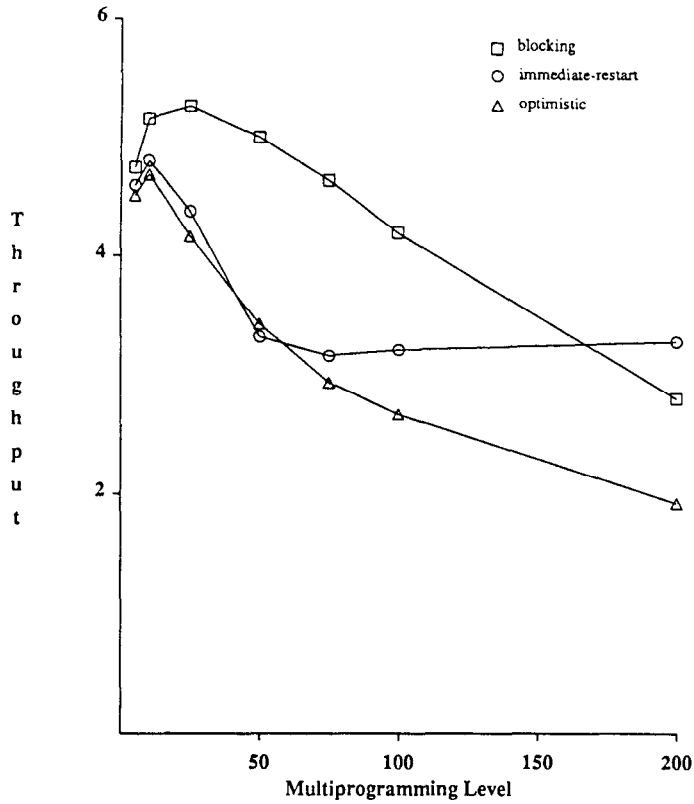


Fig. 8. Throughput (1 resource unit).

97 percent utilized, with a useful utilization of 92 percent.[6] Increasing the multiprogramming level further only increases data contention, and the throughput decreases as the amount of blocking and thus the number of deadlock-induced restarts increase rapidly. For the optimistic algorithm, the useful utilization of the disks peaks at mpl = 10, and the throughput decreases with an increase in the multiprogramming level because of the increase in the restart ratio. This increase in the restart ratio means that a larger fraction of the disk time is spent doing work that will be redone later. For the immediate-restart algorithm, the throughput also peaks at mpl = 10 and then decreases, remaining roughly constant beyond 50. The throughput remains constant for this algorithm for the same reason as described in the last experiment: Increasing the allowable number of transactions has no effect beyond 50, since all of the nonactive transactions are either in a restart delay state or thinking.

With regard to the throughput for the three strategies, several observations are in order. First, the maximum throughput (i.e., the best global throughput) was obtained with the blocking algorithm. Second, immediate-restart performed

[6] The actual throughput peak may of course be somewhere to the left or right of 25, in the 10–50 range, but that cannot be determined from our data.

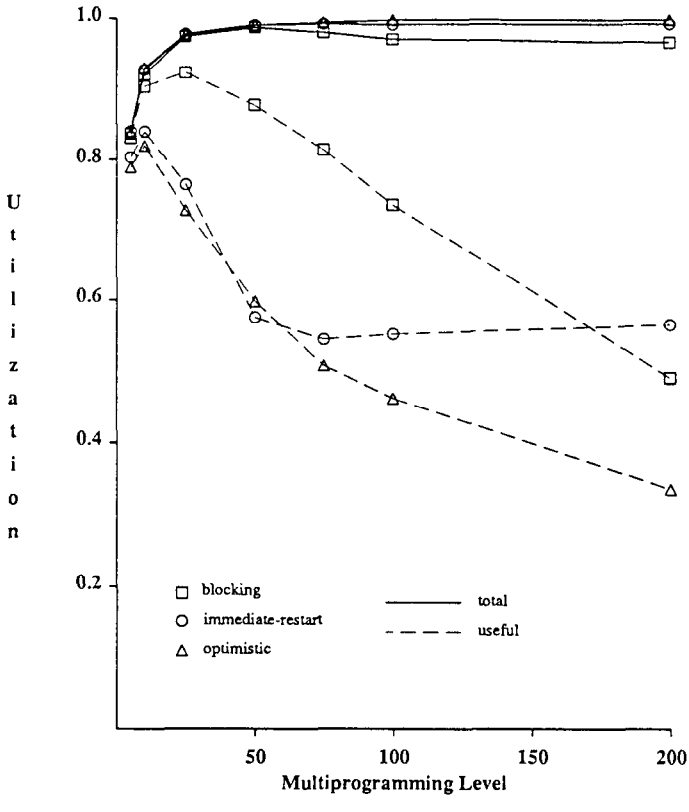


Fig. 9. Disk utilization (1 resource unit).

as well as or better than the optimistic algorithm. There were more restarts with the optimistic algorithm, and each restart was more expensive; this is reflected in the relative useful disk utilizations for the two strategies. Finally, the throughput achieved with the immediate-restart strategy for mpl = 200 was somewhat better than the throughput achieved with either blocking or the optimistic algorithm at this same multiprogramming level.

Figure 10 gives the average and the standard deviation of response time for the three algorithms in the limited resource case. The differences are even more noticeable than in the infinite resource case. Blocking has the lowest delay (fastest response time) over most of the multiprogramming levels. The immediate-restart algorithm is next, and the optimistic algorithm has the worst response time. As for the standard deviations, blocking is the best, immediate-restart is the worst, and the optimistic algorithm is in between the two. As in Experiment 1, the immediate-restart algorithm exhibits a high response time variance.

One of the points raised earlier merits further discussion. Should the performance of the immediate-restart algorithm at mpl = 200 lead us to conclude that immediate-restart is a better strategy at high levels of multiprogramming? We believe that the answer is no, for several reasons. First, the multiprogramming

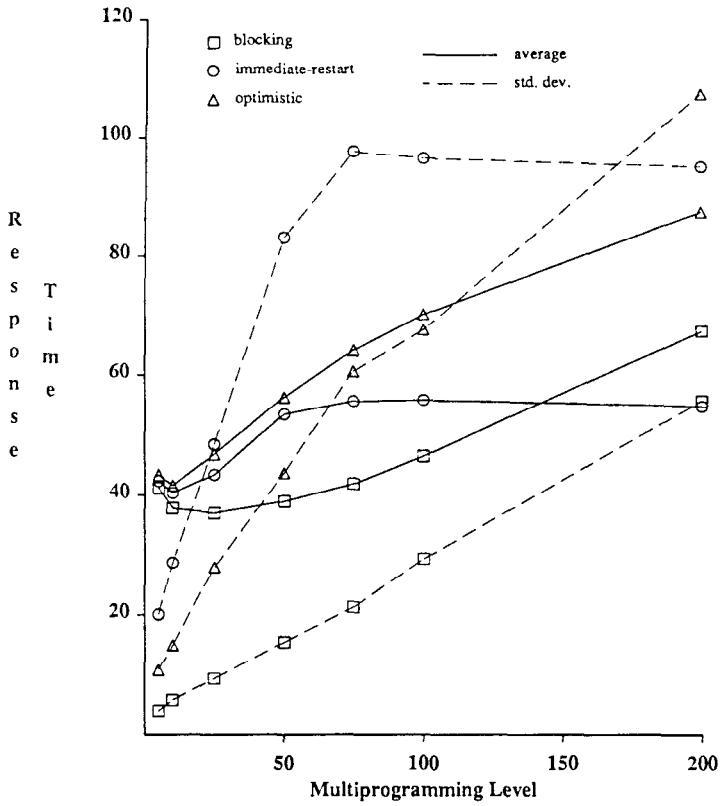


Fig. 10. Response time (1 resource unit).

level is internal to the database system, controlling the number of transactions that may concurrently compete for data and resources, and has nothing to do with the number of users that the database system may support; the latter is determined by the number of terminals. Thus, one should configure the system to keep multiprogramming at a level that gives the best performance. In this experiment, the highest throughput and smallest response time were achieved using the blocking algorithm at mpl = 25. Second, the restart delay in the immediate-restart strategy is there so that the conflicting transaction can complete before the restarted transaction is placed back into the ready queue. However, an unintended side effect of this restart delay in a system with a finite population of users is that it limits the actual multiprogramming level, and hence also limits the number of conflicts and resulting restarts due to reduced data contention. Although the multiprogramming level was increased to the total number of users (200), the actual average multiprogramming level never exceeded about 60. Thus, the restart delay provides a crude mechanism for limiting the multiprogramming level when restarts become overly frequent, and adding a restart delay to the other two algorithms should improve their performance at high levels of multiprogramming as well.

To verify this latter argument, we performed another experiment in which the adaptive restart delay was used for restarted transactions in both the blocking

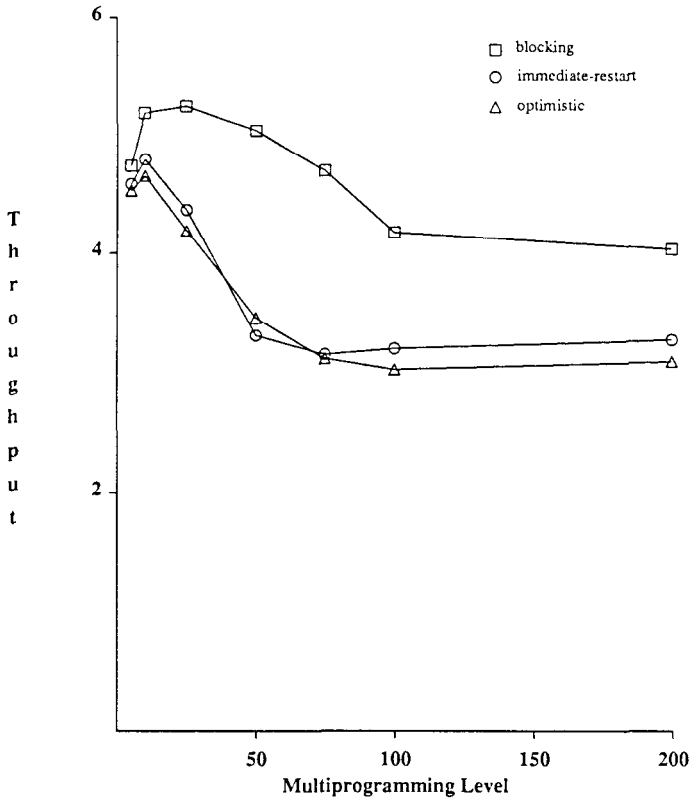


Fig. 11. Throughput (adaptive restart delays).

and optimistic algorithms as well. The throughput results that we obtained are shown in Figure 11. It can be seen that introducing an adaptive restart delay helped to limit the multiprogramming level for the blocking and optimistic algorithms under high conflicts, as it does for immediate-restart, reducing data contention at the upper range of multiprogramming levels. Blocking emerges as the clear winner, and the performance of the optimistic algorithm becomes comparable to the immediate-restart strategy. The one negative effect that we observed from adding this delay was an increase in the standard deviation of the response times for the blocking and optimistic algorithms. Since a restart delay only helps performance for high multiprogramming levels, it seems that a better strategy is to enforce a lower multiprogramming level limit to avoid thrashing due to high contention and to maintain a small standard deviation of response time.

5.3 A Brief Aside

Before discussing the remainder of the experiments, a brief aside is in order. Our concurrency control performance model includes a time delay, *ext_think_time*, between the completion of one transaction and the initiation of the next transaction from a terminal. Although we feel that such a time delay is necessary in a

realistic performance model, a side effect of the delay is that it can lead the database system to become "starved" for transactions when the multiprogramming level is increased beyond a certain point. That is, increasing the multiprogramming level has no effect on system throughput beyond this point because the actual number of active transactions does not change. This form of starvation can lead an otherwise increasing throughput to reach a plateau when viewed as a function of the multiprogramming level. In order to verify that our conclusions were not distorted by the inclusion of a think time, we repeated Experiments 1 and 2 with no think time (i.e., with *ext_think_time* = 0).

The throughput results for these experiments are shown in Figures 12 and 13, and the figures to which these results should be compared are Figures 5 and 8. It is clear from these figures that, although the exact performance numbers are somewhat different (because it is now never the case that the system is starved for transactions while one or more terminals is in a thinking state), the relative performance of the algorithms is not significantly affected. The explanations given earlier for the observed performance trends are almost all applicable here as well. In the infinite resource case (Figure 12), blocking begins thrashing beyond a certain point, and the immediate-restart algorithm reaches a plateau because of the large number of restarted transactions that are delaying (due to the restart delay) before running again. The only significant difference in the infinite resource performance trends is that the throughput of the optimistic algorithm continues to improve as the multiprogramming level is increased, instead of reaching a plateau as it did when terminals spent some time in a thinking state (and thus sometimes caused the actual number of transactions in the system to be less than that allowed by the multiprogramming level). Franaszek and Robinson predicted this [20], predicting logarithmically increasing throughput for the optimistic algorithm as the number of active transactions increases under the infinite resource assumption. Still, this result does not alter the general conclusions that were drawn from Figure 5 regarding the relative performance of the algorithms. In the limited resource case (Figure 13), the throughput for each of the algorithms peaks when resources become saturated, decreasing beyond this point as more and more resources are wasted because of restarts, just as it did before (Figure 8). Again, fewer and/or earlier restarts lead to better performance in the case of limited resources. On the basis of the lack of significant differences between the results obtained with and without the external think time, then, we can safely conclude that incorporating this delay in our model has not distorted our results. The remainder of the experiments in this paper will thus be run using a nonzero external think time (just like Experiments 1 and 2).

5.4 Experiment 3: Multiple Resources

In this experiment we moved the system from limited resources toward infinite resources, increasing the level of resources available to 5, 10, 25, and finally 50 resource units. This experiment was motivated by a desire to investigate performance trends as one moves from the limited resource situation of Experiment 2 toward the infinite resource situation of Experiment 1. Since the infinite resource assumption has sometimes been justified as a way of investigating what performance trends to expect in systems with many processors [20], we were interested

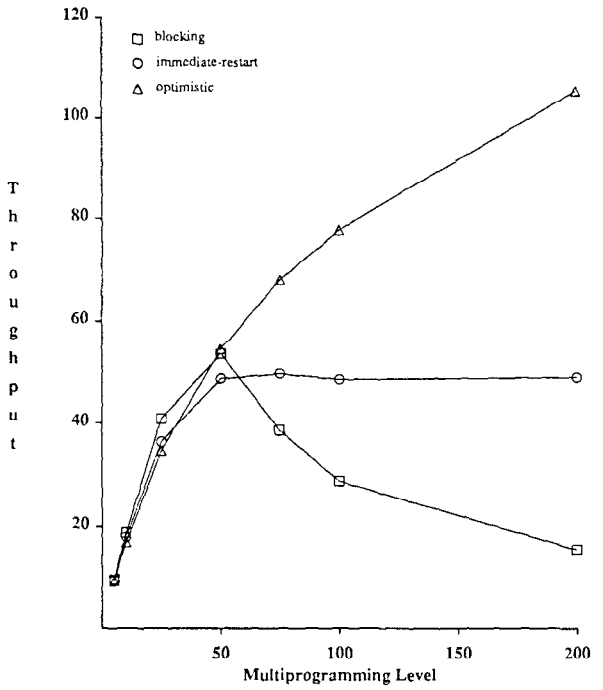


Fig. 12. Throughput (∞ resources, no external think time).

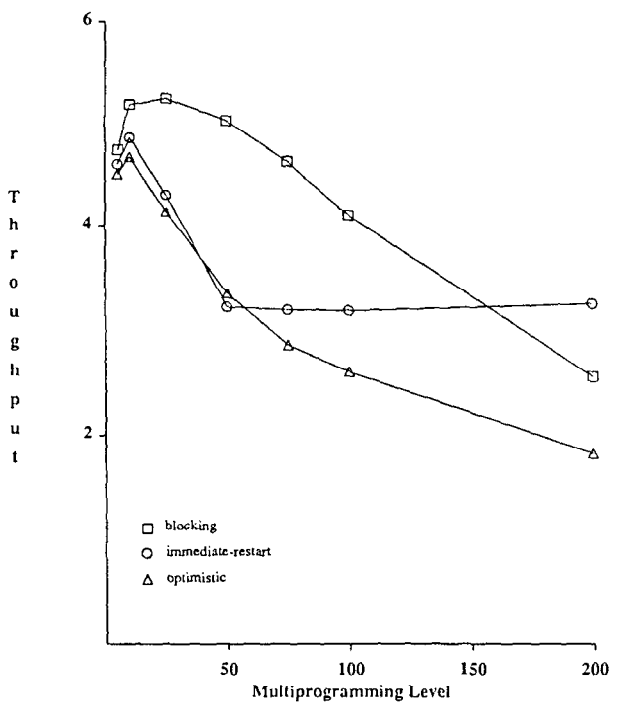


Fig. 13. Throughput (1 resource unit, no external think time).

in determining where (i.e., at what level of resources) the behavior of the system would begin to approach that of the infinite resource case in an environment such as a multiprocessor database machine.

For the cases with 5 and 10 resource units, the relative behavior of the three concurrency control strategies was fairly similar to the behavior in the case of just 1 resource unit. The throughput results for these two cases are shown in Figures 14 and 16, respectively, and the associated disk utilization figures are given in Figures 15 and 17. Blocking again provided the highest overall throughput. For large multiprogramming levels, however, the immediate-restart strategy provided better throughput than blocking (because of its restart delay), but not enough so as to beat the highest throughput provided by the blocking algorithm. With 5 resource units, where the maximum useful disk utilizations for blocking, immediate-restart, and the optimistic algorithm were 72, 60, and 58 percent, respectively, the results followed the same trends as those of Experiment 2. Quite similar trends were obtained with 10 resource units, where the maximum useful utilizations of the disks for blocking, immediate-restart, and optimistic were 56, 45, and 47 percent, respectively. Note that in all cases, the total disk utilizations for the restart-oriented algorithms are higher than those for the blocking algorithm because of restarts; this difference is partly due to wasted resources. By *wasted resources* here, we mean resources used to process objects that were later undone because of restarts—these resources are wasted in the sense that they were consumed, making them unavailable for other purposes such as background tasks.

With 25 resource units, the maximum throughput obtained with the optimistic algorithm beats the maximum throughput obtained with blocking (although not by very much). The throughput results for this case are shown in Figure 18, and the utilizations are given in Figure 19. The total and the useful disk utilizations for the maximum throughput point for blocking were 34 and 30 percent (respectively), whereas the corresponding numbers for the optimistic algorithm were 81 and 30 percent. Thus, the optimistic algorithm has become attractive because a large amount of otherwise unused resources are available, and thus the waste of resources due to restarts does not adversely affect performance. In other words, with useful utilizations in the 30 percent range, the system begins to behave somewhat like it has infinite resources. As the number of available resources is increased still further to 50 resource units, the results become very close indeed to those of the infinite resource case; this is illustrated by the throughput and utilizations shown in Figures 20 and 21. Here, with maximum useful utilizations down in the range of 15 to 25 percent, the shapes and relative positions of the throughput curves are very much like those of Figure 5 (although the actual throughput values here are still not quite as large).

Another interesting observation from these latter results is that, with blocking, resource utilization decreases as the level of multiprogramming increases and hence throughput decreases. This is a further indication that blocking may thrash due to waiting for locks before it thrashes due to the number of restarts [6, 50, 51], as we saw in the infinite resource case. On the other hand, with the optimistic algorithm, as the multiprogramming level increases, the total utilization of resources and resource waste increases, and the throughput decreases

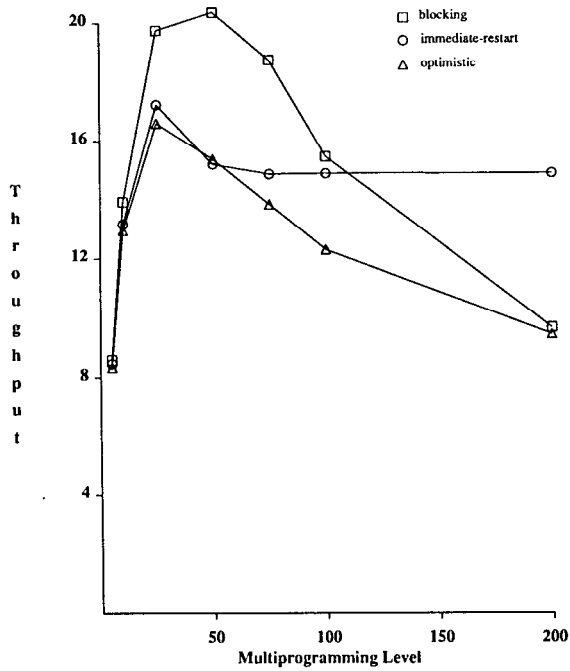


Fig. 14. Throughput (5 resource units).

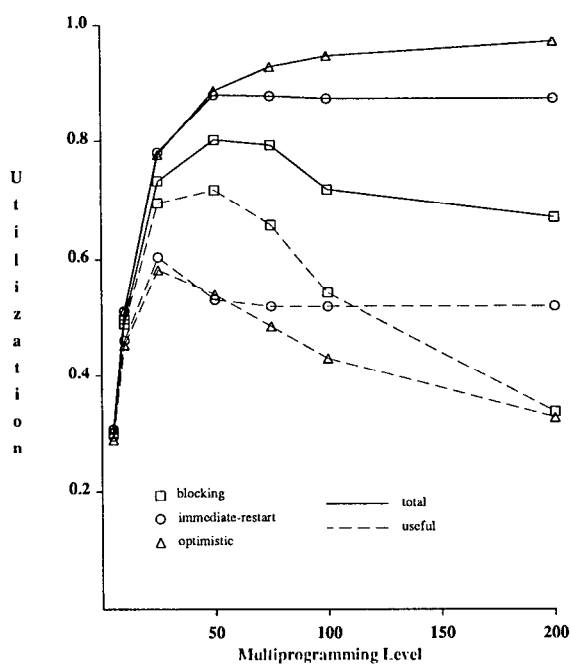


Fig. 15. Disk utilization (5 resource units).

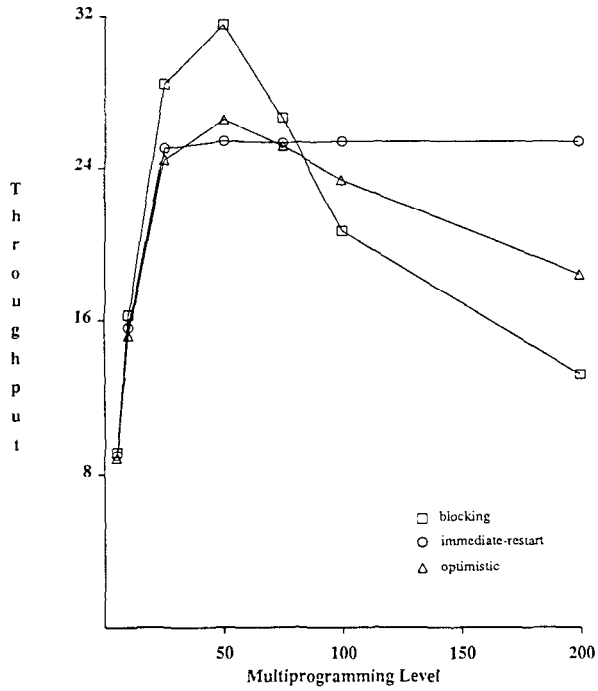


Fig. 16. Throughput (10 resource units).

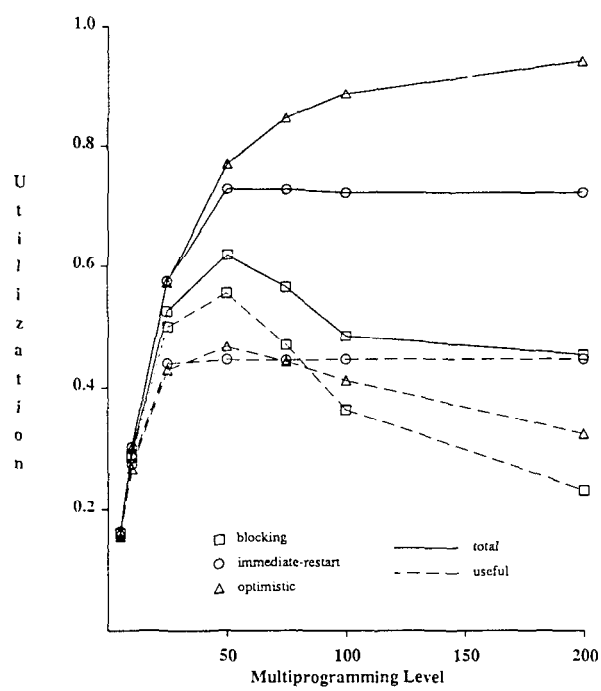


Fig. 17. Disk utilization (10 resource units).

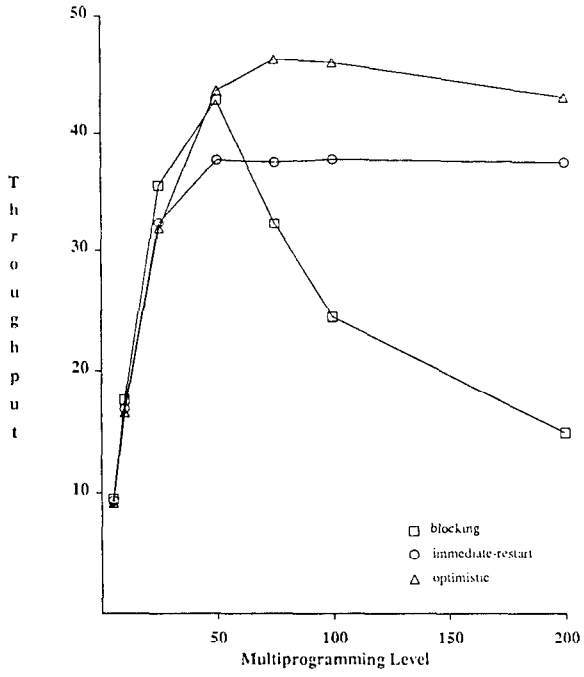


Fig. 18. Throughput (25 resource units).

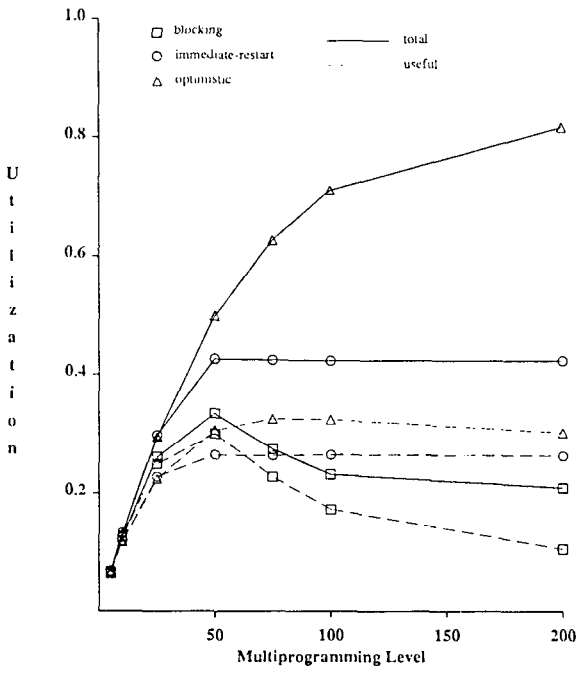


Fig. 19. Disk utilization (25 resource units).

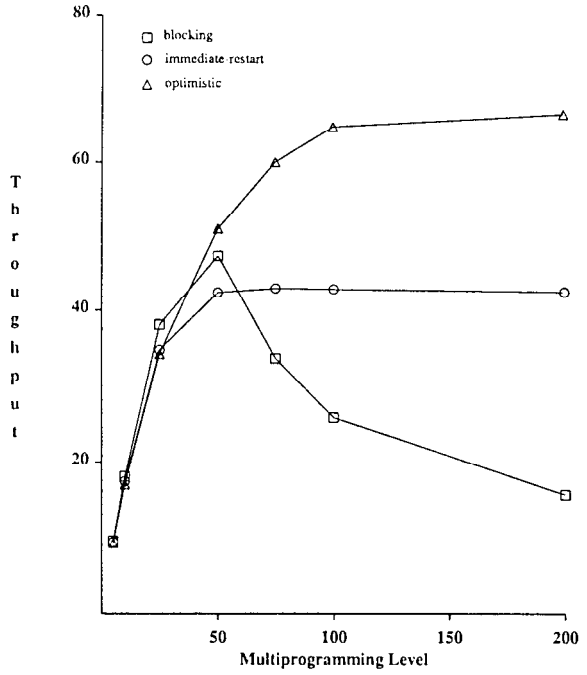


Fig. 20. Throughput (50 resource units).

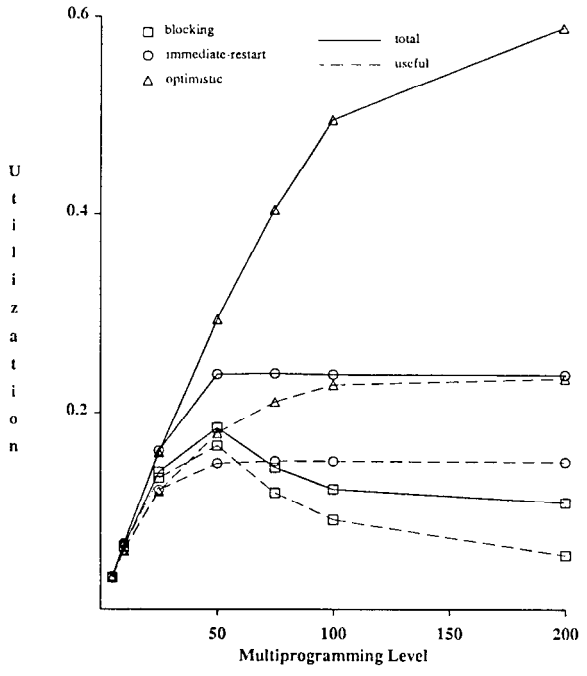


Fig. 21. Disk utilization (50 resource units).

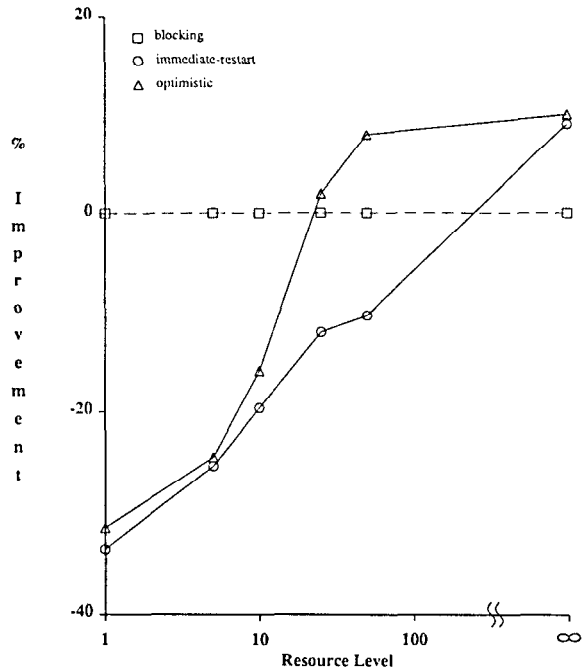


Fig. 22. Improvement over blocking (MPL = 50).

somewhat (except with 50 resource units). Thus, this strategy eventually thrashes because of the number of restarts (i.e., because of resources). With immediate-restart, as explained earlier, a plateau is reached for throughput and resource utilization because the actual multiprogramming level is limited by the restart delay under high data contention.

As a final illustration of how the level of available resources affects the choice of a concurrency control algorithm, we plotted in Figures 22 through 24 the percent throughput improvement of the algorithms with respect to that of the blocking algorithm as a function of the resource level. The resource level axis gives the number of resource units used, which ranges from 1 to infinity (the infinite resource case). Figure 22 shows that, for a multiprogramming level of 50, blocking is preferable with up to almost 25 resource units; beyond this point the optimistic algorithm is preferable. For a multiprogramming level of 100, as shown in Figure 23, the crossover point comes earlier because the throughput for blocking is well below its peak at this multiprogramming level. Figure 24 compares the maximum attainable throughput (over all multiprogramming levels) for each algorithm as a function of the resource level, in which case locking again wins out to nearly 25 resource units. (Recall that useful utilizations were down in the mid-20 percent range by the time this resource level, with 25 CPUs and 50 disks, was reached in our experiments.)

5.5 Experiment 4: Interactive Workloads

In our last resource-related experiment, we modeled interactive transactions that perform a number of reads, think for some period of time, and then perform their

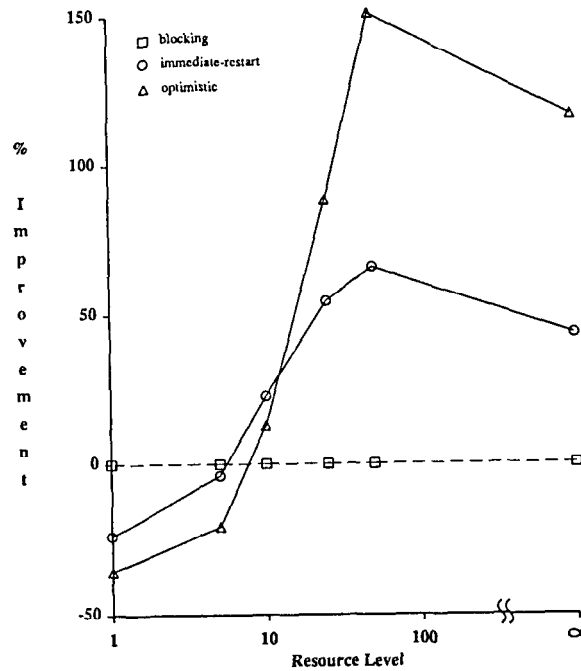


Fig. 23. Improvement over blocking (MPL = 100).

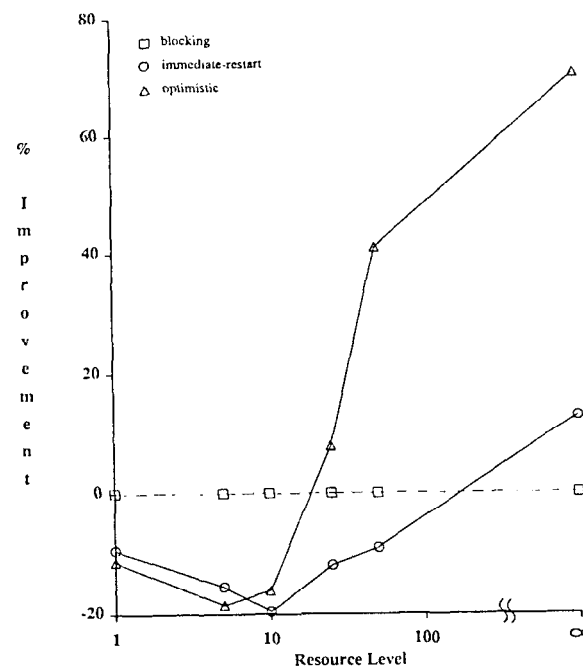


Fig. 24. Improvement over blocking (maximum).

writes. This model of interactive transactions was motivated by a large body of form-screen applications where data is put up on the screen, the user may change some of the fields after staring at the screen awhile, and then the user types "enter," causing the updates to be performed. The intent of this experiment was to find out whether large intratransaction (internal) think times would be another way to cause a system with limited resources to behave like it has infinite resources. Since Experiment 3 showed that low utilizations can lead to behavior similar to the infinite resource case, we suspected that we might indeed see such behavior here. The interactive workload experiment was performed for internal think times of 1, 5, and 10 seconds. At the same time, the external think times were increased to 3, 11, and 21 seconds, respectively, in order to maintain roughly the same ratio of idle terminals (those in an external thinking state) to active transactions. We have assumed a limited resource environment with 1 resource unit for the system in this experiment.

Figure pairs (25, 26), (27, 28), and (29, 30) show the throughput and disk utilizations obtained for the 1, 5, and 10 second intratransaction think time experiments, respectively. On the average, a transaction requires 150 milliseconds of CPU time and 350 milliseconds of disk time, so an internal think time of 5 seconds or more is an order of magnitude larger than the time spent consuming CPU or I/O resources. Even with many transactions in the system, resource contention is significantly reduced because of such think times, and the result is that the CPU and I/O resources behave more or less like infinite resources. Consequently, for large think times, the optimistic algorithm performs better than the blocking strategy (see Figures 27 and 29). For an internal think time of 10 seconds, the useful utilization of resources is much higher with the optimistic algorithm than the blocking strategy, and its highest throughput value is also considerably higher than that of blocking. For a 5-second internal think time, the throughput and the useful utilization with the optimistic algorithm are again better than those for blocking. For a 1-second internal think time, however, blocking performs better (see Figure 25). In this last case, in which the internal think time for transactions is closer to their processing time requirements, the resource utilizations are such that resources wasted because of restarts make the optimistic algorithm the loser.

The highest throughput obtained with the optimistic algorithm was consistently better than that for immediate-restart, although for higher levels of multiprogramming the throughput obtained with immediate-restart was better than the throughput obtained with the optimistic algorithm due to the *mpl*-limiting effect of immediate-restart's restart delay. As noted before, this high multiprogramming level difference could be reversed by adding a restart delay to the optimistic algorithm.

5.6 Resource-Related Conclusions

Reflecting on the results of the experiments reported in this section, several conclusions are clear. First, a blocking algorithm like dynamic two-phase locking is a better choice than a restart-oriented concurrency control algorithm like the immediate-restart or optimistic algorithms for systems with medium to high

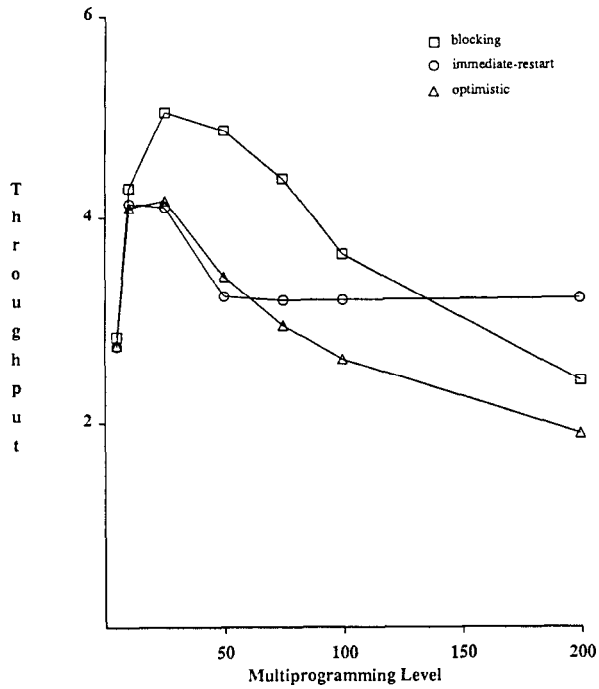


Fig. 25. Throughput (1 second thinking).

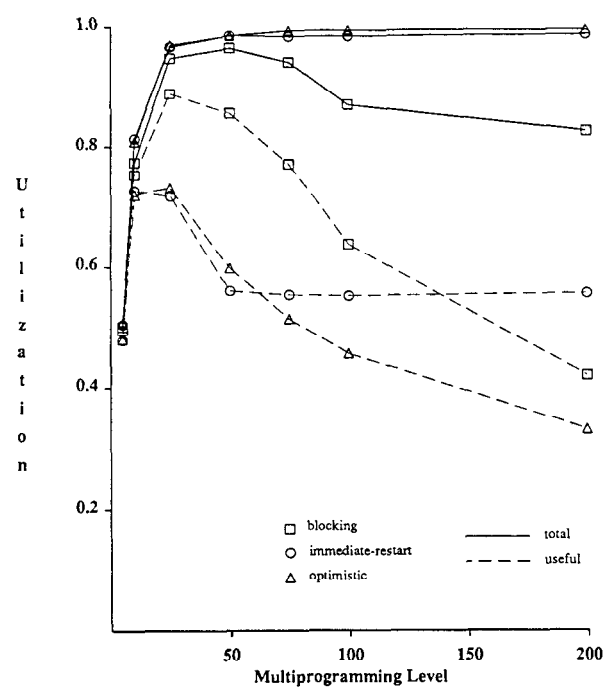


Fig. 26. Disk utilization (1 second thinking).

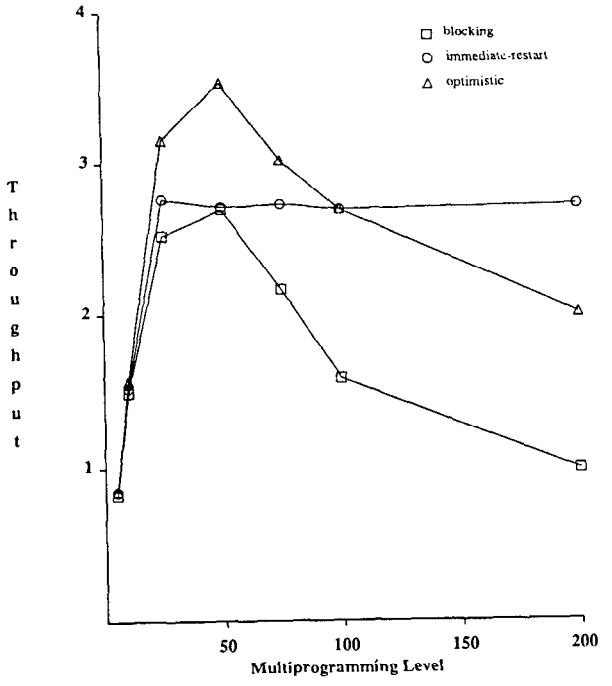


Fig. 27. Throughput (5 seconds thinking).

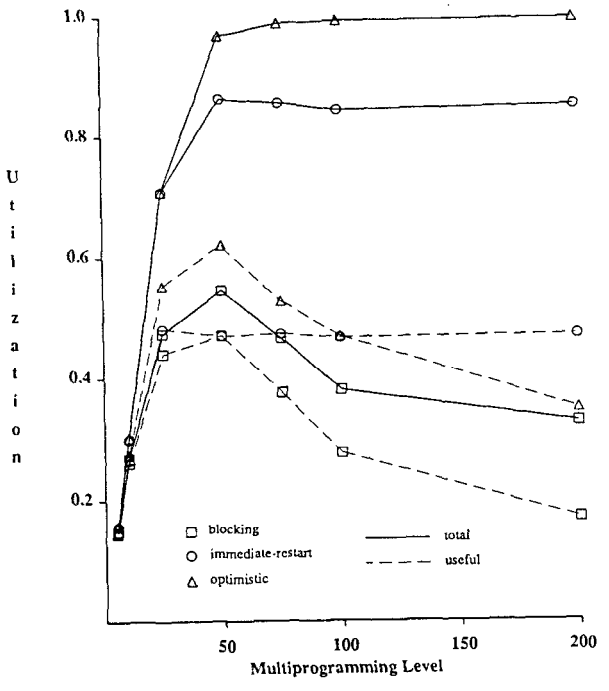


Fig. 28. Disk utilization (5 seconds thinking).

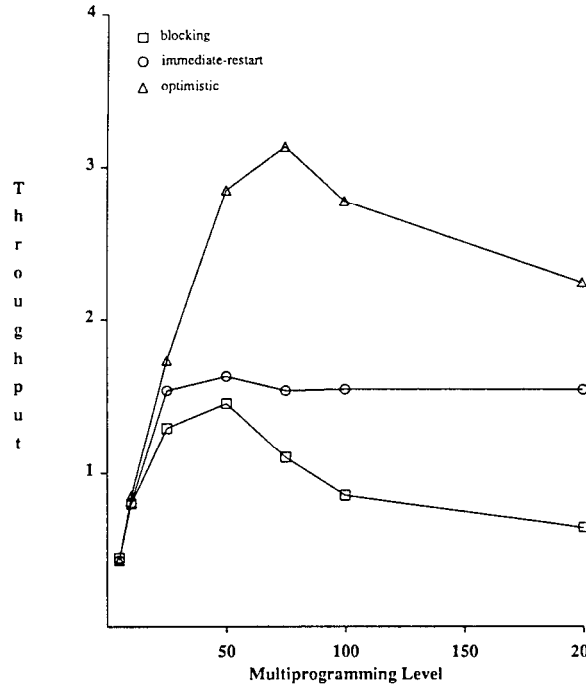


Fig. 29. Throughput (10 seconds thinking).

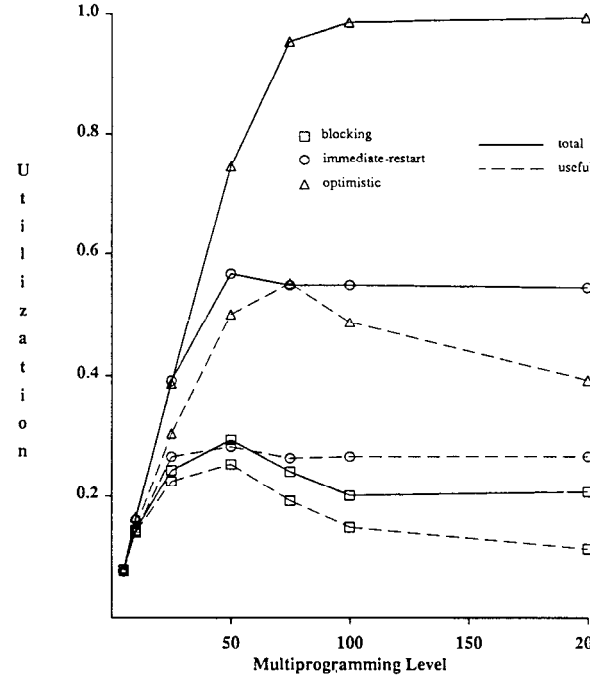


Fig. 30. Disk utilization (10 seconds thinking).

levels of resource utilization. On the other hand, if utilizations are sufficiently low, a restart-oriented algorithm becomes a better choice. Such low resource utilizations arose in our experiments with large numbers of resource units and in our interactive workload experiments with large intratransaction think times. The optimistic algorithm provided the best performance in these cases. Second, the past performance studies discussed in Section 1 were not really contradictory after all: they simply obtained different results because of very different resource modeling assumptions. We obtained results similar to each of the various studies [1, 2, 6, 12, 15, 20, 50, 51] by varying the level of resources that we employed in our database model. Clearly, then, a physically justifiable resource model is a critical component for a reasonable concurrency control performance model. Third, our results indicate that it is important to control the multiprogramming level in a database system for concurrency control reasons. We observed thrashing behavior for locking in the infinite resource case, as did [6, 20, 50, and 51], but in addition we observed that a significant thrashing effect occurs for both locking *and* optimistic concurrency control under higher levels of resource contention. (A similar thrashing effect would also have occurred for the immediate-restart algorithm under higher resource contention levels were it not for the *mpl*-limiting effects of its adaptive restart delay.)

6. TRANSACTION BEHAVIOR ASSUMPTIONS

This section describes experiments that were performed to investigate the performance implications of two modeling assumptions related to transaction behavior. In particular, we examined the impact of alternative assumptions about how restarts are modeled (real versus fake restarts) and how write locks are acquired (with or without upgrades from read locks). Based on the results of the previous section, we performed these experiments under just two resource settings: infinite resources and one resource unit. These two settings are sufficient to demonstrate the important effects of the alternative assumptions, since the results under other settings can be predicted from these two. Except where explicitly noted, the simulation parameters used in this section are the same as those given in Section 4.

6.1 Experiment 6: Modeling Restarts

In this experiment we investigated the impact of transaction-restart modeling on performance. Up to this point, restarts have been modeled by "reincarnating" transactions with their previous read and write sets and then placing them at the end of the ready queue, as described in Section 3. An alternative assumption that has been used for modeling convenience in a number of studies is the *fake restart* assumption, in which a restarted transaction is assumed to be replaced by a new transaction that is independent of the restarted one. In order to model this assumption, we had the simulator reinitialize the read and write sets for restarted transactions in this experiment. The throughput results for the infinite resource case are shown in Figure 31, and Figure 32 shows the associated conflict ratios. Solid lines show the new results obtained using the fake restart assumption, and the dotted lines show the results obtained previously under the real restart model. For the conflict ratio curves, hollow points show restart ratios and

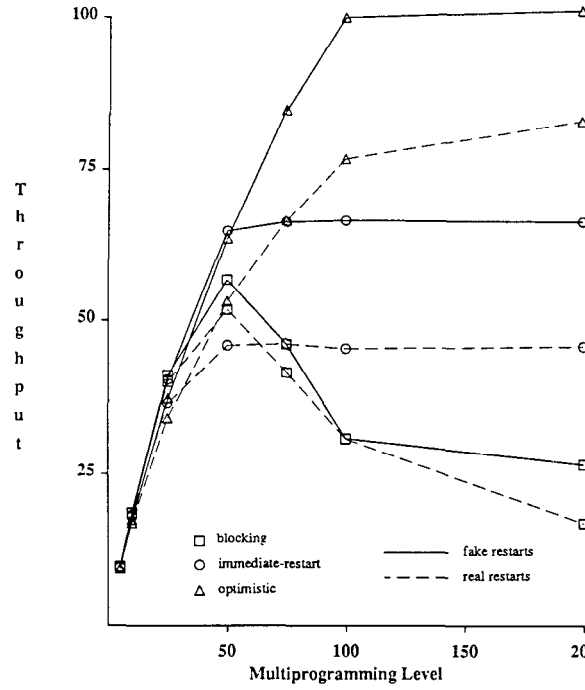


Fig. 31. Throughput (fake restarts, ∞ resources).

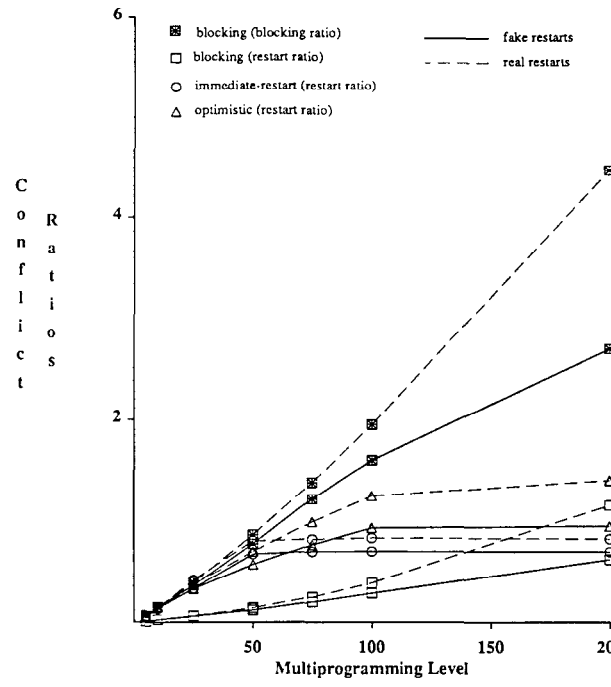


Fig. 32. Conflict ratios (fake restarts, ∞ resources).

solid points show blocking ratios. Figures 33 and 34 show the throughput and conflict ratio results for the limited resource (1 resource unit) case.

In comparing the fake and real restart results for the infinite resource case in Figure 31, several things are clear. The fake restart assumption produces significantly higher throughputs for the immediate-restart and optimistic algorithms. The throughput results for blocking are also higher than under the real restart assumption, but the difference is quite a bit smaller in the case of the blocking algorithm. The restart-oriented algorithms are more sensitive to the fake-restart assumption because they restart transactions much more often. Figure 32 shows how the conflict ratios changed in this experiment, helping to account for the throughput results in more detail. The restart ratios are lower for each of the algorithms under the fake-restart assumption, as is the blocking algorithm's blocking ratio. For each algorithm, if three or more transactions wish to concurrently update an item, repeated conflicts can occur. For blocking, the three transactions will all block and then deadlock when upgrading read locks to write locks, causing two to be restarted, and these two will again block and possibly deadlock. For optimistic, one of the three will commit, which causes the other two to detect readset/writeset intersections and restart, after which one of the remaining two transactions will again restart when the other one commits. A similar problem will occur for immediate-restart, as the three transactions will collide when upgrading their read locks to write locks—only the last of the three will be able to proceed, with the other two being restarted. Fake restarts eliminate this problem, since a restarted transaction comes back as an entirely new transaction. Note that the immediate-restart algorithm has the smallest reduction in its restart ratio. This is because it has a restart delay that helps to alleviate such problems even with real restarts.

Figure 33 shows that, for the limited resource case, the fake-restart assumption again leads to higher throughput predictions for all three concurrency control algorithms. This is due to the reduced restart ratios for all three algorithms (see Figure 34). Fewer restarts lead to better throughput with limited resources, as more resources are available for doing useful (as opposed to wasted) work. For the two restart-oriented algorithms, the difference between fake and real restart performance is fairly constant over most of the range of multiprogramming levels. For blocking, however, fake restarts lead to only a slight increase in throughput at the lower multiprogramming levels. This is expected since its restart ratio is small in this region. As higher multiprogramming levels cause the restart ratio to increase, the difference between fake and real restart performance becomes large. Thus, the results produced under the fake-restart assumption in the limited resource case are biased in favor of the restart-oriented algorithms for low multiprogramming levels. At higher multiprogramming levels, all of the algorithms benefit almost equally from the fake restart assumption (with a slight bias in favor of blocking at the highest multiprogramming level).

6.2 Experiment 7: Write-Lock Acquisition

In this experiment we investigated the impact of write-lock acquisition modeling on performance. Up to now we have assumed that write locks are obtained by upgrading read locks to write locks, as is the case in many real database systems.

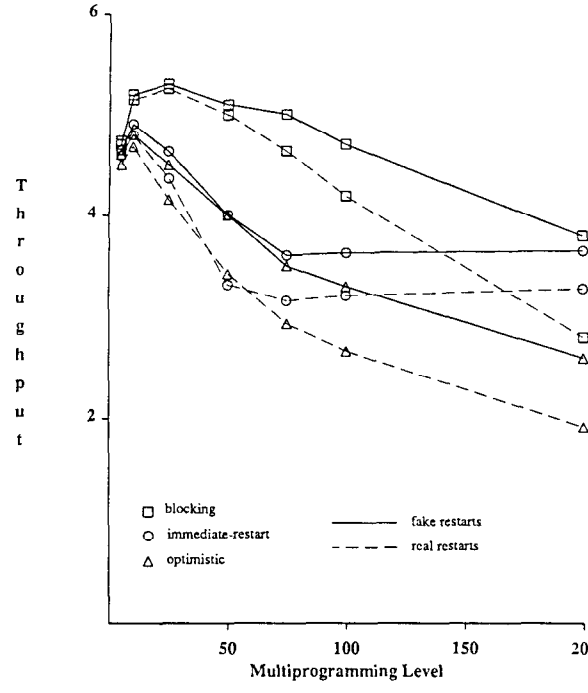


Fig. 33. Throughput (fake restarts, 1 resource unit).

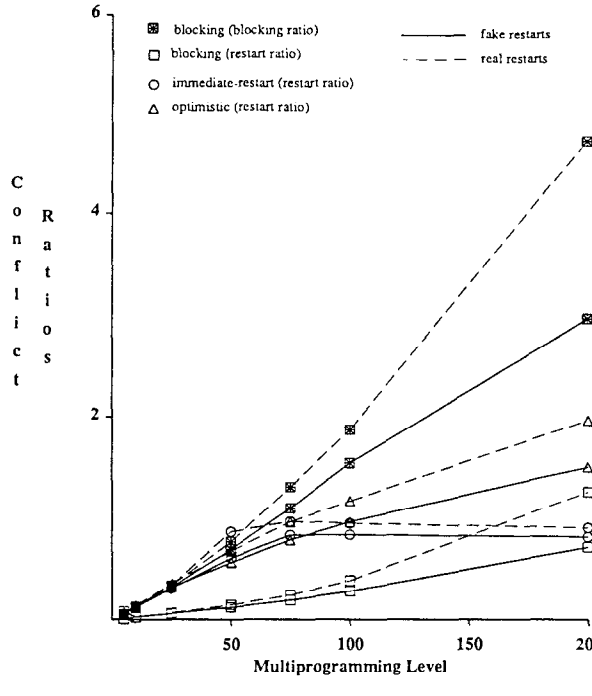


Fig. 34. Conflict ratios (fake restarts, 1 resource unit).

In this section we make an alternative assumption, the *no lock upgrades* assumption, in which a write lock is obtained instead of a read lock on each item that is to eventually be updated the first time the item is read. Figures 35 and 36 show the throughputs and conflict ratios obtained under this new assumption for the infinite resource case, and Figures 37 and 38 show the results for the limited resource case. The line and point-style conventions are the same as those in the previous experiment. Since the optimistic algorithm is (obviously) unaffected by the lock upgrade model, results are only given for the blocking and immediate-restart algorithms.

The results obtained in this experiment are quite easily explained. The upgrade assumption has little effect at the lowest multiprogramming levels, as conflicts are rare there anyway. At higher multiprogramming levels, however, the upgrade assumption does make a difference. The reasons can be understood by considering what happens when two transactions attempt to read and then write the same data item. We consider the blocking algorithm first. With lock upgrades, each transaction will first set a read lock on the item. Later, when one of the transactions is ready to write the item, it will block when it attempts to upgrade its read lock to a write lock; the other transaction will block as well when it requests its lock upgrade. This causes a deadlock, and the younger of the two transactions will be restarted. Without lock upgrades, the first transaction to lock the item will do so using a write lock, and then the other transaction will simply block without causing a deadlock when it makes its lock request. As indicated in Figures 36 and 38, this leads to lower blocking and restart ratios for the blocking algorithm under the no-lock upgrades assumption. For the immediate-restart algorithm, no restart will be eliminated in such a case, since one of the two conflicting transactions must be still restarted. The restart will occur much sooner under the no-lock upgrades assumption, however.

For the infinite resource case (Figures 35 and 36), the throughput predictions are significantly lower for blocking under the no-lock upgrades assumption. This is because write locks are obtained earlier and held significantly longer under this assumption, which leads to longer blocking times and therefore to lower throughput. The elimination of deadlock-induced restarts as described above does not help in this case, since wasted resources are not really an issue with infinite resources. For the immediate-restart algorithm, the no-lock upgrades assumption leads to only a slight throughput increase—although restarts occur earlier, as described above, again this makes little difference with infinite resources.

For the limited resource case (Figures 37 and 38), the throughput predictions for both algorithms are significantly higher under the no-lock upgrades assumption. This is easily explained as well. For blocking, eliminating lock upgrades eliminates upgrade-induced deadlocks, which leads to fewer transactions being restarted. For the immediate-restart algorithm, although no restarts are eliminated, they do occur much sooner in the lives of the restarted transactions under the no-lock upgrades assumption. The resource waste avoided by having fewer restarts with the blocking algorithm or by restarting transactions earlier with the immediate-restart algorithm leads to considerable performance increases for both algorithms when resources are limited.

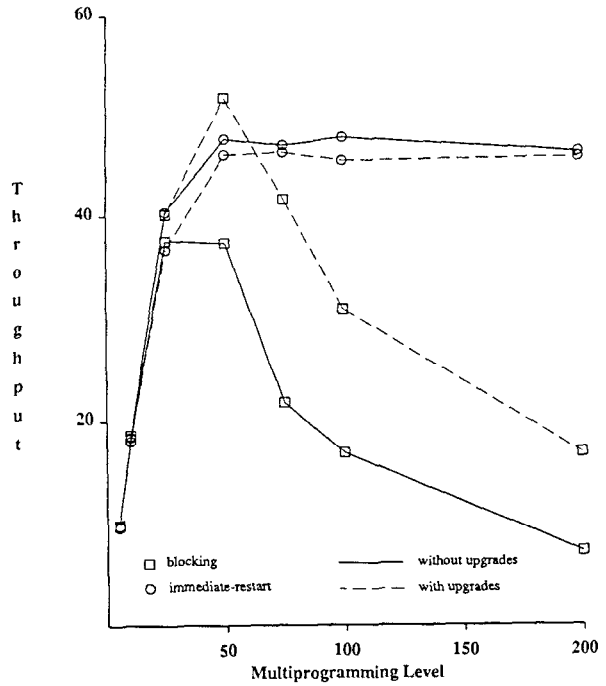


Fig. 35. Throughput (no lock upgrades, ∞ resources).

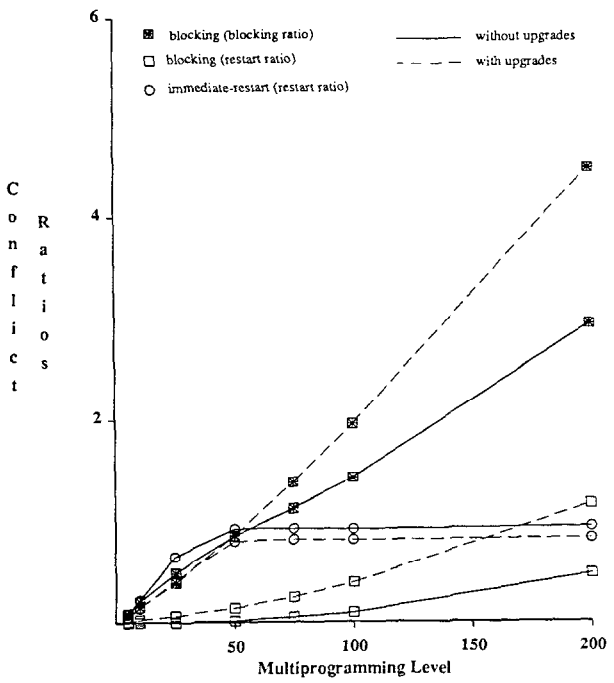


Fig. 36. Conflict ratios (no lock upgrades, ∞ resources).

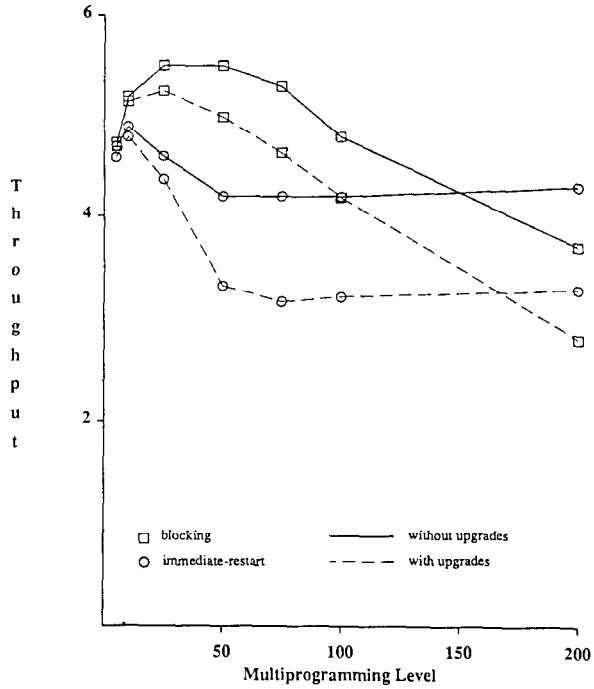


Fig. 37. Throughput (no lock upgrades, 1 resource unit).

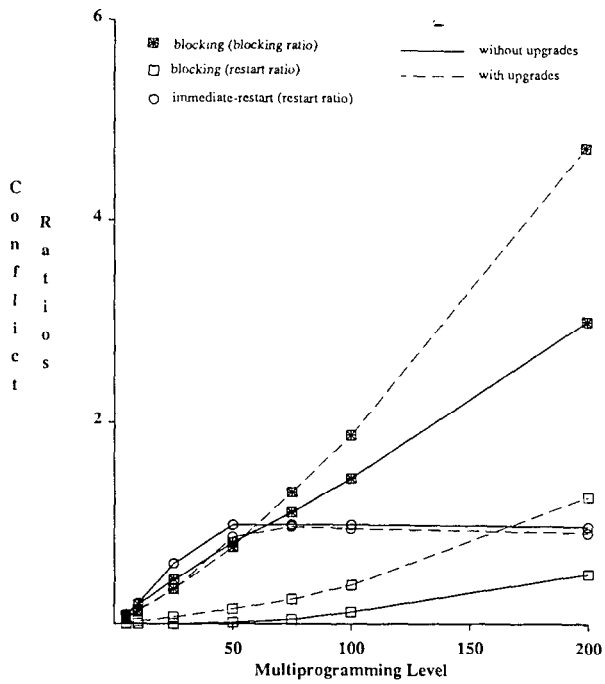


Fig. 38. Conflict ratios (no lock upgrades, 1 resource unit).

6.3 Transaction Behavior Conclusions

Reviewing the results of Experiments 6 and 7, several conclusions can be drawn. First, it is clear from Experiment 6 that the fake-restart assumption does have a significant effect on predicted throughput, particularly for high multiprogramming levels (i.e., when conflicts are frequent). In the infinite resource case, the fake-restart assumption raises the throughput of the restart-oriented algorithms more than it does for blocking, so fake restarts bias the results against blocking somewhat in this case. In the limited resource case, the results produced under the fake-restart assumption are biased in favor of the restart-oriented algorithms at low multiprogramming levels, and all algorithms benefit about equally from the assumption at higher levels of multiprogramming. In both cases, however, the relative performance results are not all that different with and without fake restarts, at least in the sense that assuming fake restarts does not change which algorithm performs the best of the three. Second, it is clear from Experiment 7 that the no-lock upgrades assumption biases the results in favor of the immediate-restart algorithm, particularly in the infinite resource case. That is, the performance of blocking is significantly underestimated using this assumption in the case of infinite resources, and the throughput of the immediate-restart algorithm benefits slightly more from this assumption than blocking does in the limited resource case.

7. CONCLUSIONS AND IMPLICATIONS

In this paper, we argued that a physically justifiable database system model is a requirement for concurrency control performance studies. We described what we feel are the key components of a reasonable model, including a model of the database system and its resources, a model of the user population, and a model of transaction behavior. We then presented our simulation model, which includes all of these components, and we used it to study alternative assumptions about database system resources and transaction behavior.

One specific conclusion of this study is that a concurrency control algorithm that tends to conserve physical resources by blocking transactions that might otherwise have to be restarted is a better choice than a restart-oriented algorithm in an environment where physical resources are limited. Dynamic two-phase locking was found to outperform the immediate-restart and optimistic algorithms for medium to high levels of resource utilization. However, if resource utilizations are low enough so that a large amount of wasted resources can be tolerated, and in addition there are a large number of transactions available to execute, then a restart-oriented algorithm that allows a higher degree of concurrent execution is a better choice. We found the optimistic algorithm to perform the best of the three algorithms tested under these conditions. Low resource utilizations such as these could arise in a database machine with a large number of CPUs and disks and with a number of users similar to those of today's medium to large time-sharing systems. They could also arise in primarily interactive applications in which large think times are common and in which the number of users is such that the utilization of the system is low as a result. It is an open question whether or not such low utilizations will ever actually occur in real systems (i.e., whether

or not such operating regions are sufficiently cost-effective). If not, blocking algorithms will remain the preferred method for database concurrency control.

A more general result of this study is that we have reconfirmed results from a number of other studies, including studies reported in [1, 2, 6, 12, 15, 20, 50, and 51]. We have shown that seemingly contradictory performance results, some of which favored blocking algorithms and others of which favored restarts, are not contradictory at all. The studies are all correct within the limits of their assumptions, particularly their assumptions about system resources. Thus, although it is possible to study the effects of data contention and resource contention separately in some models [50, 51], and although such a separation may be useful in iterative approximation methods for solving concurrency control performance models [M. Vernon, personal communication, 1985], it is clear that one cannot select a concurrency control algorithm for a real system on the basis of such a separation—the proper algorithm choice is strongly resource dependent. A reasonable model of database system resources is a crucial ingredient for studies in which algorithm selection is the goal.

Another interesting result of this study is that the level of multiprogramming in database systems should be carefully controlled. We refer here to the multiprogramming level internal to the database system, which controls the number of transactions that may concurrently compete for data, CPU, and I/O services (as opposed to the number of users that may be attached to the system). As in the case of paging operating systems, if the multiprogramming level is increased beyond a certain level, the blocking and optimistic concurrency control strategies start thrashing. We have confirmed the results of [6, 20, 50, and 51] for locking in the low resource contention case, but more important we have also seen that the effect can be significant for both locking and optimistic concurrency control under higher levels of resource contention. We found that when we delayed restarted transactions by an amount equal to the running average response time, it had the beneficial side effect of limiting the actual multiprogramming level, and the degradation in throughput was arrested (albeit a little bit late). Since the use of a restart delay to limit the multiprogramming level is at best a crude strategy, an adaptive algorithm that dynamically adjusts the multiprogramming level in order to maximize system throughput needs to be designed. Some performance indicators that might be used in the design of such an algorithm are useful resource utilization or running averages of throughput, response time, or conflict ratios. The design of such an adaptive load control algorithm is an open problem.

In addition to our conclusions about the impact of resources in determining concurrency control algorithm performance, we also investigated the effects of two transaction behavior modeling assumptions. With respect to fake versus real restarts, we found that concurrency control algorithms differ somewhat in their sensitivity to this modeling assumption; the results with fake restarts tended to be somewhat biased in favor of the restart-oriented algorithms. However, the overall conclusions about which algorithm performed the best relative to the other algorithms were not altered significantly by this assumption. With respect to the issue of how write-lock acquisition is modeled, we found relative algorithm performance to be more sensitive to this assumption than to the fake-restarts

assumption. The performance of the blocking algorithm was particularly sensitive to the no-lock upgrades assumption in the infinite resource case, with its throughput being underestimated by as much as a factor of two at the higher multiprogramming levels.

In closing, we wish to leave the reader with the following thoughts about computer system resources and the future, due to Bill Wulf:

> Although the hardware costs will continue to fall dramatically and machine speeds will increase equally dramatically, we must assume that our aspirations will rise even more. Because of this, we are not about to face either a cycle or memory surplus. For the near-term future, the dominant effect will not be machine cost or speed alone, but rather a continuing attempt to increase the return from a finite resource—that is, a particular computer at our disposal. [54, p. 41]

ACKNOWLEDGMENTS

The authors wish to acknowledge the anonymous referees for their many insightful comments. We also wish to acknowledge helpful discussions that one or more of us have had with Mary Vernon, Nat Goodman, and (especially) Y. C. Tay. Comments from Rudd Canaday on an earlier version of this paper helped us to improve the presentation. The NSF-sponsored Crystal multicomputer project at the University of Wisconsin provided the many VAX 11/750 CPU-hours that were required for this study.

REFERENCES

1. AGRAWAL, R. Concurrency control and recovery in multiprocessor database machines: Design and performance evaluation, Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisc., 1983.
2. AGRAWAL, R., AND DEWITT, D. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Trans. Database Syst. 10*, 4 (Dec. 1985), 529–564.
3. AGRAWAL, R., CAREY, M., AND DEWITT, D. Deadlock detection is cheap. *ACM-SIGMOD Record 13*, 2 (Jan. 1983).
4. AGRAWAL, R., CAREY, M., AND MCVOY, L. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Trans. Softw. Eng.* To be published.
5. BADAL, D. Correctness of concurrency control and implications in distributed databases. In *Proceedings of the COMPSAC '79 Conference* (Chicago, Nov. 1979). IEEE, New York, 1979, pp. 588–593.
6. BALTER, R., BERARD, P., AND DECITRE, P. Why control of the concurrency level in distributed systems is more fundamental than deadlock management. In *Proceedings of the 1st ACM SIGACT SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Aug. 18–20, 1982). ACM, New York, 1982, pp. 183–193.
7. BERNSTEIN, P., AND GOODMAN, N. Fundamental algorithms for concurrency control in distributed database systems. Tech. Rep., Computer Corporation of America, Cambridge, Mass., 1980.
8. BERNSTEIN, P., AND GOODMAN, N. "Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the 6th International Conference on Very Large Data Bases* (Montreal, Oct. 1980), pp. 285–300.
9. BERNSTEIN, P., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–222.
10. BERNSTEIN, P., AND GOODMAN, N. A sophisticate's introduction to distributed database concurrency control. In *Proceedings of the 8th International Conference on Very Large Data Bases* (Mexico City, Sept. 1982), pp. 62–76.
11. BERNSTEIN, P., SHIPMAN, D., AND WONG, S. Formal aspects in serializability of database concurrency control. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979).

12. CAREY, M. Modeling and evaluation of database concurrency control algorithms. Ph.D. dissertation, Computer Science Division (EECS), University of California, Berkeley, Sept. 1983.
13. CAREY, M. An abstract model of database concurrency control algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Jose, Calif., May 23–26, 1983). ACM, New York, 1983, pp. 97–107.
14. CAREY, M., AND MUHANNA, W. The performance of multiversion concurrency control algorithms. *ACM Trans. Comput. Syst. 4*, 4 (Nov. 1986), 338–378.
15. CAREY, M., AND STONEBRAKER, M. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, Aug. 1984), pp. 107–118.
16. CASANOVA, M. The concurrency control problem for database systems. Ph.D. dissertation, Computer Science Department, Harvard University, Cambridge, Mass. 1979.
17. CERI, S., AND OWICKI, S. On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (Berkeley, Calif., Feb. 1982), ACM, IEEE, New York, 1982.
18. ELHARD, K., AND BAYER, R. A database cache for high performance and fast restart in database systems. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 503–525.
19. ESWAREN, K., GRAY, J., LORIE, R., AND TRAIGER, I. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
20. FRANASZEK, P., AND ROBINSON, J. Limitations of concurrency in transaction processing. *ACM Trans. Database Syst. 10*, 1 (Mar. 1985), 1–28.
21. GALLER, B. Concurrency control performance issues. Ph.D. dissertation, Computer Science Department, University of Toronto, Ontario, Sept. 1982.
22. GOODMAN, N., SURI, R., AND TAY, Y. A simple analytic model for performance of exclusive locking in database systems. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Atlanta, Ga., Mar. 21–23, 1983). ACM, New York, 1983 pp. 203–215.
23. GRAY, J. Notes on database operating systems. In *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, Eds. Springer-Verlag, New York, 1979.
24. GRAY, J., HOMAN, P., KORTH, H., AND OBERMARCK, R. A straw man analysis of the probability of waiting and deadlock in a database system. Tech. Rep. RJ3066, IBM San Jose Research Laboratory, San Jose, Calif., Feb. 1981.
25. HAERDER, T., AND PEINL, P. Evaluating multiple server DBMS in general purpose operating system environments. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, Aug. 1984).
26. IRANI, K., AND LIN, H. Queuing network models for concurrent transaction processing in a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Boston, May 30–June 1, 1979). ACM, New York, 1979.
27. KUNG, H., AND ROBINSON, J. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.
28. LIN, W., AND NOLTE, J. Distributed database control and allocation: Semi-annual report. Tech. Rep., Computer Corporation of America, Cambridge, Mass., Jan. 1982.
29. LIN, W., AND NOLTE, J. Performance of two phase locking. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (Berkeley, Feb. 1982), ACM, IEEE, New York, 1982, pp. 131–160.
30. LIN, W., AND NOLTE, J. Basic timestamp, multiple version timestamp, and two-phase locking. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Oct. 1983).
31. LINDSAY, B., ET AL. Notes on distributed databases, Tech. Rep. RJ2571, IBM San Jose Research Laboratory, San Jose, Calif., 1979.
32. MENASCE, D., AND MUNTZ, R. Locking and deadlock detection in distributed databases. In *Proceedings of the 3rd Berkeley Workshop on Distributed Data Management and Computer Networks* (San Francisco, Aug. 1978). ACM, IEEE, New York, 1978, pp. 215–232.
33. PAPADIMITRIOU, C. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.
34. PEINL, P., AND REUTER, A. Empirical comparison of database concurrency control schemes. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Oct. 1983), pp. 97–108.

35. POTIER, D., AND LEBLANC, P. Analysis of locking policies in database management systems. *Commun. ACM 23*, 10 (Oct. 1980), 584–593.
36. REED, D. Naming and synchronization in a decentralized computer system. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1978.
37. REUTER, A. An analytic model of transaction interference in database systems. IB 68/83, University of Kaiserslautern, West Germany, 1983.
38. REUTER, A. Performance analysis of recovery techniques. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 526–559.
39. RIES, D. The effects of concurrency control on database management system performance. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, Calif., 1979.
40. RIES, D., AND STONEBRAKER, M. Effects of locking granularity on database management system performance. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 233–246.
41. RIES, D., AND STONEBRAKER, M. Locking granularity revisited. *ACM Trans. Database Syst. 4*, 2 (June 1979), 210–227.
42. ROBINSON, J. Design of concurrency controls for transaction processing systems. Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 1982.
43. ROBINSON, J. Experiments with transaction processing on a multi-microprocessor. Tech. Rep. RC9725, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., Dec. 1982.
44. ROSENKRANTZ, D., STEARNS, R., AND LEWIS, P., II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, 2 (June 1978), 178–198.
45. ROWE, L., AND STONEBRAKER, M. The commercial INGRES epilogue. In *The INGRES Papers: Anatomy of a Relational Database System*, M. Stonebraker, Ed. Addison-Wesley, Reading, Mass. 1986.
46. SARGENT, R. Statistical analysis of simulation output data. In *Proceedings of the 4th Annual Symposium on the Simulation of Computer Systems* (Aug. 1976), pp. 39–50.
47. SPITZER, J. Performance prototyping of data management applications. In *Proceedings of the ACM '76 Annual Conference* (Houston, Tx., Oct. 20–22, 1976). ACM, New York, 1976, pp. 287–292.
48. STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng. 5*, 3 (May 1979).
49. STONEBRAKER, M., AND ROWE, L. The Design of POSTGRES. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Washington, D.C., May 28–30, 1986). ACM, New York, 1986, pp. 340–355.
50. TAY, Y. A mean value performance model for locking in databases. Ph.D. dissertation, Computer Science Department, Harvard University, Cambridge, Mass. Feb. 1984.
51. TAY, Y., GOODMAN, N., AND SURI, R. Locking performance in centralized databases. *ACM Trans. Database Syst. 10*, 4 (Dec. 1985), 415–462.
52. THOMAS, R. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180–209.
53. THOMASIAN, A., AND RYU, I. A decomposition solution to the queuing network model of the centralized DBMS with static locking. In *Proceedings of the ACM-SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Minneapolis, Minn., Aug. 29–31, 1983). ACM, New York, 1983, pp. 82–92.
54. WULF, W. Compilers and computer architecture. *IEEE Computer* (July 1981).

Received August 1985; revised August 1986; accepted May 1987

Lottery Scheduling: Flexible Proportional-Share Resource Management

Carl A. Waldspurger * William E. Weihl *

MIT Laboratory for Computer Science
Cambridge, MA 02139 USA

Abstract

This paper presents *lottery scheduling*, a novel randomized resource allocation mechanism. Lottery scheduling provides efficient, responsive control over the relative execution rates of computations. Such control is beyond the capabilities of conventional schedulers, and is desirable in systems that service requests of varying importance, such as databases, media-based applications, and networks. Lottery scheduling also supports modular resource management by enabling concurrent modules to insulate their resource allocation policies from one another. A *currency* abstraction is introduced to flexibly name, share, and protect resource rights. We also show that lottery scheduling can be generalized to manage many diverse resources, such as I/O bandwidth, memory, and access to locks. We have implemented a prototype lottery scheduler for the Mach 3.0 microkernel, and found that it provides flexible and responsive control over the relative execution rates of a wide range of applications. The overhead imposed by our unoptimized prototype is comparable to that of the standard Mach timesharing policy.

1 Introduction

Scheduling computations in multithreaded systems is a complex, challenging problem. Scarce resources must be multiplexed to service requests of varying importance, and the policy chosen to manage this multiplexing can have an enormous impact on throughput and response time. Accurate control over the quality of service provided to users and applications requires support for specifying relative computation rates. Such control is desirable across a wide spectrum of systems. For long-running computations such as scientific applications and simulations, the consumption of computing resources that are shared among users and applications of varying importance must be regulated [Hel93]. For interactive computations such as databases and media-based applications, programmers and users need the ability to rapidly focus available resources on tasks that are currently important [Dui90].

Few general-purpose schemes even come close to supporting flexible, responsive control over service rates. Those that do exist generally rely upon a simple notion of *priority* that does not provide the encapsulation and modularity properties required for the engineering of large software systems. In fact, with the exception of hard real-time systems, it has been observed that the assignment of priorities and dynamic priority adjustment schemes are often ad-hoc [Dei90]. Even popular priority-based schemes for CPU allocation such as *decay-usage scheduling* are poorly understood, despite the fact that they are employed by numerous operating systems, including Unix [Hel93].

Existing *fair share* schedulers [Hen84, Kay88] and *microeconomic* schedulers [Fer88, Wal92] successfully address some of the problems with absolute priority schemes. However, the assumptions and overheads associated with these systems limit them to relatively coarse control over long-running computations. Interactive systems require rapid, dynamic control over scheduling at a time scale of milliseconds to seconds.

We have developed *lottery scheduling*, a novel randomized mechanism that provides responsive control over the relative execution rates of computations. Lottery scheduling efficiently implements *proportional-share* resource management — the resource consumption rates of active computations are proportional to the relative shares that they are allocated. Lottery scheduling also provides excellent support for modular resource management. We have developed a prototype lottery scheduler for the Mach 3.0 microkernel, and found that it provides efficient, flexible control over the relative execution rates of compute-bound tasks, video-based applications, and client-server interactions. This level of control is not possible with current operating systems, in which adjusting scheduling parameters to achieve specific results is at best a black art.

Lottery scheduling can be generalized to manage many diverse resources, such as I/O bandwidth, memory, and access to locks. We have developed a prototype lottery-scheduled mutex implementation, and found that it provides flexible control over mutex acquisition rates. A variant of lottery scheduling can also be used to efficiently manage space-shared resources such as memory.

*E-mail: {carl, weihl}@lcs.mit.edu. World Wide Web: http://www.psg.lcs.mit.edu/. The first author was supported in part by an AT&T USL Fellowship and by a grant from the MIT X Consortium. Prof. Weihl is currently supported by DEC while on sabbatical at DEC SRC. This research was also supported by ARPA under contract N00014-94-1-0985, by grants from AT&T and IBM, and by an equipment grant from DEC. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

In the next section, we describe the basic lottery scheduling mechanism. Section 3 discusses techniques for modular resource management based on lottery scheduling. Implementation issues and a description of our prototype are presented in Section 4. Section 5 discusses the results of several quantitative experiments. Generalizations of the lottery scheduling approach are explored in Section 6. In Section 7, we examine related work. Finally, we summarize our conclusions in Section 8.

2 Lottery Scheduling

Lottery scheduling is a randomized resource allocation mechanism. Resource rights are represented by lottery *tickets*.[1] Each allocation is determined by holding a *lottery*; the resource is granted to the client with the winning ticket. This effectively allocates resources to competing clients in proportion to the number of tickets that they hold.

2.1 Resource Rights

Lottery tickets encapsulate resource rights that are abstract, relative, and uniform. They are *abstract* because they quantify resource rights independently of machine details. Lottery tickets are *relative*, since the fraction of a resource that they represent varies dynamically in proportion to the contention for that resource. Thus, a client will obtain more of a lightly contended resource than one that is highly contended; in the worst case, it will receive a share proportional to its share of tickets in the system. Finally, tickets are *uniform* because rights for heterogeneous resources can be homogeneously represented as tickets. These properties of lottery tickets are similar to those of money in computational economies [Wal92].

2.2 Lotteries

Scheduling by lottery is *probabilistically fair*. The expected allocation of resources to clients is proportional to the number of tickets that they hold. Since the scheduling algorithm is randomized, the actual allocated proportions are not guaranteed to match the expected proportions exactly. However, the disparity between them decreases as the number of allocations increases.

The number of lotteries won by a client has a binomial distribution. The probability p that a client holding t tickets will win a given lottery with a total of T tickets is simply $p = t/T$. After n identical lotteries, the expected number of wins w is $E[w] = np$, with variance $\sigma_w^2 = np(1-p)$. The coefficient of variation for the observed proportion of wins is $\sigma_w/E[w] = \sqrt{(1-p)/np}$. Thus, a client's throughput is proportional to its ticket allocation, with accuracy that improves with $\sqrt{n}$.

[1] A single physical ticket may represent any number of logical tickets. This is similar to monetary notes, which may be issued in different denominations.

The number of lotteries required for a client's first win has a geometric distribution. The expected number of lotteries n that a client must wait before its first win is $E[n] = 1/p$, with variance $\sigma_n^2 = (1-p)/p^2$. Thus, a client's average response time is inversely proportional to its ticket allocation. The properties of both binomial and geometric distributions are well-understood [Tri82].

With a scheduling quantum of 10 milliseconds (100 lotteries per second), reasonable fairness can be achieved over subsecond time intervals. As computation speeds continue to increase, shorter time quanta can be used to further improve accuracy while maintaining a fixed proportion of scheduler overhead.

Since any client with a non-zero number of tickets will eventually win a lottery, the conventional problem of starvation does not exist. The lottery mechanism also operates fairly when the number of clients or tickets varies dynamically. For each allocation, every client is given a fair chance of winning proportional to its share of the total number of tickets. Since any changes to relative ticket allocations are immediately reflected in the next allocation decision, lottery scheduling is extremely responsive.

3 Modular Resource Management

The explicit representation of resource rights as lottery tickets provides a convenient substrate for modular resource management. Tickets can be used to insulate the resource management policies of independent modules, because each ticket probabilistically guarantees its owner the right to a worst-case resource consumption rate. Since lottery tickets abstractly encapsulate resource rights, they can also be treated as first-class objects that may be transferred in messages.

This section presents basic techniques for implementing resource management policies with lottery tickets. Detailed examples are presented in Section 5.

3.1 Ticket Transfers

Ticket transfers are explicit transfers of tickets from one client to another. Ticket transfers can be used in any situation where a client blocks due to some dependency. For example, when a client needs to block pending a reply from an RPC, it can temporarily transfer its tickets to the server on which it is waiting. This idea also conveniently solves the conventional priority inversion problem in a manner similar to priority inheritance [Sha90]. Clients also have the ability to divide ticket transfers across multiple servers on which they may be waiting.

3.2 Ticket Inflation

Ticket inflation is an alternative to explicit ticket transfers in which a client can escalate its resource rights by creating more lottery tickets. In general, such inflation should be

disallowed, since it violates desirable modularity and load insulation properties. For example, a single client could easily monopolize a resource by creating a large number of lottery tickets. However, ticket inflation can be very useful among mutually trusting clients; inflation and deflation can be used to adjust resource allocations without explicit communication.

3.3 Ticket Currencies

In general, resource management abstraction barriers are desirable across logical trust boundaries. Lottery scheduling can easily be extended to express resource rights in units that are local to each group of mutually trusting clients. A unique *currency* is used to denominate tickets within each trust boundary. Each currency is backed, or *funded*, by tickets that are denominated in more primitive currencies. Currency relationships may form an arbitrary acyclic graph, such as a hierarchy of currencies. The effects of inflation can be locally contained by maintaining an *exchange rate* between each local currency and a *base* currency that is conserved. The currency abstraction is useful for flexibly naming, sharing, and protecting resource rights. For example, an access control list associated with a currency could specify which principals have permission to inflate it by creating new tickets.

3.4 Compensation Tickets

A client which consumes only a fraction f of its allocated resource quantum can be granted a *compensation ticket* that inflates its value by $1/f$ until the client starts its next quantum. This ensures that each client's resource consumption, equal to f times its per-lottery win probability p, is adjusted by $1/f$ to match its allocated share p. Without compensation tickets, a client that does not consume its entire allocated quantum would receive less than its entitled share of the processor.

4 Implementation

We have implemented a prototype lottery scheduler by modifying the Mach 3.0 microkernel (MK82) [Acc86, Loe92] on a 25MHz MIPS-based DECStation 5000/125. Full support is provided for ticket transfers, ticket inflation, ticket currencies, and compensation tickets.[2] The scheduling quantum on this platform is 100 milliseconds.

4.1 Random Numbers

An efficient lottery scheduler requires a fast way to generate uniformly-distributed random numbers. We have implemented a pseudo-random number generator based on the Park-Miller algorithm [Par88, Car90] that executes in approximately 10 RISC instructions. Our assembly-language implementation is listed in Appendix A.

[2] Our first lottery scheduler implementation, developed for the *Prelude* [Wei91] runtime system, lacked support for ticket transfers and currencies.

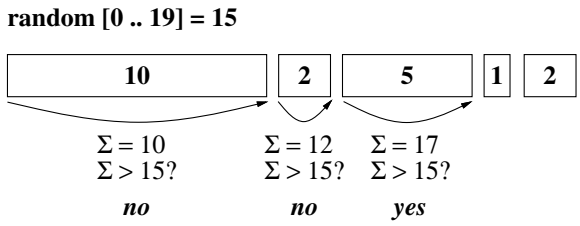


Figure 1: **Example Lottery.** Five clients compete in a list-based lottery with a total of 20 tickets. The fifteenth ticket is randomly selected, and the client list is searched for the winner. A running ticket sum is accumulated until the winning ticket value is reached. In this example, the third client is the winner.

4.2 Lotteries

A straightforward way to implement a centralized lottery scheduler is to randomly select a winning ticket, and then search a list of clients to locate the client holding that ticket. This requires a random number generation and $O(n)$ operations to traverse a client list of length n, accumulating a running ticket sum until it reaches the winning value. An example list-based lottery is presented in Figure 1.

Various optimizations can reduce the average number of clients that must be examined. For example, if the distribution of tickets to clients is uneven, ordering the clients by decreasing ticket counts can substantially reduce the average search length. Since those clients with the largest number of tickets will be selected most frequently, a simple "move to front" heuristic can be very effective.

For large n, a more efficient implementation is to use a tree of partial ticket sums, with clients at the leaves. To locate the client holding a winning ticket, the tree is traversed starting at the root node, and ending with the winning client leaf node, requiring only $O(\lg n)$ operations. Such a tree-based implementation can also be used as the basis of a distributed lottery scheduler.

4.3 Mach Kernel Interface

The kernel representation of tickets and currencies is depicted in Figure 2. A minimal lottery scheduling interface is exported by the microkernel. It consists of operations to create and destroy tickets and currencies, operations to fund and unfund a currency (by adding or removing a ticket from its list of backing tickets), and operations to compute the current value of tickets and currencies in base units.

Our lottery scheduling policy co-exists with the standard timesharing and fixed-priority policies. A few high-priority threads (such as the Ethernet driver) created by the Unix server (UX41) remain at their original fixed priorities.

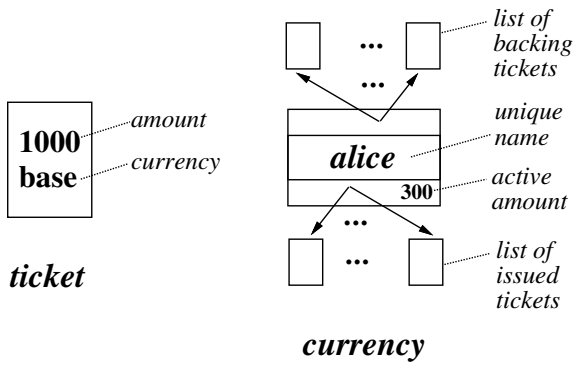


Figure 2: **Kernel Objects.** A *ticket* object contains an amount denominated in some currency. A *currency* object contains a name, a list of tickets that back the currency, a list of all tickets issued in the currency, and an active amount sum for all issued tickets.

4.4 Ticket Currencies

Our prototype uses a simple scheme to convert ticket amounts into base units. Each currency maintains an active amount sum for all of its issued tickets. A ticket is *active* while it is being used by a thread to compete in a lottery. When a thread is removed from the run queue, its tickets are deactivated; they are reactivated when the thread rejoins the run queue.[3] If a ticket deactivation changes a currency's active amount to zero, the deactivation propagates to each of its backing tickets. Similarly, if a ticket activation changes a currency's active amount from zero, the activation propagates to each of its backing tickets.

A currency's value is computed by summing the value of its backing tickets. A ticket's value is computed by multiplying the value of the currency in which it is denominated by its share of the active amount issued in that currency. The value of a ticket denominated in the base currency is defined to be its face value amount. An example currency graph with base value conversions is presented in Figure 3. Currency conversions can be accelerated by caching values or exchange rates, although this is not implemented in our prototype.

Our scheduler uses the simple list-based lottery with a move-to-front heuristic, as described earlier in Section 4.2. To handle multiple currencies, a winning ticket value is selected by generating a random number between zero and the total number of active tickets in the *base* currency. The run queue is then traversed as described earlier, except that the running ticket sum accumulates the value of each thread's currency in *base* units until the winning value is reached.

[3] A blocked thread may transfer its tickets to another thread that will actively use them. For example, a thread blocked pending a reply from an RPC transfers its tickets to the server thread on which it is waiting.

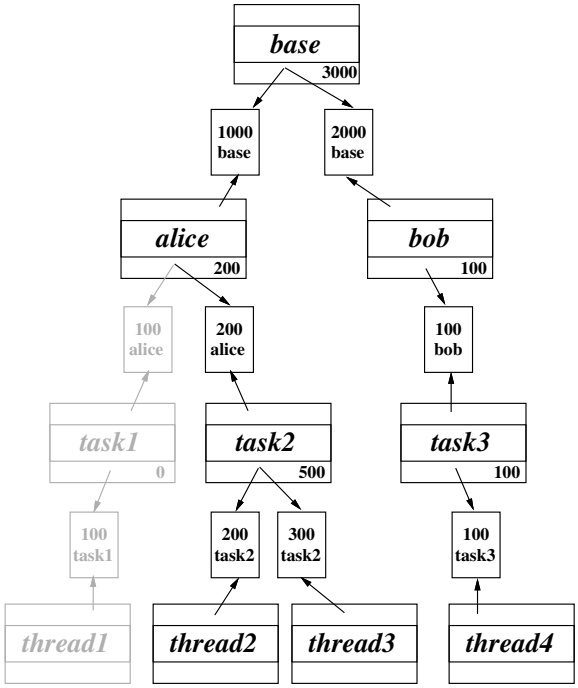


Figure 3: **Example Currency Graph.** Two users compete for computing resources. Alice is executing two tasks: *task1* is currently inactive, and *task2* has two runnable threads. Bob is executing one single-threaded task, *task3*. The current values in base units for the runnable threads are *thread2* = 400, *thread3* = 600, and *thread4* = 2000. In general, currencies can also be used for groups of users or applications, and currency relationships may form an acyclic graph instead of a strict hierarchy.

4.5 Compensation Tickets

As discussed in Section 3.4, a thread which consumes only a fraction f of its allocated time quantum is automatically granted a compensation ticket that inflates its value by $1/f$ until the thread starts its next quantum. This is consistent with proportional sharing, and permits I/O-bound tasks that use few processor cycles to start quickly.

For example, suppose threads A and B each hold tickets valued at 400 base units. Thread A always consumes its entire 100 millisecond time quantum, while thread B uses only 20 milliseconds before yielding the processor. Since both A and B have equal funding, they are equally likely to win a lottery when both compete for the processor. However, thread B uses only $f = 1/5$ of its allocated time, allowing thread A to consume five times as much CPU, in violation of their $1:1$ allocation ratio. To remedy this situation, thread B is granted a compensation ticket valued at 1600 base units when it yields the processor. When B next competes for the processor, its total funding will be $400/f = 2000$ base units. Thus, on average B will win the processor lottery five times as often as A, each time consuming $1/5$ as much of its quantum as A, achieving the desired $1:1$ allocation ratio.

4.6 Ticket Transfers

The `mach_msg` system call was modified to temporarily transfer tickets from client to server for synchronous RPCs. This automatically redirects resource rights from a blocked client to the server computing on its behalf. A transfer is implemented by creating a new ticket denominated in the client's currency, and using it to fund the server's currency. If the server thread is already waiting when `mach_msg` performs a synchronous call, it is immediately funded with the transfer ticket. If no server thread is waiting, then the transfer ticket is placed on a list that is checked by the server thread when it attempts to receive the call message.[4] During a reply, the transfer ticket is simply destroyed.

4.7 User Interface

Currencies and tickets can be manipulated via a command-line interface. User-level commands exist to create and destroy tickets and currencies (`mktkt`, `rmtkt`, `mkcur`, `rmcur`), fund and unfund currencies (`fund`, `unfund`), obtain information (`lstkt`, `lscur`), and to execute a shell command with specified funding (`fundx`). Since the Mach microkernel has no concept of user and we did not modify the Unix server, these commands are setuid root.[5] A complete lottery scheduling system should protect currencies by using access control lists or Unix-style permissions based on user and group membership.

5 Experiments

In order to evaluate our prototype lottery scheduler, we conducted experiments designed to quantify its ability to flexibly, responsively, and efficiently control the relative execution rates of computations. The applications used in our experiments include the compute-bound Dhrystone benchmark, a Monte-Carlo numerical integration program, a multithreaded client-server application for searching text, and competing MPEG video viewers.

5.1 Fairness

Our first experiment measured the accuracy with which our lottery scheduler could control the relative execution rates of computations. Each point plotted in Figure 4 indicates the relative execution rate that was observed for two tasks executing the Dhrystone benchmark [Wei84] for sixty seconds with a given relative ticket allocation. Three runs were executed for each integral ratio between one and ten.

[4] In this case, it would be preferable to instead fund all threads capable of receiving the message. For example, a server task with fewer threads than incoming messages should be directly funded. This would accelerate all server threads, decreasing the delay until one becomes available to service the waiting message.

[5] The `fundx` command only executes as root to initialize its task currency funding. It then performs a setuid back to the original user before invoking `exec`.

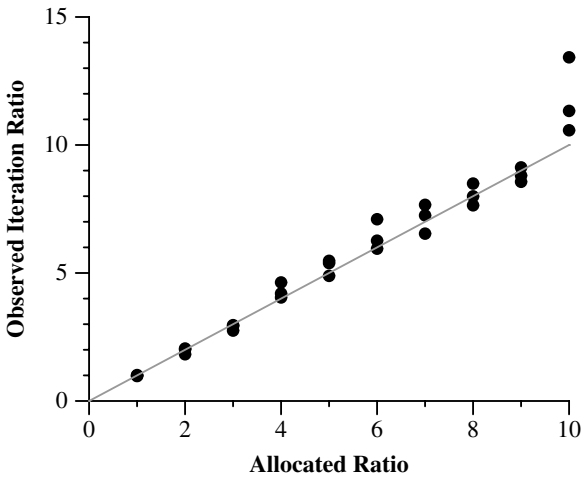


Figure 4: **Relative Rate Accuracy.** For each allocated ratio, the observed ratio is plotted for each of three 60 second runs. The gray line indicates the ideal where the two ratios are identical.

With the exception of the run for which the $10:1$ allocation resulted in an average ratio of $13.42:1$, all of the observed ratios are close to their corresponding allocations. As expected, the variance is greater for larger ratios. However, even large ratios converge toward their allocated values over longer time intervals. For example, the observed ratio averaged over a three minute period for a $20:1$ allocation was $19.08:1$.

Although the results presented in Figure 4 indicate that the scheduler can successfully control computation rates, we should also examine its behavior over shorter time intervals. Figure 5 plots average iteration counts over a series of 8 second time windows during a single 200 second execution with a $2:1$ allocation. Although there is clearly some variation, the two tasks remain close to their allocated ratios throughout the experiment. Note that if a scheduling quantum of 10 milliseconds were used instead of the 100 millisecond Mach quantum, the same degree of fairness would be observed over a series of subsecond time windows.

5.2 Flexible Control

A more interesting use of lottery scheduling involves dynamically controlled ticket inflation. A practical application that benefits from such control is the Monte-Carlo algorithm [Pre88]. Monte-Carlo is a probabilistic algorithm that is widely used in the physical sciences for computing average properties of systems. Since errors in the computed average are proportional to $1/\sqrt{n}$, where n is the number of trials, accurate results require a large number of trials.

Scientists frequently execute several separate Monte-Carlo experiments to explore various hypotheses. It is often desirable to obtain approximate results quickly whenever a new experiment is started, while allowing older experiments to continue reducing their error at a slower rate [Hog88].

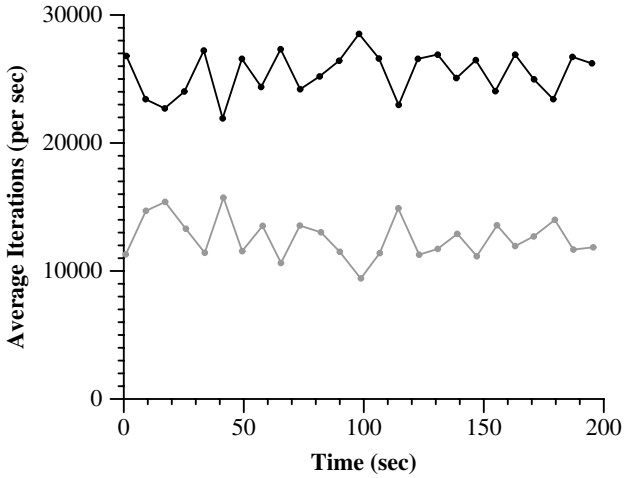


Figure 5: **Fairness Over Time.** Two tasks executing the Dhrystone benchmark with a 2 : 1 ticket allocation. Averaged over the entire run, the two tasks executed 25378 and 12619 iterations/sec., for an actual ratio of 2.01 : 1.

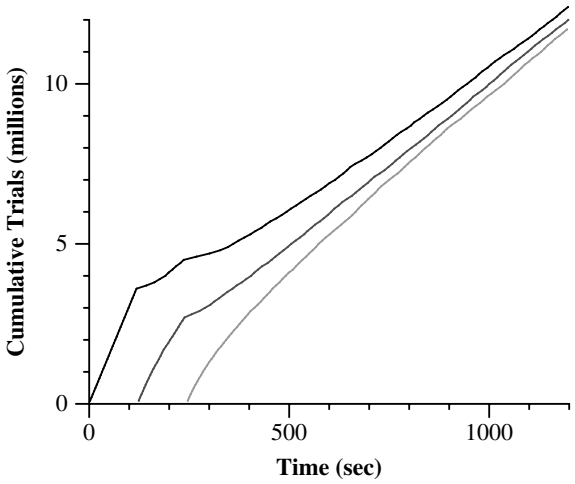


Figure 6: **Monte-Carlo Execution Rates.** Three identical Monte-Carlo integrations are started two minutes apart. Each task periodically sets its ticket value to be proportional to the square of its relative error, resulting in the convergent behavior. The "bumps" in the curves mirror the decreasing slopes of new tasks that quickly reduce their error.

This goal would be impossible with conventional schedulers, but can be easily achieved in our system by dynamically adjusting an experiment's ticket value as a function of its current relative error. This allows a new experiment with high error to quickly catch up to older experiments by executing at a rate that starts high but then tapers off as its relative error approaches that of its older counterparts.

Figure 6 plots the total number of trials computed by each of three staggered Monte-Carlo tasks. Each task is based on the sample code presented in [Pre88], and is allocated a share of time that is proportional to the square of its relative error.[6] When a new task is started, it initially receives a large share of the processor. This share diminishes as the task reduces its error to a value closer to that of the other executing tasks.

A similar form of dynamic control may also be useful in graphics-intensive programs. For example, a rendering operation could be granted a large share of processing resources until it has displayed a crude outline or wire-frame, and then given a smaller share of resources to compute a more polished image.

5.3 Client-Server Computation

As mentioned in Section 4.6, the Mach IPC primitive `mach_msg` was modified to temporarily transfer tickets from client to server on synchronous remote procedure calls. Thus, a client automatically redirects its resource rights to the server that is computing on its behalf. Multithreaded servers will process requests from different clients at the rates defined by their respective ticket allocations.

[6] Any monotonically increasing function of the relative error would cause convergence. A linear function would cause the tasks to converge more slowly; a cubic function would result in more rapid convergence.

We developed a simple multithreaded client-server application that shares properties with real databases and information retrieval systems. Our server initially loads a 4.6 Mbyte text file "database" containing the complete text to all of William Shakespeare's plays.[7] It then forks off several worker threads to process incoming queries from clients. One query operation supported by the server is a case-insensitive substring search over the entire database, which returns a count of the matches found.

Figure 7 presents the results of executing three database clients with an 8 : 3 : 1 ticket allocation. The server has no tickets of its own, and relies completely upon the tickets transferred by clients. Each client repeatedly sends requests to the server to count the occurrences of the same search string.[8] The high-priority client issues a total of 20 queries and then terminates. The other two clients continue to issue queries for the duration of the entire experiment.

The ticket allocations affect both response time and throughput. When the high-priority client has completed its 20 requests, the other clients have completed a total of 10 requests, matching their overall 8 : 4 allocation. Over the entire experiment, the clients with a 3 : 1 ticket allocation respectively complete 38 and 13 queries, which closely matches their allocation, despite their transient competition with the high-priority client. While the high-priority client is active, the average response times seen by the clients are 17.19, 43.19, and 132.20 seconds, yielding relative speeds of 7.69 : 2.51 : 1. After the high-priority client terminates,

[7] A disk-based database could use lotteries to schedule disk bandwidth; this is not implemented in our prototype.

[8] The string used for this experiment was `lottery`, which incidentally occurs a total of 8 times in Shakespeare's plays.

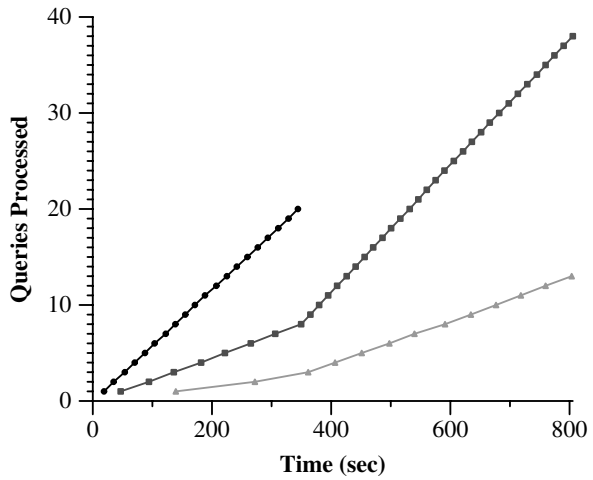


Figure 7: **Query Processing Rates.** Three clients with an $8:3:1$ ticket allocation compete for service from a multithreaded database server. The observed throughput and response time ratios closely match this allocation.

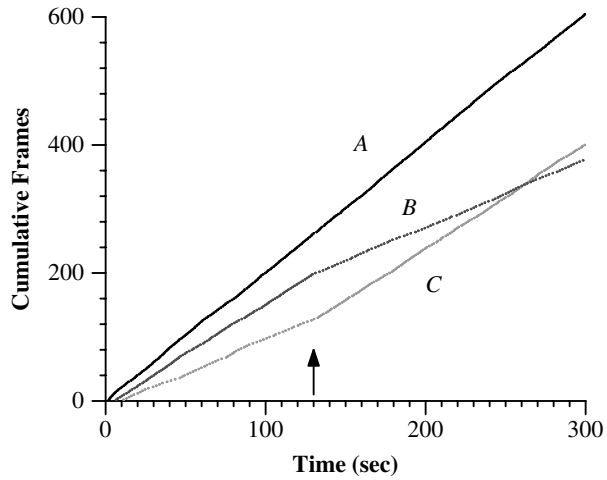


Figure 8: **Controlling Video Rates.** Three MPEG viewers are given an initial $A:B:C = 3:2:1$ allocation, which is changed to $3:1:2$ at the time indicated by the arrow. The total number of frames displayed is plotted for each viewer. The actual frame rate ratios were $1.92:1.50:1$ and $1.92:1:1.53$, respectively, due to distortions caused by the X server.

the response times are 44.17 and 15.18 seconds, for a $2.91:1$ ratio. For all average response times, the standard deviation is less than 7% of the average.

A similar form of control could be employed by database or transaction-processing applications to manage the response times seen by competing clients or transactions. This would be useful in providing different levels of service to clients or transactions with varying importance (or real monetary funding).

5.4 Multimedia Applications

Media-based applications are another domain that can benefit from lottery scheduling. Compton and Tennenhouse described the need to control the quality of service when two or more video viewers are displayed — a level of control not offered by current operating systems [Com94]. They attempted, with mixed success, to control video display rates at the application level among a group of mutually trusting viewers. Cooperating viewers employed feedback mechanisms to adjust their relative frame rates. Inadequate and unstable metrics for system load necessitated substantial tuning, based in part on the number of active viewers. Unexpected positive feedback loops also developed, leading to significant divergence from intended allocations.

Lottery scheduling enables the desired control at the operating-system level, eliminating the need for mutually trusting or well-behaved applications. Figure 8 depicts the execution of three `mpeg_play` video viewers (A, B, and C) displaying the same music video. Tickets were initially allocated to achieve relative display rates of $A:B:C = 3:2:1$, and were then changed to $3:1:2$ at the time indicated by the arrow. The observed per-second frame rates were initially $2.03:1.59:1.06$ ($1.92:1.50:1$ ratio), and then $2.02:1.05:1.61$ ($1.92:1:1.53$ ratio) after the change. Unfortunately, these results were distorted by the round-robin processing of client requests by the single-threaded X11R5 server. When run with the `-no_display` option, frame rates such as $6.83:4.56:2.23$ ($3.06:2.04:1$ ratio) were typical.

5.5 Load Insulation

Support for multiple ticket currencies facilitates modular resource management. A currency defines a resource management abstraction barrier that locally contains intra-currency fluctuations such as inflation. The currency abstraction can be used to flexibly isolate or group users, tasks, and threads.

Figure 9 plots the progress of five tasks executing the Dhrystone benchmark. Let *amount.currency* denote a ticket allocation of *amount* denominated in *currency*. Currencies A and B have identical funding. Tasks $A1$ and $A2$ have allocations of $100.A$ and $200.A$, respectively. Tasks $B1$ and $B2$ have allocations of $100.B$ and $200.B$, respectively. Halfway through the experiment, a new task, $B3$, is started with an allocation of $300.B$. Although this inflates the total number of tickets denominated in currency B from 300 to 600, there is no effect on tasks in currency A. The aggregate iteration ratio of A tasks to B tasks is $1.01:1$ before $B3$ is started, and $1.00:1$ after $B3$ is started. The slopes for the individual tasks indicate that $A1$ and $A2$ are not affected by task $B3$, while $B1$ and $B2$ are slowed to approximately half their original rates, corresponding to the factor of two inflation caused by $B3$.

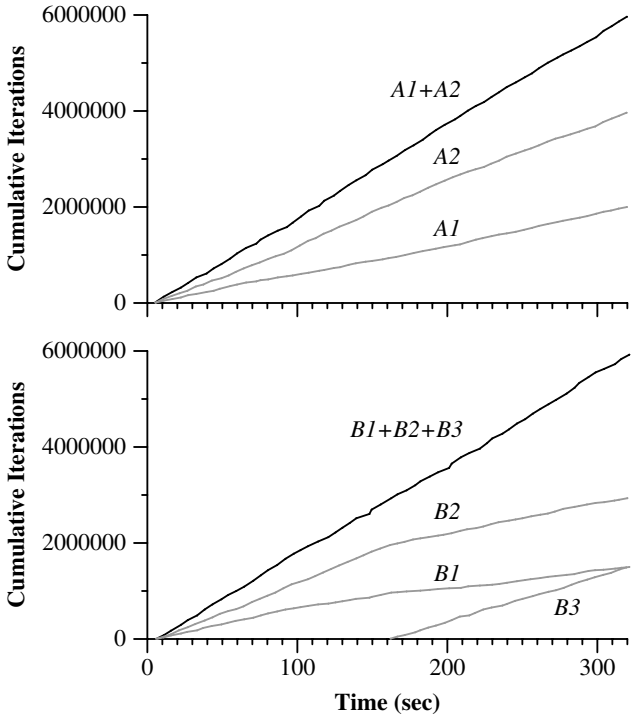


Figure 9: **Currencies Insulate Loads.** Currencies A and B are identically funded. Tasks $A1$ and $A2$ are respectively allocated tickets worth $100.A$ and $200.A$. Tasks $B1$ and $B2$ are respectively allocated tickets worth $100.B$ and $200.B$. Halfway through the experiment, task $B3$ is started with an allocation of $300.B$. The resulting inflation is locally contained within currency B, and affects neither the progress of tasks in currency A, nor the aggregate $A:B$ progress ratio.

5.6 System Overhead

The core lottery scheduling mechanism is extremely lightweight; a tree-based lottery need only generate a random number and perform $\lg n$ additions and comparisons to select a winner among n clients. Thus, low-overhead lottery scheduling is possible in systems with a scheduling granularity as small as a thousand RISC instructions.

Our prototype scheduler, which includes full support for currencies, has not been optimized. To assess system overhead, we used the same executables and workloads under both our kernel and the unmodified Mach kernel; three separate runs were performed for each experiment. Overall, we found that the overhead imposed by our prototype lottery scheduler is comparable to that of the standard Mach timesharing policy. Since numerous optimizations could be made to our list-based lottery, simple currency conversion scheme, and other untuned aspects of our implementation, efficient lottery scheduling does not pose any challenging problems.

Our first experiment consisted of three Dhrystone benchmark tasks running concurrently for 200 seconds. Compared to unmodified Mach, 2.7% fewer iterations were executed under lottery scheduling. For the same experiment with eight tasks, lottery scheduling was observed to be 0.8% slower. However, the standard deviations across individual runs for unmodified Mach were comparable to the absolute differences observed between the kernels. Thus, the measured differences are not very significant.

We also ran a performance test using the multithreaded database server described in Section 5.3. Five client tasks each performed 20 queries, and the time between the start of the first query and the completion of the last query was measured. We found that this application executed 1.7% faster under lottery scheduling. For unmodified Mach, the average run time was 1155.5 seconds; with lottery scheduling, the average time was 1135.5 seconds. The standard deviations across runs for this experiment were less than 0.1% of the averages, indicating that the small measured differences are significant.[9]

6 Managing Diverse Resources

Lotteries can be used to manage many diverse resources, such as processor time, I/O bandwidth, and access to locks. Lottery scheduling also appears promising for scheduling communication resources, such as access to network ports. For example, ATM switches schedule virtual circuits to determine which buffered cell should next be forwarded. Lottery scheduling could be used to provide different levels of service to virtual circuits competing for congested channels. In general, a lottery can be used to allocate resources wherever queueing is necessary for resource access.

6.1 Synchronization Resources

Contention due to synchronization can substantially affect computation rates. Lottery scheduling can be used to control the relative waiting times of threads competing for lock access. We have extended the Mach CThreads library to support a lottery-scheduled mutex type in addition to the standard mutex implementation. A lottery-scheduled mutex has an associated *mutex currency* and an *inheritance ticket* issued in that currency.

All threads that are blocked waiting to acquire the mutex perform ticket transfers to fund the mutex currency. The mutex transfers its inheritance ticket to the thread which currently holds the mutex. The net effect of these transfers is that a thread which acquires the mutex executes with its own funding plus the funding of all waiting threads, as depicted in Figure 10. This solves the priority inversion problem [Sha90], in which a mutex owner with little funding could execute very slowly due to competition with other threads

[9] Under unmodified Mach, threads with equal priority are run round-robin; with lottery scheduling, it is possible for a thread to win several lotteries in a row. We believe that this ordering difference may affect locality, resulting in slightly improved cache and TLB behavior for this application under lottery scheduling.

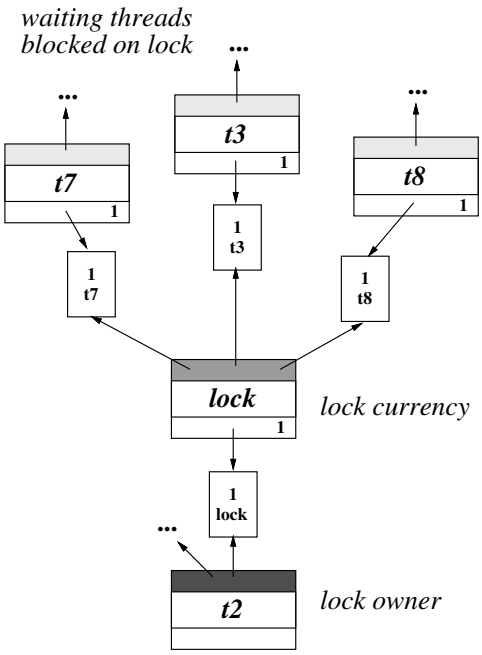


Figure 10: **Lock Funding.** Threads *t3*, *t7*, and *t8* are waiting to acquire a lottery-scheduled lock, and have transferred their funding to the lock currency. Thread *t2* currently holds the lock, and inherits the aggregate waiter funding through the backing ticket denominated in the lock currency. Instead of showing the backing tickets associated with each thread, shading is used to indicate relative funding levels.

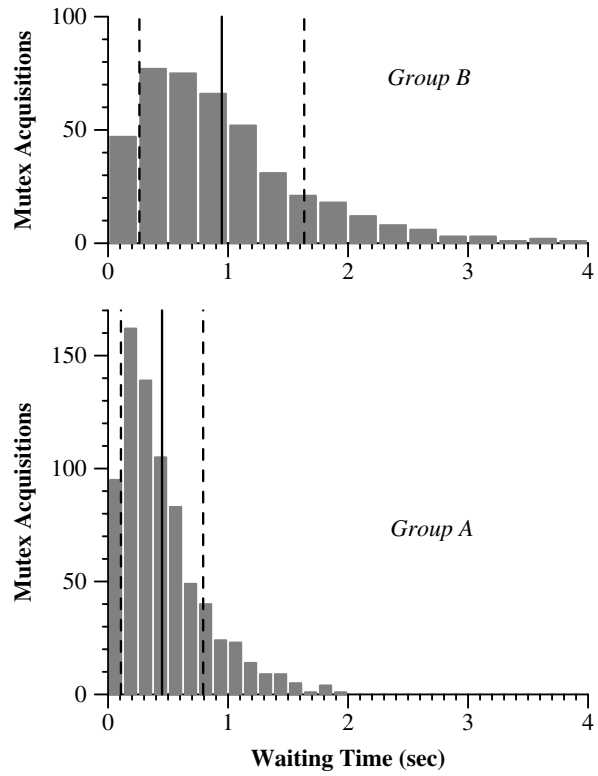


Figure 11: **Mutex Waiting Times.** Eight threads compete to acquire a lottery-scheduled mutex. The threads are divided into two groups (A, B) of four threads each, with the ticket allocation $A:B = 2:1$. For each histogram, the solid line indicates the mean (μ); the dashed lines indicate one standard deviation about the mean ($\mu \pm \sigma$). The ratio of average waiting times is $A:B = 1:2.11$; the mutex acquisition ratio is $1.80:1$.

for the processor, while a highly funded thread remains blocked on the mutex.

When a thread releases a lottery-scheduled mutex, it holds a lottery among the waiting threads to determine the next mutex owner. The thread then moves the mutex inheritance ticket to the winner, and yields the processor. The next thread to execute may be the selected waiter or some other thread that does not need the mutex; the normal processor lottery will choose fairly based on relative funding.

We have experimented with our mutex implementation using a synthetic multithreaded application in which n threads compete for the same mutex. Each thread repeatedly acquires the mutex, holds it for h milliseconds, releases the mutex, and computes for another t milliseconds. Figure 11 provides frequency histograms for a typical experiment with $n = 8$, $h = 50$, and $t = 50$. The eight threads were divided into two groups (A, B) of four threads each, with the ticket allocation $A:B = 2:1$. Over the entire two-minute experiment, group A threads acquired the mutex a total of 763 times, while group B threads completed 423 acquisitions, for a relative throughput ratio of $1.80:1$. The group A threads had a mean waiting time of $\mu = 450$ milliseconds , while the group B threads had a mean waiting time of $\mu = 948$ milliseconds, for a relative waiting time ratio of $1:2.11$. Thus, both throughput and response time closely tracked the specified $2:1$ ticket allocation.

6.2 Space-Shared Resources

Lotteries are useful for allocating indivisible time-shared resources, such as an entire processor. A variant of lottery scheduling can efficiently provide the same type of probabilistic proportional-share guarantees for finely divisible space-shared resources, such as memory. The basic idea is to use an *inverse lottery*, in which a "loser" is chosen to relinquish a unit of a resource that it holds. Conducting an inverse lottery is similar to holding a normal lottery, except that inverse probabilities are used. The probability p that a client holding t tickets will be selected by an inverse lottery with a total of n clients and T tickets is $p = \frac{1}{n-1}(1 - t/T)$. Thus, the more tickets a client has, the more likely it is to avoid having a unit of its resource revoked.[10]

For example, consider the problem of allocating a physical page to service a virtual memory page fault when all

[10]The $\frac{1}{n-1}$ factor is a normalization term which ensures that the client probabilities sum to unity.

physical pages are in use. A proportional-share policy based on inverse lotteries could choose a client from which to select a victim page with probability proportional to both $(1 - t/T)$ and the fraction of physical memory in use by that client.

6.3 Multiple Resources

Since rights for numerous resources are uniformly represented by lottery tickets, clients can use quantitative comparisons to make decisions involving tradeoffs between different resources. This raises some interesting questions regarding application funding policies in environments with multiple resources. For example, when does it make sense to shift funding from one resource to another? How frequently should funding allocations be reconsidered?

One way to abstract the evaluation of resource management options is to associate a separate *manager* thread with each application. A manager thread could be allocated a small fixed percentage (*e.g.*, 1%) of an application's overall funding, causing it to be periodically scheduled while limiting its overall resource consumption. For inverse lotteries, it may be appropriate to allow the losing client to execute a short manager code fragment in order to adjust funding levels. The system should supply default managers for most applications; sophisticated applications could define their own management strategies. We plan to explore these preliminary ideas and other alternatives for more complex environments with multiple resources.

7 Related Work

Conventional operating systems commonly employ a simple notion of *priority* in scheduling tasks. A task with higher priority is given absolute precedence over a task with lower priority. Priorities may be static, or they may be allowed to vary dynamically. Many sophisticated priority schemes are somewhat arbitrary, since priorities themselves are rarely meaningfully assigned [Dei90]. The ability to express priorities provides absolute, but extremely crude, control over scheduling, since resource rights do not vary smoothly with priorities. Conventional priority mechanisms are also inadequate for insulating the resource allocation policies of separate modules. Since priorities are absolute, it is difficult to compose or abstract inter-module priority relationships.

Fair share schedulers allocate resources so that users get fair machine shares over long periods of time [Hen84, Kay88]. These schedulers monitor CPU usage and dynamically adjust conventional priorities to push actual usage closer to entitled shares. However, the algorithms used by these systems are complex, requiring periodic usage updates, complicated dynamic priority adjustments, and administrative parameter setting to ensure fairness on a time scale of minutes. A technique also exists for achieving service rate objectives in systems that employ *decay-usage scheduling* by manipulating base priorities and various scheduler parameters [Hel93]. While this technique avoids the addition of feedback loops introduced by other fair share schedulers, it still assumes a fixed workload consisting of long-running compute-bound processes to ensure steady-state fairness at a time scale of minutes.

Microeconomic schedulers [Dre88, Fer88, Wal92] use auctions to allocate resources among clients that bid monetary funds. Funds encapsulate resource rights and serve as a form of priority. Both the *escalator algorithm* proposed for uniprocessor scheduling [Dre88] and the distributed *Spawn* system [Wal89, Wal92] rely upon auctions in which bidders increase their bids linearly over time. The *Spawn* system successfully allocated resources proportional to client funding in a network of heterogeneous workstations. However, experience with *Spawn* revealed that auction dynamics can be unexpectedly volatile. The overhead of bidding also limits the applicability of auctions to relatively coarse-grain tasks.

A market-based approach for memory allocation has also been developed to allow memory-intensive applications to optimize their memory consumption in a decentralized manner [Har92]. This scheme charges applications for both memory *leases* and I/O capacity, allowing application-specific tradeoffs to be made. However, unlike a true market, prices are not permitted to vary with demand, and ancillary parameters are introduced to restrict resource consumption [Che93].

The *statistical matching* technique for fair switching in the AN2 network exploits randomness to support frequent changes of bandwidth allocation [And93]. This work is similar to our proposed application of lottery scheduling to communication channels.

8 Conclusions

We have presented lottery scheduling, a novel mechanism that provides efficient and responsive control over the relative execution rates of computations. Lottery scheduling also facilitates modular resource management, and can be generalized to manage diverse resources. Since lottery scheduling is conceptually simple and easily implemented, it can be added to existing operating systems to provide greatly improved control over resource consumption rates. We are currently exploring various applications of lottery scheduling in interactive systems, including graphical user interface elements. We are also examining the use of lotteries for managing memory, virtual circuit bandwidth, and multiple resources.

Acknowledgements

We would like to thank Kavita Bala, Eric Brewer, Dawson Engler, Wilson Hsieh, Bob Gruber, Anthony Joseph, Frans Kaashoek, Ulana Legedza, Paige Parsons, Patrick

Sobalvarro, and Debby Wallach for their comments and assistance. Special thanks to Kavita for her invaluable help with Mach, and to Anthony for his patient critiques of several drafts. Thanks also to Jim Lipkis and the anonymous reviewers for their many helpful suggestions.

References

[Acc86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Conference*, June 1986.

[And93] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. "High-Speed Switch Scheduling for Local-Area Networks," *ACM Transactions on Computer Systems*, November 1993.

[Car90] D. G. Carta. "Two Fast Implementations of the 'Minimal Standard' Random Number Generator," *Communications of the ACM*, January 1990.

[Che93] D. R. Cheriton and K. Harty. "A Market Approach to Operating System Memory Allocation," Working Paper, Computer Science Department, Stanford University, June 1993.

[Com94] C. L. Compton and D. L. Tennenhouse. "Collaborative Load Shedding for Media-based Applications," *Proceedings of the International Conference on Multimedia Computing and Systems*, May 1994.

[Dei90] H. M. Deitel. *Operating Systems*, Addison-Wesley, 1990.

[Dre88] K. E. Drexler and M. S. Miller. "Incentive Engineering for Computational Resource Management" in *The Ecology of Computation*, B. Huberman (ed.), North-Holland, 1988.

[Dui90] D. Duis and J. Johnson. "Improving User-Interface Responsiveness Despite Performance Limitations," *Proceedings of the Thirty-Fifth IEEE Computer Society International Conference (COMPCON)*, March 1990.

[Fer88] D. Ferguson, Y. Yemini, and C. Nikolaou. "Microeconomic Algorithms for Load-Balancing in Distributed Computer Systems," *International Conference on Distributed Computer Systems*, 1988.

[Har92] K. Harty and D. R. Cheriton. "Application-Controlled Physical Memory using External Page-Cache Management," *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[Hel93] J. L. Hellerstein. "Achieving Service Rate Objectives with Decay Usage Scheduling," *IEEE Transactions on Software Engineering*, August 1993.

[Hen84] G. J. Henry. "The Fair Share Scheduler," *AT&T Bell Laboratories Technical Journal*, October 1984.

[Hog88] T. Hogg. Private communication (during *Spawn* system development), 1988.

[Kan89] G. Kane. *Mips RISC Architecture*, Prentice-Hall, 1989.

[Kay88] J. Kay and P. Lauder. "A Fair Share Scheduler," *Communications of the ACM*, January 1988.

[Loe92] K. Loepere. *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie Mellon University, 1992.

[Par88] S. K. Park and K. W. Miller. "Random Number Generators: Good Ones Are Hard to Find," *Communications of the ACM*, October 1988.

[Pre88] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1988.

[Sha90] L. Sha, R. Rajkumar, and J. P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, September 1990.

[Tri82] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, 1982.

[Wal89] C. A. Waldspurger. "A Distributed Computational Economy for Utilizing Idle Resources," Master's thesis, MIT, May 1989.

[Wal92] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. "Spawn: A Distributed Computational Economy," *IEEE Transactions on Software Engineering*, February 1992.

[Wei84] R. P. Weicker. "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, October 1984.

[Wei91] W. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger, and P. Wang. "Prelude: A System for Portable Parallel Software," Technical Report MIT/LCS/TR-519, MIT Lab for Computer Science, October 1991.

A Random Number Generator

This MIPS assembly-language code [Kan89] is a fast implementation of the Park-Miller pseudo-random number generator [Par88, Car90]. It uses the multiplicative linear congruential generator $S' = (A \times S) \bmod (2^{31} - 1)$, for $A = 16807$. The generator's ANSI C prototype is: `unsigned int fastrand(unsigned int s)`.

```
fastrand:
    move    $2, $4          | R2 = S (arg passed in R4)
    li      $8, 33614       | R8 = 2 × constant A
    multu   $2, $8          | HI, LO = A × S
    mflo    $9              | R9 = Q = bits 00..31 of A × S
    srl     $9, $9, 1
    mfhi    $10             | R10 = P = bits 32..63 of A × S
    addu    $2, $9, $10     | R2 = S' = P + Q
    bltz    $2, overflow    | handle overflow (rare)
    j       $31             | return (result in R2)

overflow:
    sll     $2, $2, 1       | zero bit 31 of S'
    srl     $2, $2, 1
    addiu   $2, 1           | increment S'
    j       $31             | return (result in R2)
```

Stride Scheduling: Deterministic Proportional-Share Resource Management

Carl A. Waldspurger * William E. Weihl *

Technical Memorandum MIT/LCS/TM-528
MIT Laboratory for Computer Science
Cambridge, MA 02139

June 22, 1995

Abstract

This paper presents *stride scheduling*, a deterministic scheduling technique that efficiently supports the same flexible resource management abstractions introduced by *lottery scheduling*. Compared to lottery scheduling, stride scheduling achieves significantly improved accuracy over relative throughput rates, with significantly lower response time variability. Stride scheduling implements proportional-share control over processor time and other resources by cross-applying elements of rate-based flow control algorithms designed for networks. We introduce new techniques to support dynamic changes and higher-level resource management abstractions. We also introduce a novel *hierarchical* stride scheduling algorithm that achieves better throughput accuracy and lower response time variability than prior schemes. Stride scheduling is evaluated using both simulations and prototypes implemented for the Linux kernel.

Keywords: dynamic scheduling, proportional-share resource allocation, rate-based service, service rate objectives

1 Introduction

Schedulers for multithreaded systems must multiplex scarce resources in order to service requests of varying importance. Accurate control over relative computation rates is required to achieve service rate objectives for users and applications. Such control is desirable across a broad spectrum of systems, including databases, media-based applications, and networks. Motivating examples include control over frame rates for competing video viewers, query rates for concurrent clients by databases and Web servers, and the consumption of shared resources by long-running computations.

Few general-purpose approaches have been proposed to support flexible, responsive control over service rates. We recently introduced *lottery scheduling*, a randomized resource allocation mechanism that provides efficient, responsive control over relative computation rates [Wal94]. Lottery scheduling implements *proportional-share* resource management – the resource consumption rates of active clients are proportional to the relative shares that they are allocated. Higher-level abstractions for flexible, modular resource management were also introduced with lottery scheduling, but they do not depend on the randomized implementation of proportional sharing.

In this paper we introduce *stride scheduling*, a deterministic scheduling technique that efficiently supports the same flexible resource management abstractions introduced by lottery scheduling. One contribution of our work is a cross-application and generalization of rate-based flow control algorithms designed for networks [Dem90, Zha91, ZhK91, Par93] to schedule other resources such as processor time. We present new techniques to support dynamic operations such as the modification of relative allocations and the transfer of resource rights between clients. We also introduce a novel *hierarchical* stride scheduling algorithm. Hierarchical stride

*E-mail: {carl, weihl}@lcs.mit.edu. World Wide Web: http://www.psg.lcs.mit.edu/. Prof. Weihl is currently supported by DEC while on sabbatical at DEC SRC. This research was also supported by ARPA under contract N00014-94-1-0985, by grants from AT&T and IBM, and by an equipment grant from DEC. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

scheduling is a recursive application of the basic technique that achieves better throughput accuracy and lower response time variability than previous schemes.

Simulation results demonstrate that, compared to lottery scheduling, stride scheduling achieves significantly improved accuracy over relative throughput rates, with significantly lower response time variability. In contrast to other deterministic schemes, stride scheduling efficiently supports operations that dynamically modify relative allocations and the number of clients competing for a resource. We have also implemented prototype stride schedulers for the Linux kernel, and found that they provide accurate control over both processor time and the relative network transmission rates of competing sockets.

In the next section, we present the core stride-scheduling mechanism. Section 3 describes extensions that support the resource management abstractions introduced with lottery scheduling. Section 4 introduces hierarchical stride scheduling. Simulation results with quantitative comparisons to lottery scheduling appear in Section 5. A discussion of our Linux prototypes and related implementation issues are presented in Section 6. In Section 7, we examine related work. Finally, we summarize our conclusions in Section 8.

2 Stride Scheduling

Stride scheduling is a deterministic allocation mechanism for time-shared resources. Resources are allocated in discrete time slices; we refer to the duration of a standard time slice as a *quantum*. Resource rights are represented by *tickets* – abstract, first-class objects that can be issued in different amounts and passed between clients.[1] Throughput rates for active clients are directly proportional to their ticket allocations. Thus, a client with twice as many tickets as another will receive twice as much of a resource in a given time interval. Client response times are inversely proportional to ticket allocations. Therefore a client with twice as many tickets as another will wait only half as long before acquiring a resource.

The throughput accuracy of a proportional-share scheduler can be characterized by measuring the difference between the specified and actual number of allocations that a client receives during a series of allocations. If a client has t tickets in a system with a total of T tickets, then its *specified* allocation after n_a consecutive allocations is $n_a \times t/T$. Due to quantization, it is typically impossible to achieve this ideal exactly. We define a client's *absolute error* as the absolute value of the difference between its specified and actual number of allocations. We define the pairwise *relative error* between clients c_i and c_j as the absolute error for the subsystem containing only c_i and c_j, where $T = t_i + t_j$, and n_a is the total number of allocations received by both clients.

While lottery scheduling offers probabilistic guarantees about throughput and response time, stride scheduling provides stronger deterministic guarantees. For lottery scheduling, after a series of n_a allocations, a client's expected relative error and expected absolute error are both $O(\sqrt{n_a})$. For stride scheduling, the relative error for any pair of clients is never greater than *one*, independent of n_a. However, for skewed ticket distributions it is still possible for a client to have $O(n_c)$ absolute error, where n_c is the number of clients. Nevertheless, stride scheduling is considerably more accurate than lottery scheduling, since its error does not grow with the number of allocations. In Section 4, we introduce a hierarchical variant of stride scheduling that provides a tighter $O(\lg n_c)$ bound on each client's absolute error.

This section first presents the basic stride-scheduling algorithm, and then introduces extensions that support dynamic client participation, dynamic modifications to ticket allocations, and nonuniform quanta.

2.1 Basic Algorithm

The core stride scheduling idea is to compute a representation of the time interval, or *stride*, that a client must wait between successive allocations. The client with the smallest stride will be scheduled most frequently. A client with half the stride of another will execute twice as quickly; a client with double the stride of another will execute twice as slowly. Strides are represented in virtual time units called *passes*, instead of units of real time such as seconds.

Three state variables are associated with each client: *tickets*, *stride*, and *pass*. The *tickets* field specifies the client's resource allocation, relative to other clients.

[1] In this paper we use the same terminology (*e.g.*, *tickets* and *currencies*) that we introduced for lottery scheduling [Wal94].

```
/* per-client state */
typedef struct {
  ...
  int tickets, stride, pass;
} *client_t;

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
client_t current;

/* initialize client with specified allocation */
void client_init(client_t c, queue_t q, int tickets)
{
  /* stride is inverse of tickets */
  c->tickets = tickets;
  c->stride = stride1 / tickets;
  c->pass = c->stride;

  /* join competition for resource */
  queue_insert(q, c);
}

/* proportional-share resource allocation */
void allocate(queue_t q)
{
  /* select client with minimum pass value */
  current = queue_remove_min(q);

  /* use resource for quantum */
  use_resource(current);

  /* compute next pass using stride */
  current->pass += current->stride;
  queue_insert(q, current);
}
```

Figure 1: **Basic Stride Scheduling Algorithm.** ANSI C code for scheduling a static set of clients. Queue manipulations can be performed in $O(\lg n_c)$ time by using an appropriate data structure.

The *stride* field is inversely proportional to *tickets*, and represents the interval between selections, measured in passes. The *pass* field represents the virtual time index for the client's next selection.

Performing a resource allocation is very simple: the client with the minimum *pass* is selected, and its *pass* is advanced by its *stride*. If more than one client has the same minimum pass value, then any of them may be selected. A reasonable deterministic approach is to use a consistent ordering to break ties, such as one defined by unique client identifiers.

Figure 1 lists ANSI C code for the basic stride scheduling algorithm. For simplicity, we assume a static set of clients with fixed ticket assignments. The stride scheduling state for each client must be initialized via *client_init()* before any allocations are performed by *allocate()*. These restrictions will be relaxed in subsequent sections to permit more dynamic behavior.

To accurately represent *stride* as the reciprocal of *tickets*, a floating-point representation could be used. We present a more efficient alternative that uses a high-precision fixed-point integer representation. This is easily implemented by multiplying the inverted ticket value by a large integer constant. We will refer to this constant as $stride_1$, since it represents the stride corresponding to the minimum ticket allocation of one.[2]

The cost of performing an allocation depends on the data structure used to implement the client queue. A priority queue can be used to implement *queue_remove_min()* and other queue operations in $O(\lg n_c)$ time or better, where n_c is the number of clients [Cor90]. A skip list could also provide expected $O(\lg n_c)$ time queue operations with low constant overhead [Pug90]. For small n_c or heavily skewed ticket distributions, a simple sorted list is likely to be most efficient in practice.

Figure 2 illustrates an example of stride scheduling. Three clients, A, B, and C, are competing for a time-shared resource with a 3 : 2 : 1 ticket ratio. For simplicity, a convenient $stride_1 = 6$ is used instead of a large number, yielding respective strides of 2, 3, and 6. The pass value of each client is plotted as a function of time. For each quantum, the client with the minimum pass value is selected, and its pass is advanced by its stride. Ties are

[2] Appendix A discusses the representation of strides in more detail.

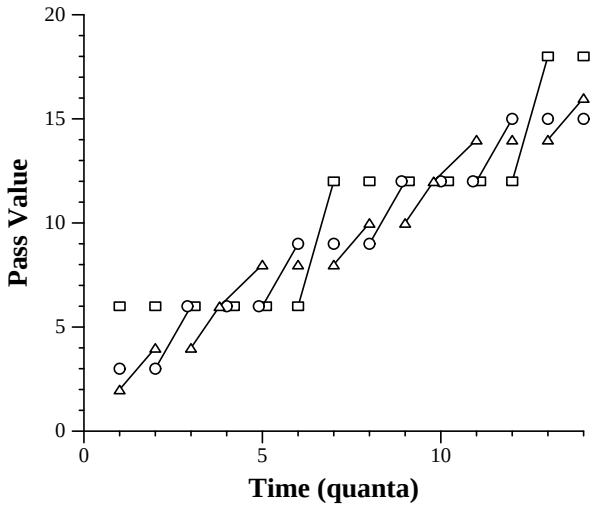


Figure 2: **Stride Scheduling Example.** Clients A (triangles), B (circles), and C (squares) have a $3:2:1$ ticket ratio. In this example, $stride_1$ = 6, yielding respective strides of 2, 3, and 6. For each quantum, the client with the minimum pass value is selected, and its pass is advanced by its stride.

broken using the arbitrary but consistent client ordering A, B, C.

2.2 Dynamic Client Participation

The algorithm presented in Figure 1 does not support dynamic changes in the number of clients competing for a resource. When clients are allowed to join and leave at any time, their state must be appropriately modified. Figure 3 extends the basic algorithm to efficiently handle dynamic changes.

A key extension is the addition of global variables that maintain aggregate information about the set of active clients. The *global_tickets* variable contains the total ticket sum for all active clients. The *global_pass* variable maintains the "current" pass for the scheduler. The *global_pass* advances at the rate of *global_stride* per quantum, where *global_stride* = $stride_1$ / *global_tickets*. Conceptually, the *global_pass* continuously advances at a smooth rate. This is implemented by invoking the *global_pass_update()* routine whenever the *global_pass* value is needed.[3]

A state variable is also associated with each client to store the remaining portion of its stride when a dynamic change occurs. The *remain* field represents the number of passes that are left before a client's next selection. When a client leaves the system, *remain* is computed as the difference between the client's *pass* and the *global_pass*. When a client rejoins the system, its *pass* value is recomputed by adding its *remain* value to the *global_pass*.

This mechanism handles situations involving either positive or negative error between the specified and actual number of allocations. If *remain* $<$ *stride*, then the client is effectively given credit when it rejoins for having previously waited for part of its stride without receiving a quantum. If *remain* $>$ *stride*, then the client is effectively penalized when it rejoins for having previously received a quantum without waiting for its entire stride.[4]

This approach makes an implicit assumption that a partial quantum now is equivalent to a partial quantum later. In general, this is a reasonable assumption, and resembles the treatment of nonuniform quanta that will be presented Section 2.4. However, it may not be appropriate if the total number of tickets competing for a resource varies significantly between the time that a client leaves and rejoins the system.

The time complexity for both the *client_leave()* and *client_join()* operations is $O(\lg n_c)$, where n_c is the number of clients. These operations are efficient because the stride scheduling state associated with distinct clients is completely independent; a change to one client does not require updates to any other clients. The $O(\lg n_c)$ cost results from the need to perform queue manipulations.

2.3 Dynamic Ticket Modifications

Additional support is needed to dynamically modify client ticket allocations. Figure 4 illustrates a dynamic allocation change, and Figure 5 lists ANSI C code for

[3] Due to the use of a fixed-point integer representation for strides, small quantization errors may accumulate slowly, causing *global_pass* to drift away from client pass values over a long period of time. This is unlikely to be a practical problem, since client pass values are recomputed using *global_pass* each time they leave and rejoin the system. However, this problem can be avoided by very infrequently resetting *global_pass* to the minimum pass value for the set of active clients.

[4] Several interesting alternatives could also be implemented. For example, a client could be given credit for some or all of the passes that elapse while it is inactive.

```
/* per-client state */
typedef struct {
  ...
  int tickets, stride, pass, remain;
} *client_t;

/* quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
client_t current;

/* global aggregate tickets, stride, pass */
int global_tickets, global_stride, global_pass;

/* update global pass based on elapsed real time */
void global_pass_update(void)
{
  static int last_update = 0;
  int elapsed;

  /* compute elapsed time, advance last_update */
  elapsed = time() - last_update;
  last_update += elapsed;

  /* advance global pass by quantum-adjusted stride */
  global_pass +=
    (global_stride * elapsed) / quantum;
}

/* update global tickets and stride to reflect change */
void global_tickets_update(int delta)
{
  global_tickets += delta;
  global_stride = stride1 / global_tickets;
}

/* initialize client with specified allocation */
void client_init(client_t c, int tickets)
{
  /* stride is inverse of tickets, whole stride remains */
  c->tickets = tickets;
  c->stride = stride1 / tickets;
  c->remain = c->stride;
}

/* join competition for resource */
void client_join(client_t c, queue_t q)
{
  /* compute pass for next allocation */
  global_pass_update();
  c->pass = global_pass + c->remain;

  /* add to queue */
  global_tickets_update(c->tickets);
  queue_insert(q, c);
}

/* leave competition for resource */
void client_leave(client_t c, queue_t q)
{
  /* compute remainder of current stride */
  global_pass_update();
  c->remain = c->pass - global_pass;

  /* remove from queue */
  global_tickets_update(-c->tickets);
  queue_remove(q, c);
}

/* proportional-share resource allocation */
void allocate(queue_t q)
{
  int elapsed;

  /* select client with minimum pass value */
  current = queue_remove_min(q);

  /* use resource, measuring elapsed real time */
  elapsed = use_resource(current);

  /* compute next pass using quantum-adjusted stride */
  current->pass +=
    (current->stride * elapsed) / quantum;
  queue_insert(q, current);
}
```

Figure 3: **Dynamic Stride Scheduling Algorithm.** ANSI C code for stride scheduling operations, including support for joining, leaving, and nonuniform quanta. Queue manipulations can be performed in $O(\lg n_c)$ time by using an appropriate data structure.

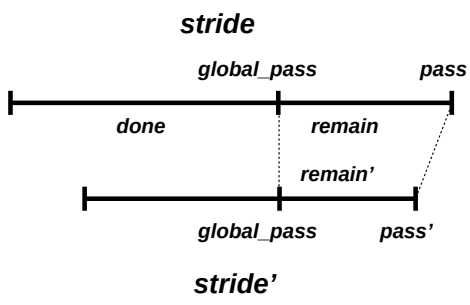


Figure 4: **Allocation Change.** Modifying a client's allocation from *tickets* to *tickets*$'$ requires only a constant-time recomputation of its *stride* and *pass*. The new *stride*$'$ is inversely proportional to *tickets*$'$. The new *pass*$'$ is determined by scaling *remain*, the remaining portion of the the current *stride*, by *stride*$'$ / *stride*.

```
/* dynamically modify client ticket allocation */
void client_modify(client_t c, queue_t q, int tickets)
{
  int remain, stride;

  /* leave queue for resource */
  client_leave(c, q);

  /* compute new stride */
  stride = stride1 / tickets;

  /* scale remaining passes to reflect change in stride */
  remain = (c->remain * stride) / c->stride;

  /* update client state */
  c->tickets = tickets;
  c->stride = stride;
  c->remain = remain;

  /* rejoin queue for resource */
  client_join(c, q);
}
```

Figure 5: **Dynamic Ticket Modification.** ANSI C code for dynamic modifications to client ticket allocations. Queue manipulations can be performed in $O(\lg n_c)$ time by using an appropriate data structure.

dynamically changing a client's ticket allocation. When a client's allocation is dynamically changed from *tickets* to *tickets*$'$, its *stride* and *pass* values must be recomputed. The new *stride*$'$ is computed as usual, inversely proportional to *tickets*$'$. To compute the new *pass*$'$, the remaining portion of the client's current *stride*, denoted by *remain*, is adjusted to reflect the new *stride*$'$. This is accomplished by scaling *remain* by *stride*$'$ / *stride*. In Figure 4, the client's ticket allocation is increased, so *pass* is decreased, compressing the time remaining until the client is next selected. If its allocation had decreased, then *pass* would have increased, expanding the time remaining until the client is next selected.

The *client_modify()* operation requires $O(\lg n_c)$ time, where n_c is the number of clients. As with dynamic changes to the number of clients, ticket allocation changes are efficient because the stride scheduling state associated with distinct clients is completely independent; the dominant cost is due to queue manipulations.

2.4 Nonuniform Quanta

With the basic stride scheduling algorithm presented in Figure 1, a client that does not consume its entire allocated quantum would receive less than its entitled share of a resource. Similarly, it may be possible for a client's usage to exceed a standard quantum in some situations. For example, under a non-preemptive scheduler, client run lengths can vary considerably.

Fortunately, fractional and variable-size quanta can easily be accommodated. When a client consumes a fraction f of its allocated time quantum, its *pass* should be advanced by $f \times$ *stride* instead of *stride*. If $f < 1$, then the client's *pass* will be increased less, and it will be scheduled sooner. If $f > 1$, then the client's *pass* will be increased more, and it will be scheduled later. The extended code listed in Figure 3 supports nonuniform quanta by effectively computing f as the *elapsed* resource usage time divided by a standard *quantum* in the same time units.

Another extension would permit clients to specify the quantum size that they require.[5] This could be implemented by associating an additional *quantum*$_c$ field with each client, and scaling each client's stride field by

[5] An alternative would be to allow a client to specify its scheduling period. Since a client's period and quantum are related by its relative resource share, specifying one quantity yields the other.

quantum$_c$ / *quantum*. Deviations from a client's specified quantum would still be handled as described above, with f redefined as the *elapsed* resource usage divided by the client-specific *quantum*$_c$.

3 Flexible Resource Management

Since stride scheduling enables low-overhead dynamic modifications, it can efficiently support the flexible resource management abstractions introduced with lottery scheduling [Wal94]. In this section, we explain how ticket transfers, ticket inflation, and ticket currencies can be implemented on top of a stride-based substrate for proportional sharing.

3.1 Ticket Transfers

A *ticket transfer* is an explicit transfer of tickets from one client to another. Ticket transfers are particularly useful when one client blocks waiting for another. For example, during a synchronous RPC, a client can loan its resource rights to the server computing on its behalf. A transfer of t tickets between clients A and B essentially consists of two dynamic ticket modifications. Using the code presented in Figure 5, these modifications are implemented by invoking *client_modify(A, q, A.tickets – t)* and *client_modify(B, q, B.tickets + t)*. When A transfers tickets to B, A's stride and pass will increase, while B's stride and pass will decrease.

A slight complication arises in the case of a complete ticket transfer; *i.e.*, when A transfers its entire ticket allocation to B. In this case, A's adjusted ticket value is zero, leading to an adjusted stride of infinity (division by zero). To circumvent this problem, we record the fraction of A's stride that is remaining at the time of the transfer, and then adjust that remaining fraction when A once again obtains tickets. This can easily be implemented by computing A's *remain* value at the time of the transfer, and deferring the computation of its stride and pass values until A receives a non-zero ticket allocation (perhaps via a return transfer from B).

3.2 Ticket Inflation

An alternative to explicit ticket transfers is *ticket inflation*, in which a client can escalate its resource rights by creating more tickets. Ticket inflation (or deflation) simply consists of a dynamic ticket modification for a client. Ticket inflation causes a client's stride and pass to decrease; deflation causes its stride and pass to increase.

Ticket inflation is useful among mutually trusting clients, since it permits resource rights to be reallocated without explicitly reshuffling tickets among clients. However, ticket inflation is also dangerous, since any client can monopolize a resource simply by creating a large number of tickets. In order to avoid the dangers of inflation while still exploiting its advantages, we introduced a *currency* abstraction for lottery scheduling [Wal94] that is loosely borrowed from economics.

3.3 Ticket Currencies

A *ticket currency* defines a resource management abstraction barrier that contains the effects of ticket inflation in a modular way. Tickets are denominated in currencies, allowing resource rights to be expressed in units that are local to each group of mutually trusting clients. Each currency is backed, or *funded*, by tickets that are denominated in more primitive currencies. Currency relationships may form an arbitrary acyclic graph, such as a hierarchy of currencies. The effects of inflation are locally contained by effectively maintaining an *exchange rate* between each local currency and a common *base* currency that is conserved. The currency abstraction is useful for flexibly naming, sharing, and protecting resource rights.

The currency abstraction introduced for lottery scheduling can also be used with stride scheduling. One implementation technique is to always immediately convert ticket values denominated in arbitrary currencies into units of the common base currency. Any changes to the value of a currency would then require dynamic modifications to all clients holding tickets denominated in that currency, or one derived from it.[6] Thus, the scope of any changes in currency values is limited to exactly those clients which are affected. Since currencies are used to group and isolate logical sets of clients, the impact of currency fluctuations will typically be very localized.

[6] An important exception is that changes to the number of tickets in the base currency do not require any modifications. This is because all stride scheduling state is computed from ticket values expressed in base units, and the state associated with distinct clients is independent.

4 Hierarchical Stride Scheduling

Stride scheduling guarantees that the *relative* throughput error for any pair of clients never exceeds a single quantum. However, depending on the distribution of tickets to clients, a large $O(n_c)$ *absolute* throughput error is still possible, where n_c is the number of clients.

For example, consider a set of 101 clients with a $100:1:\ldots:1$ ticket allocation. A schedule that minimizes absolute error and response time variability would alternate the 100-ticket client with each of the single-ticket clients. However, the standard stride algorithm schedules the clients in order, with the 100-ticket client receiving 100 quanta before any other client receives a single quantum. Thus, after 100 allocations, the intended allocation for the 100-ticket client is 50, while its actual allocation is 100, yielding a large absolute error of 50. This behavior is also exhibited by similar rate-based flow control algorithms for networks [Dem90, Zha91, ZhK91, Par93].

In this section we describe a novel hierarchical variant of stride scheduling that limits the absolute throughput error of any client to $O(\lg n_c)$ quanta. For the 101-client example described above, hierarchical stride scheduler simulations produced a maximum absolute error of only 4.5. Our algorithm also significantly reduces response time variability by aggregating clients to improve interleaving. Since it is common for systems to consist of a small number of high-throughput clients together with a large number of low-throughput clients, hierarchical stride scheduling represents a practical improvement over previous work.

4.1 Basic Algorithm

Hierarchical stride scheduling is essentially a recursive application of the basic stride scheduling algorithm. Individual clients are combined into groups with larger aggregate ticket allocations, and correspondingly smaller strides. An allocation is performed by invoking the normal stride scheduling algorithm first among groups, and then among individual clients within groups.

Although many different groupings are possible, we consider a balanced binary tree of groups. Each leaf node represents an individual client. Each internal node represents the group of clients (leaf nodes) that it covers, and contains their aggregate tickets, stride, and pass

```
/* binary tree node */
typedef struct node {
  ...
  struct node *left, *right, *parent;
  int tickets, stride, pass;
} *node_t;

/* quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
client_t current;

/* proportional-share resource allocation */
void allocate(node_t root)
{
  int elapsed;
  node_t n;

  /* traverse root-to-leaf path following min pass */
  for (n = root; !node_is_leaf(n); )
    if (n->left == NULL ||
        n->right->pass < n->left->pass)
      n = n->right;
    else
      n = n->left;

  /* use resource, measuring elapsed real time */
  current = n;
  elapsed = use_resource(current);

  /* update pass for each ancestor using its stride */
  for (n = current; n != NULL; n = n->parent)
    n->pass += (n->stride * elapsed) / quantum;
}
```

Figure 6: **Hierarchical Stride Scheduling Algorithm.** ANSI C code for hierachical stride scheduling with a static set of clients. The main data structure is a binary tree of nodes. Each node represents either a client (leaf) or a group (internal node) that summarizes aggregate information.

values. Thus, for an internal node, *tickets* is the total ticket sum for all of the clients that it covers, and *stride* = $stride_1$ / *tickets*. The *pass* value for an internal node is updated whenever the pass value for any of the clients that it covers is modified.

Figure 6 presents ANSI C code for the basic hierarchical stride scheduling algorithm. Each node has the normal tickets, stride, and pass scheduling state, as well as the usual tree links to its parent, left child, and right child. An allocation is performed by tracing a path from the root of the tree to a leaf, choosing the child with the smaller pass value at each level. Once the selected client has finished using the resource, its pass value is updated to reflect its usage. The client update is identical to that used in the dynamic stride algorithm that supports nonuniform quanta, listed in Figure 3. However, the hierarchical scheduler requires additional updates to each of the client's ancestors, following the leaf-to-root path formed by successive parent links.

Each client allocation can be viewed as a series of pairwise allocations among groups of clients at each level in the tree. The maximum error for each pairwise allocation is 1, and in the worst case, error can accumulate at each level. Thus, the maximum absolute error for the overall tree-based allocation is the height of the tree, which is $\lceil \lg n_c \rceil$, where n_c is the number of clients. Since the error for a pairwise A : B ratio is minimized when $A = B$, absolute error can be further reduced by carefully choosing client leaf positions to better balance the tree based on the number of tickets at each node.

4.2 Dynamic Modifications

Extending the basic hierarchical stride algorithm to support dynamic modifications requires a careful consideration of the effects of changes at each level in the tree. Figure 7 lists ANSI C code for performing a ticket modification that works for both clients and internal nodes. Changes to client ticket allocations essentially follow the same scaling and update rules used for normal stride scheduling, listed in Figure 5. The hierarchical scheduler requires additional updates to each of the client's ancestors, following the leaf-to-root path formed by successive parent links. Note that the root *pass* value used in Figure 7 effectively takes the place of the *global_pass* variable used in Figure 5; both represent the aggregate global scheduler pass.

```
/* dynamically modify node allocation by delta tickets */
void node_modify(node_t n, node_t root, int delta)
{
  int old_stride, remain;

  /* compute new tickets, stride */
  old_stride = n->stride;
  n->tickets += delta;
  n->stride = stride1 / n->tickets;

  /* done when reach root */
  if (n == root)
    return;

  /* scale remaining passes to reflect change in stride */
  remain = n->pass - root->pass;
  remain = (remain * n->stride) / old_stride;
  n->pass = root->pass + remain;

  /* propagate change to ancestors */
  node_modify(n->parent, root, delta);
}
```

Figure 7: **Dynamic Ticket Modification.** ANSI C code for dynamic modifications to client ticket allocations under hierarchical stride scheduling. A modification requires $O(\lg n_c)$ time to propagate changes.

Although not presented here, we have also developed operations to support dynamic client participation under hierarchical stride scheduling [Wal95]. As for *allocate()*, the time complexity for *client_join()* and *client_leave()* operations is $O(\lg n_c)$, where n_c is the number of clients.

5 Simulation Results

This section presents the results of several quantitative experiments designed to evaluate the effectiveness of stride scheduling. We examine the behavior of stride scheduling in both static and dynamic environments, and also test hierarchical stride scheduling. When stride scheduling is compared to lottery scheduling, we find that the stride-based approach provides more accurate control over relative throughput rates, with much lower variance in response times.

For example, Figure 8 presents the results of scheduling three clients with a $3:2:1$ ticket ratio for 100 allocations. The dashed lines represent the ideal allocations for each client. It is clear from Figure 8(a) that lottery scheduling exhibits significant variability at this time scale, due to the algorithm's inherent use of randomization. In contrast, Figure 8(b) indicates that the deterministic stride scheduler produces precise periodic behavior.

5.1 Throughput Accuracy

Under randomized lottery scheduling, the expected value for the absolute error between the specified and actual number of allocations for any set of clients is $O(\sqrt{n_a})$, where n_a is the number of allocations. This is because the number of lotteries won by a client has a binomial distribution. The probability p that a client holding t tickets will win a given lottery with a total of T tickets is simply $p = t/T$. After n_a identical lotteries, the expected number of wins w is $E[w] = n_a p$, with variance $\sigma_w^2 = n_a p(1-p)$.

Under deterministic stride scheduling, the relative error between the specified and actual number of allocations for any pair of clients never exceeds *one*, independent of n_a. This is because the only source of relative error is due to quantization.

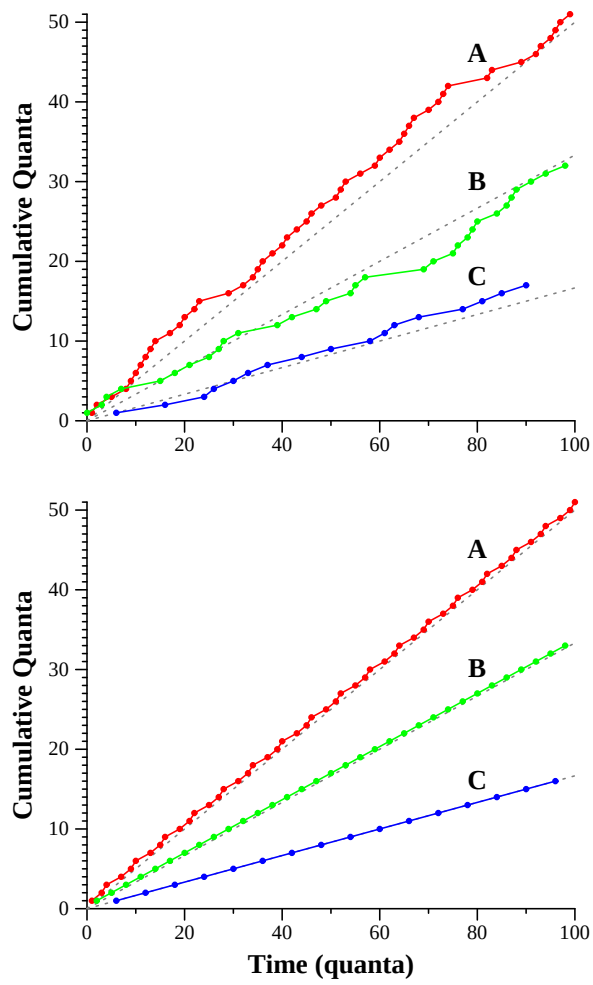


Figure 8: **Lottery vs. Stride Scheduling.** Simulation results for 100 allocations involving three clients, A, B, and C, with a $3:2:1$ allocation. The dashed lines represent ideal proportional-share behavior. (a) Allocation by randomized lottery scheduler shows significant variability. (b) Allocation by deterministic stride scheduler exhibits precise periodic behavior: A, B, A, A, B, C.

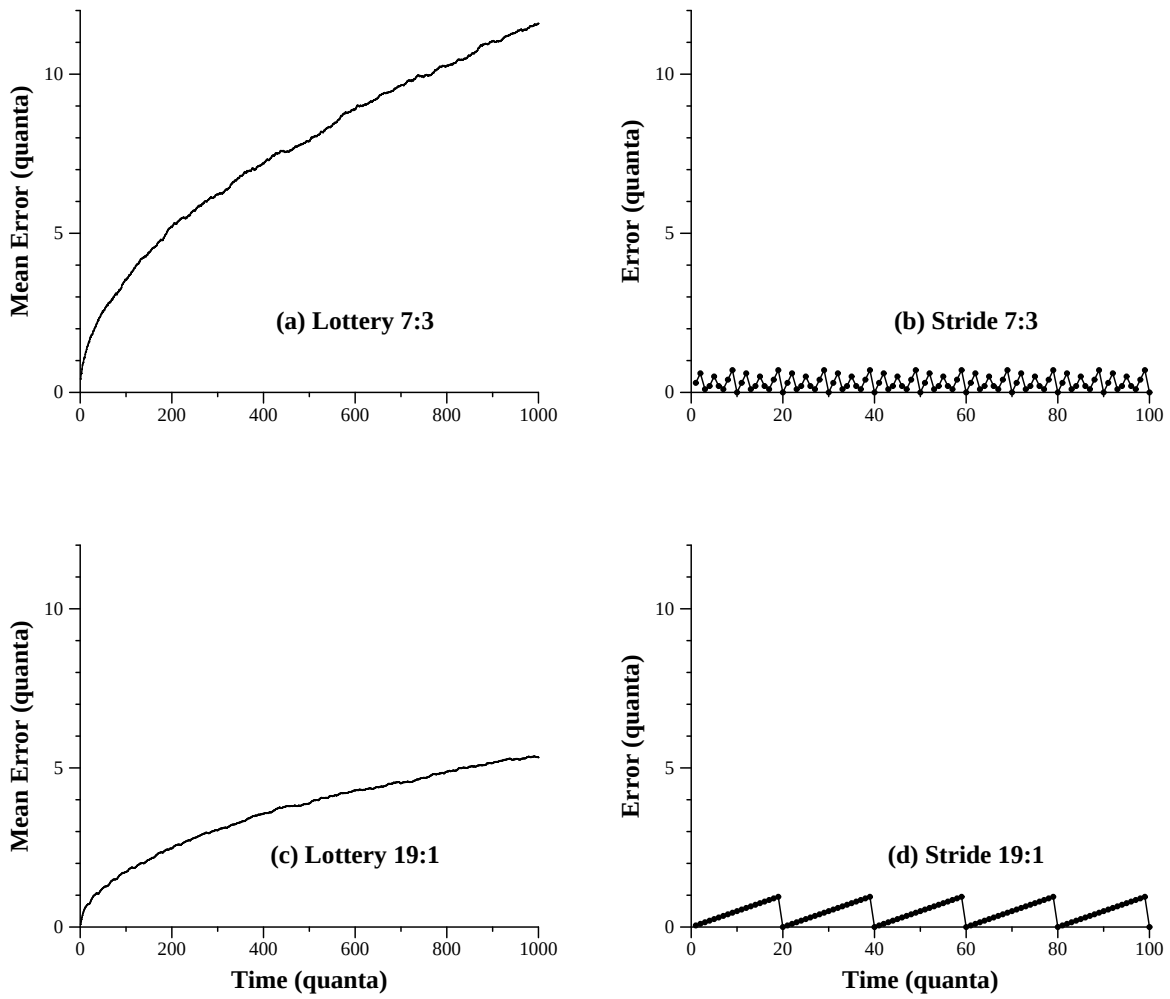


Figure 9: **Throughput Accuracy.** Simulation results for two clients with $7:3$ (top) and $19:1$ (bottom) ticket ratios over 1000 allocations. Only the first 100 quanta are shown for the stride scheduler, since its quantization error is deterministic and periodic. (a) Mean lottery scheduler error, averaged over 1000 separate $7:3$ runs. (b) Stride scheduler error for a single $7:3$ run. (c) Mean lottery scheduler error, averaged over 1000 separate $19:1$ runs. (d) Stride scheduler error for a single $19:1$ run.

Figure 9 plots the absolute error[7] that results from simulating two clients under both lottery scheduling and stride scheduling. The data depicted is representative of our simulation results over a large range of pairwise ratios. Figure 9(a) shows the mean error averaged over 1000 separate lottery scheduler runs with a $7:3$ ticket ratio. As expected, the error increases slowly with n_a, indicating that accuracy steadily improves when error is measured as a percentage of n_a. Figure 9(b) shows the error for a single stride scheduler run with the same $7:3$ ticket ratio. As expected, the error never exceeds a single quantum, and follows a deterministic pattern with period 10. The error drops to zero at the end of each complete period, corresponding to a precise $7:3$ allocation. Figures 9(c) and 9(d) present data for similar experiments involving a larger $19:1$ ticket ratio.

5.2 Dynamic Ticket Allocations

Figure 10 plots the absolute error that results from simulating two clients under both lottery scheduling and stride scheduling with rapidly-changing dynamic ticket allocations. This data is representative of simulation results over a large range of pairwise ratios and a variety of dynamic modification techniques. For easy comparison, the *average* dynamic ticket ratios are identical to the static ticket ratios used in Figure 9.

The notation $[A,B]$ indicates a random ticket allocation that is uniformly distributed from A to B. New, randomly-generated ticket allocations were dynamically assigned every other quantum. The *client_modify()* operation was executed for each change under stride scheduling; no special actions were necessary under lottery scheduling. To compute error values, specified allocations were determined incrementally. Each client's specified allocation was advanced by t/T on every quantum, where t is the client's current ticket allocation, and T is the current ticket total.

Figure 10(a) shows the mean error averaged over 1000 separate lottery scheduler runs with a $[2,12]:3$ ticket ratio. Despite the dynamic changes, the mean error is nearly the same as that measured for the static $7:3$ ratio depicted in Figure 9(a). Similarly, Figure 10(b) shows the error for a single stride scheduler run with the same dynamic $[2,12]:3$ ratio. The error never exceeds a single quantum, although it is much more erratic than the periodic pattern exhibited for the static $7:3$ ratio in Figure 9(b). Figures 10(c) and 10(d) present data for similar experiments involving a larger dynamic $190:[5,15]$ ratio. The results for this allocation are comparable to those measured for the static $19:1$ ticket ratio depicted in Figures 9(c) and 9(d).

Overall, the error measured under both lottery scheduling and stride scheduling is largely unaffected by dynamic ticket modifications. This suggests that both mechanisms are well-suited to dynamic environments. However, stride scheduling is clearly more accurate in both static and dynamic environments.

5.3 Response Time Variability

Another important performance metric is response time, which we measure as the elapsed time from a client's completion of one quantum up to and including its completion of another. Under randomized lottery scheduling, client response times have a geometric distribution. The expected number of lotteries n_a that a client must wait before its first win is $E[n_a] = 1/p$, with variance $\sigma^2_{n_a} = (1-p)/p^2$. Deterministic stride scheduling exhibits dramatically less response-time variability.

Figures 11 and 12 present client response time distributions under both lottery scheduling and stride scheduling. Figure 11 shows the response times that result from simulating two clients with a $7:3$ ticket ratio for one million allocations. The stride scheduler distributions are very tight, while the lottery scheduler distributions are geometric with long tails. For example, the client with the smaller allocation had a maximum response time of 4 quanta under stride scheduling, while the maximum response time under lottery scheduling was 39.

Figure 12 presents similar data for a larger $19:1$ ticket ratio. Although there is little difference in the response time distributions for the client with the larger allocation, the difference is enormous for the client with the smaller allocation. Under stride scheduling, virtually all of the response times were exactly 20 quanta. The lottery scheduler produced geometrically-distributed response times ranging from 1 to 194 quanta. In this case, the standard deviation of the stride scheduler's distribution is three orders of magnitude smaller than the standard deviation of the lottery scheduler's distribution.

[7] In this case the relative and absolute errors are identical, since there are only two clients.

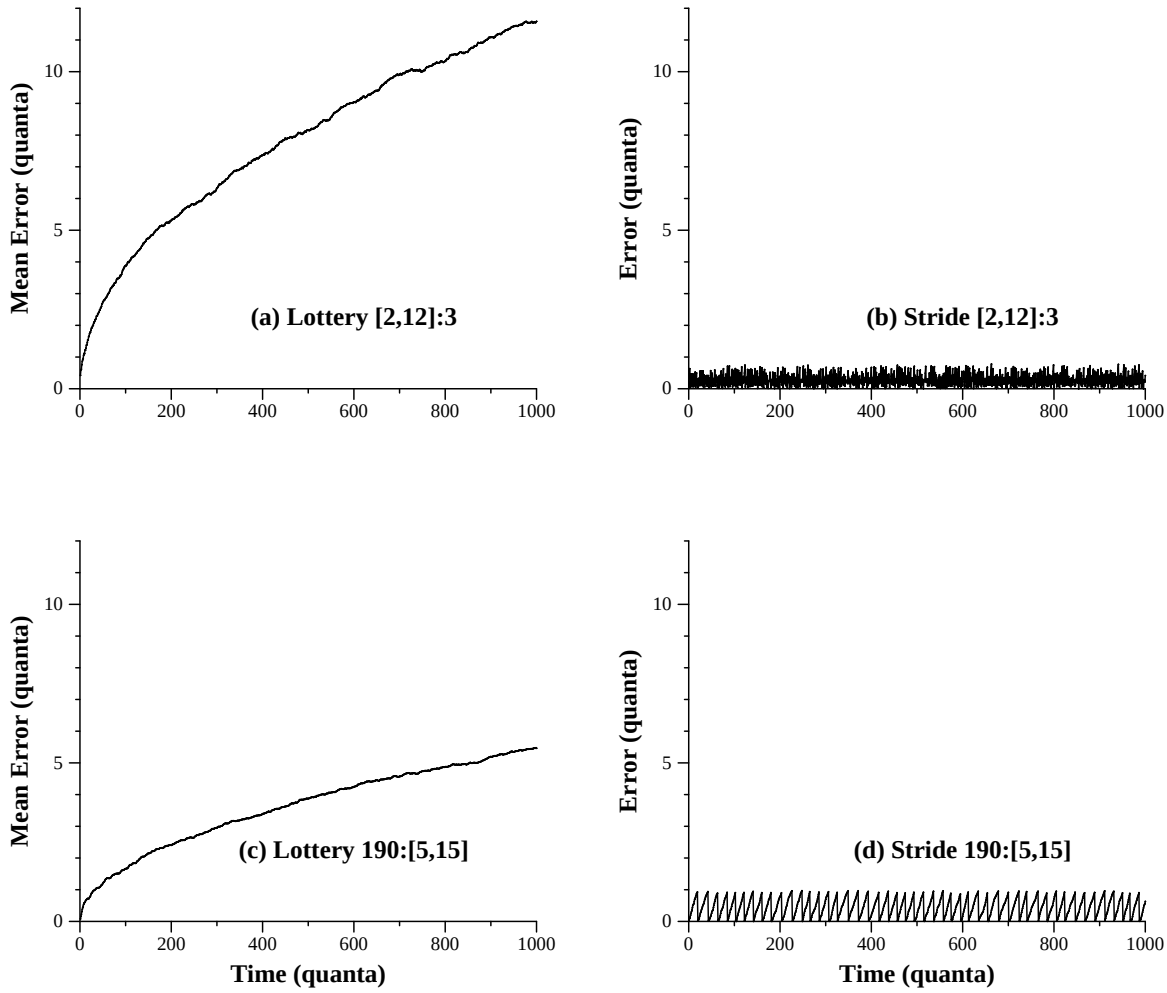


Figure 10: **Throughput Accuracy – Dynamic Allocations.** Simulation results for two clients with $[2,12]:3$ (top) and $190:[5,15]$ (bottom) ticket ratios over 1000 allocations. The notation $[A,B]$ indicates a random ticket allocation that is uniformly distributed from A to B. Random ticket allocations were dynamically updated every other quantum. (a) Mean lottery scheduler error, averaged over 1000 separate $[2,12]:3$ runs. (b) Stride scheduler error for a single $[2,12]:3$ run. (c) Mean lottery scheduler error, averaged over 1000 separate $190:[5,15]$ runs. (d) Stride scheduler error for a single $190:[5,15]$ run.

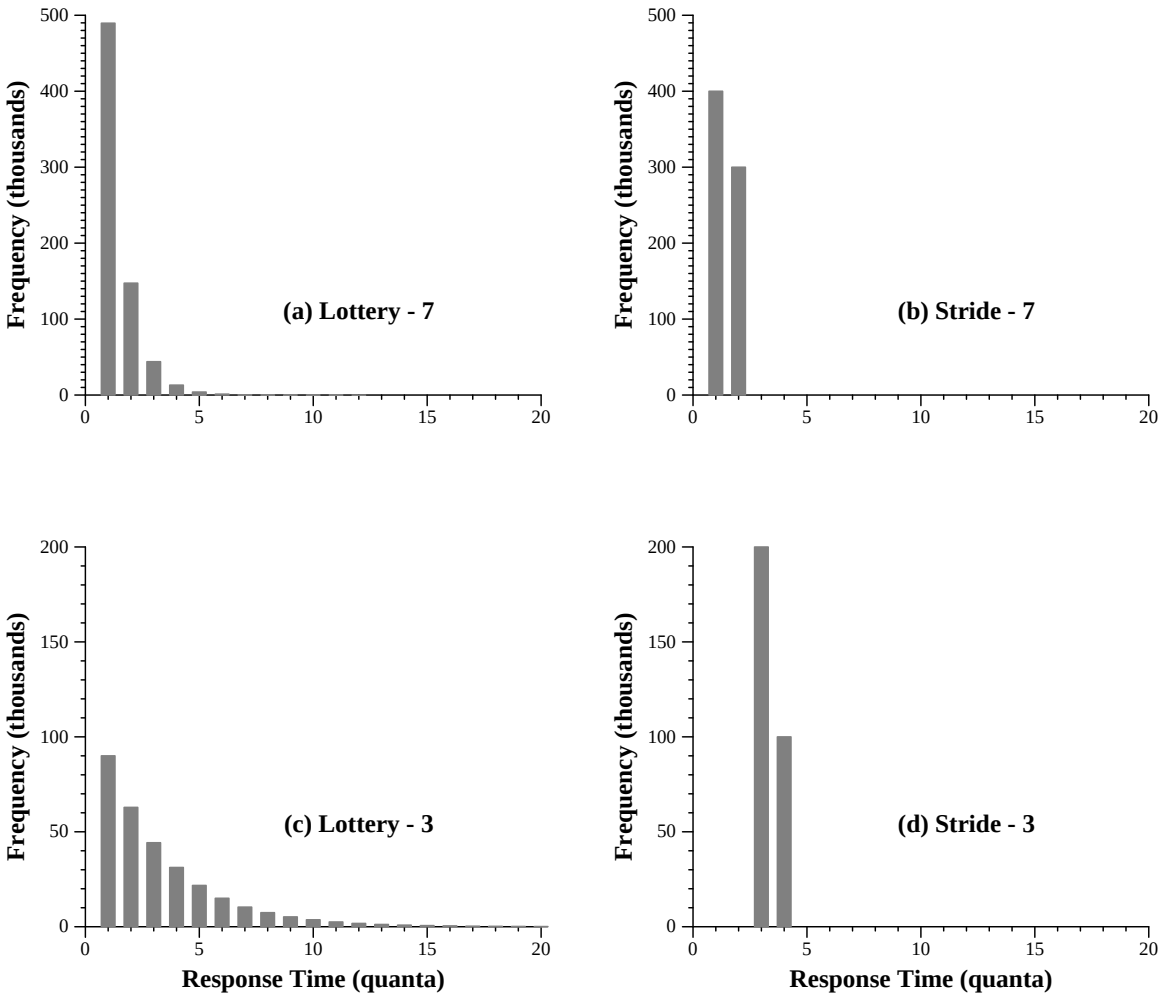


Figure 11: **Response Time Distribution.** Simulation results for two clients with a $7:3$ ticket ratio over one million allocations. (a) Client with 7 tickets under lottery scheduling: $\mu = 1.43$, $\sigma = 0.78$. (b) Client with 7 tickets under stride scheduling: $\mu = 1.43$, $\sigma = 0.49$. (c) Client with 3 tickets under lottery scheduling: $\mu = 3.33$, $\sigma = 2.79$. (d) Client with 3 tickets under stride scheduling: $\mu = 3.33$, $\sigma = 0.47$.

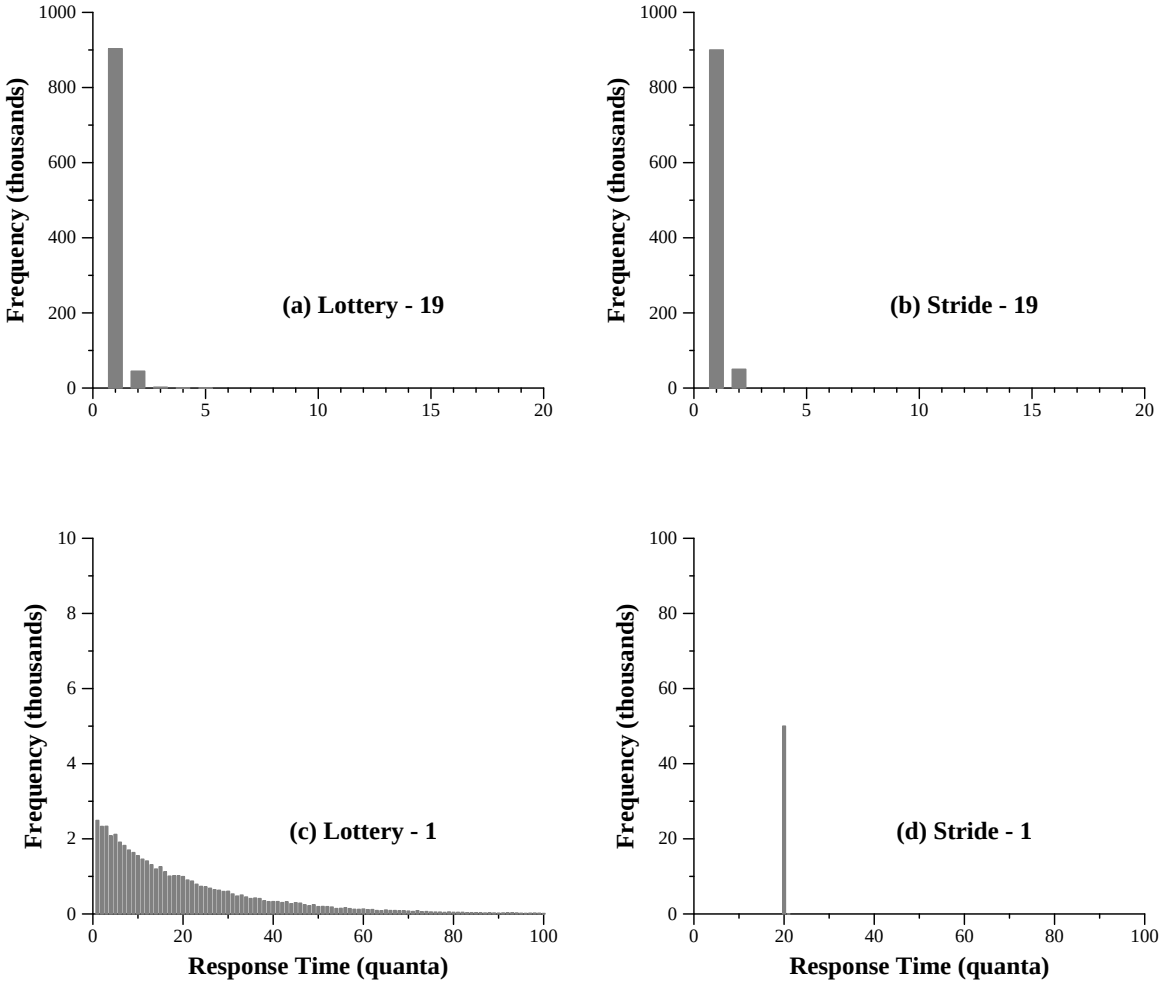


Figure 12: **Response Time Distribution.** Simulation results for two clients with a 19 : 1 ticket ratio over one million allocations. (a) Client with 19 tickets under lottery scheduling: $\mu = 1.05$, $\sigma = 0.24$. (b) Client with 19 tickets under stride scheduling: $\mu = 1.05$, $\sigma = 0.22$. (c) Client with 1 ticket under lottery scheduling: $\mu = 20.13$, $\sigma = 19.64$. (d) Client with 1 ticket under stride scheduling: $\mu = 20.00$, $\sigma = 0.01$.

5.4 Hierarchical Stride Scheduling

As discussed in Section 4, stride scheduling can produce an absolute error of $O(n_c)$ for skewed ticket distributions, where n_c is the number of clients. In contrast, hierarchical stride scheduling bounds the absolute error to $O(\lg n_c)$. As a result, response-time variability can be significantly reduced under hierarchical stride scheduling.

Figure 13 presents client response time distributions under both hierarchical stride scheduling and ordinary stride scheduling. Eight clients with a $7:1:\ldots:1$ ticket ratio were simulated for one million allocations. Excluding the very first allocation, the response time for each of the low-throughput clients was always 14, under both schedulers. Thus we only present response time distributions for the high-throughput client.

The ordinary stride scheduler runs the high-throughput client for 7 consecutive quanta, and then runs each of the low-throughput clients for one quantum. The hierarchical stride scheduler interleaves the clients, resulting in a tighter distribution. In this case, the standard deviation of the ordinary stride scheduler's distribution is more than twice as large as that for the hierarchical stride scheduler. We observed a maximum absolute error of 4 quanta for the high-throughput client under ordinary stride scheduling, and only 1.5 quanta under hierarchical stride scheduling.

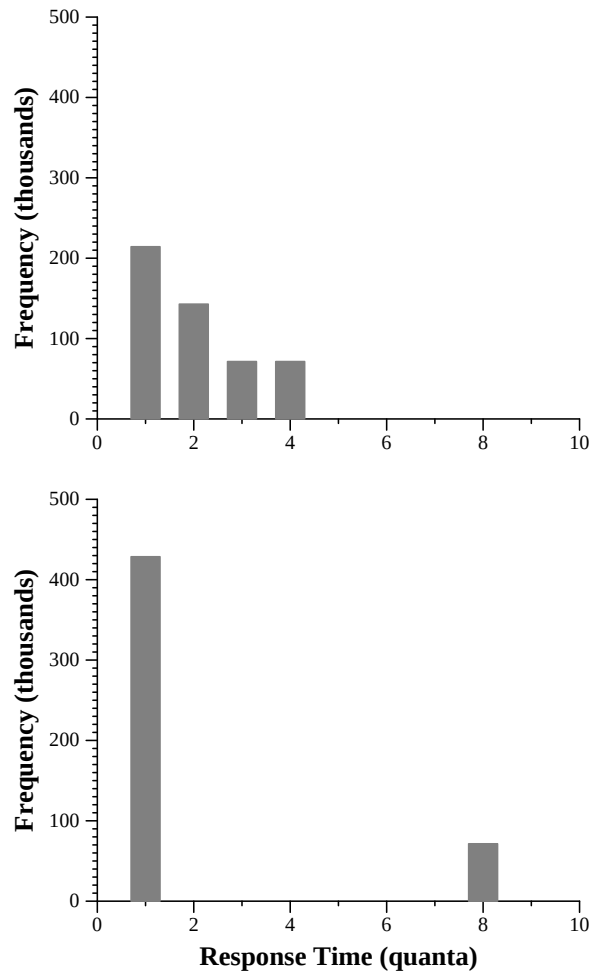


Figure 13: **Hierarchical Stride Scheduling.** Response time distributions for a simulation of eight clients with a $7:1:\ldots:1$ ticket ratio over one million allocations. Response times are shown only for the client with 7 tickets. (a) Hierarchical Stride Scheduler: $\mu = 2.00$, $\sigma = 1.07$. (b) Ordinary Stride Scheduler: $\mu = 2.00$, $\sigma = 2.45$.

6 Prototype Implementations

We implemented two prototype stride schedulers by modifying the Linux 1.1.50 kernel on a 25MHz i486-based IBM Thinkpad 350C. The first prototype enables proportional-share control over processor time, and the second enables proportional-share control over network transmission bandwidth.

6.1 Process Scheduler

The goal of our first prototype was to permit proportional-share allocation of processor time to control relative computation rates. We primarily changed the kernel code that handles process scheduling, switching from a conventional priority scheduler to a stride-based algorithm with a scheduling quantum of 100 milliseconds. Ticket allocations can be specified via a new

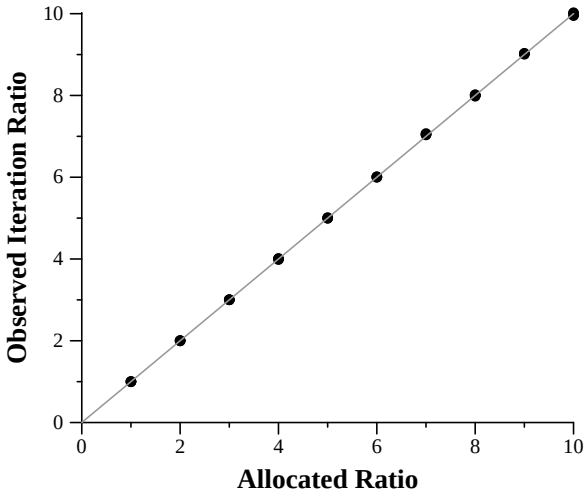


Figure 14: **CPU Rate Accuracy.** For each allocation ratio, the observed iteration ratio is plotted for each of three 30 second runs. The gray line indicates the ideal where the two ratios are identical. The observed ratios are within 1% of the ideal for all data points.

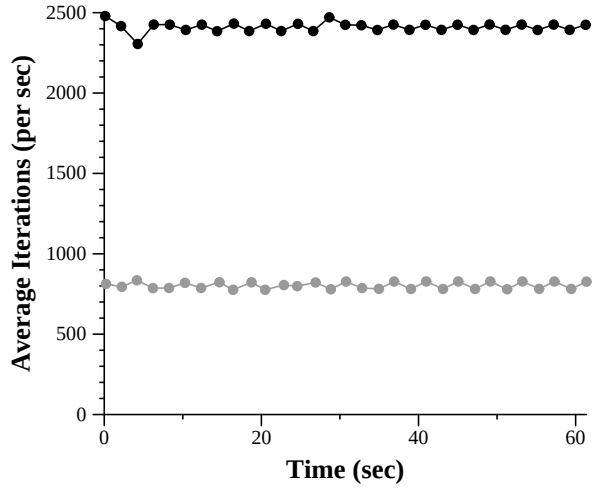


Figure 15: **CPU Fairness Over Time.** Two processes executing the compute-bound `arith` benchmark with a $3:1$ ticket allocation. Averaged over the entire run, the two processes executed 2409.18 and 802.89 iterations/sec., for an actual ratio of 3.001:1.

`stride_cpu_set_tickets()` system call. We did not implement support for higher-level abstractions such as ticket transfers and currencies. Fewer than 300 lines of source code were added or modified to implement our changes.

Our first experiment tested the accuracy with which our prototype could control the relative execution rate of computations. Each point plotted in Figure 14 indicates the relative execution rate that was observed for two processes running the compute-bound `arith` integer arithmetic benchmark [Byt91]. Three thirty-second runs were executed for each integral ratio between one and ten. In all cases, the observed ratios are within 1% of the ideal. We also ran experiments involving higher ratios, and found that the observed ratio for a $20:1$ allocation ranged from 19.94 to 20.04, and the observed ratio for a $50:1$ allocation ranged from 49.93 to 50.44.

Our next experiment examined the scheduler's behavior over shorter time intervals. Figure 15 plots average iteration counts over a series of 2-second time windows during a single 60 second execution with a $3:1$ allocation. The two processes remain close to their allocated ratios throughout the experiment. Note that if we used a 10 millisecond time quantum instead of the scheduler's 100 millisecond quantum, the same degree of fairness would be observed over a series of 200 millisecond time windows.

To assess the overhead imposed by our prototype stride scheduler, we ran performance tests consisting of concurrent `arith` benchmark processes. Overall, we found that the performance of our prototype was comparable to that of the standard Linux process scheduler. Compared to unmodified Linux, groups of 1, 2, 4, and 8 `arith` processes each completed fewer iterations under stride scheduling, but the difference was always less than 0.2%.

However, neither the standard Linux scheduler nor our prototype stride scheduler are particularly efficient. For example, the Linux scheduler performs a linear scan of all processes to find the one with the highest priority. Our prototype also performs a linear scan to find the process with the minimum pass; an $O(\lg n_c)$ time implementation would have required substantial changes to existing kernel code.

6.2 Network Device Scheduler

The goal of our second prototype was to permit proportional-share control over transmission bandwidth for network devices such as Ethernet and SLIP interfaces. Such control would be particularly useful for applications such as concurrent `ftp` file transfers, and concurrent `http` Web server replies. For example, many Web servers have relatively slow connections to the Internet, resulting in substantial delays for transfers of large objects such as graphical images. Given control over relative transmission rates, a Web server could provide different levels of service to concurrent clients. For example, tickets[8] could be issued by servers based upon the requesting user, machine, or domain. Commercial servers could even sell tickets to clients demanding faster service.

We primarily changed the kernel code that handles generic network device queueing. This involved switching from conventional FIFO queueing to stride-based queueing that respects per-socket ticket allocations. Ticket allocations can be specified via a new `SO_TICKETS` option to the `setsockopt()` system call. Although not implemented in our prototype, a more complete system should also consider additional forms of admission control to manage other system resources, such as network buffers. Fewer than 300 lines of source code were added or modified to implement our changes.

Our first experiment tested the prototype's ability to control relative network transmission rates on a local area network. We used the `ttcp` network test program[9] [TTC91] to transfer fabricated buffers from an IBM Thinkpad 350C running our modified Linux kernel, to a DECStation 5000/133 running Ultrix. Both machines were on the same physical subnet, connected via a 10Mbps Ethernet that also carried network traffic for other users.

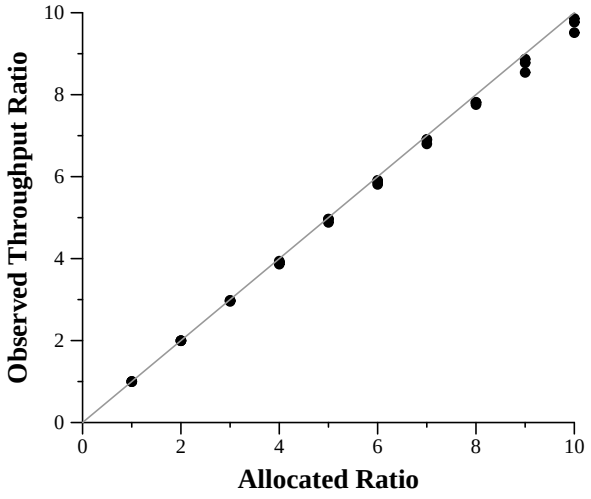


Figure 16: **Ethernet UDP Rate Accuracy.** For each allocation ratio, the observed data transmission ratio is plotted for each of three runs. The gray line indicates the ideal where the two ratios are identical. The observed ratios are within 5% of the ideal for all data points.

Each point plotted in Figure 16 indicates the relative UDP data transmission rate that was observed for two processes running the `ttcp` benchmark. Each experiment started with both processes on the sending machine attempting to transmit 4K buffers, each containing 8Kbytes of data, for a total 32Mbyte transfer. As soon as one process finished sending its data, it terminated the other process via a Unix signal. Metrics were recorded on the receiving machine to capture end-to-end application throughput. The observed ratios are very accurate; all data points are within 5% of the ideal. For larger ticket ratios, the observed throughput ratio is slightly lower than the specified allocation. For example, a $20:1$ allocation resulted in actual throughput ratios ranging from $18.51:1$ to $18.77:1$.

To assess the overhead imposed by our prototype, we ran performance tests consisting of concurrent `ttcp` benchmark processes. Overall, we found that the performance of our prototype was comparable to that of standard Linux. Although the prototype increases the length

[8] To be included with `http` requests, tickets would require an external data representation. If security is a concern, cryptographic techniques could be employed to prevent forgery and theft.

[9] We made a few minor modifications to the standard `ttcp` benchmark. Other than extensions to specify ticket allocations and facilitate coordinated timing, we also decreased the value of a hard-coded delay constant. This constant is used to temporarily put a transmitting process to sleep when it is unable to write to a socket due to a lack of buffer space (`ENOBUFS`). Without this modification, the observed throughput ratios were consistently lower than specified allocations, with significant differences for large ratios. With the larger delay constant, we believe that the low-throughput client is able to continue sending packets while the high-throughput client is sleeping, distorting the intended throughput ratio. Of course, changing the kernel interface to signal a process when more buffer space becomes available would probably be preferable to polling.

of the critical path for sending a network packet, we were unable to observe any significant difference between unmodified Linux and stride scheduling. We believe that the small additional overhead of stride scheduling was masked by the variability of external network traffic from other users; individual differences were in the range of $\pm 5\%$.

7 Related Work

We independently developed stride scheduling as a deterministic alternative to the randomized selection aspect of lottery scheduling [Wal94]. We then discovered that the core allocation algorithm used in stride scheduling is nearly identical to elements of rate-based flow-control algorithms designed for packet-switched networks [Dem90, Zha91, ZhK91, Par93]. Despite the relevance of this networking research, to the best of our knowledge it has not been discussed in the processor scheduling literature. In this section we discuss a variety of related scheduling work, including rate-based network flow control, deterministic proportional-share schedulers, priority schedulers, real-time schedulers, and microeconomic schedulers.

7.1 Rate-Based Network Flow Control

Our basic stride scheduling algorithm is very similar to Zhang's *VirtualClock* algorithm for packet-switched networks [Zha91]. In this scheme, a network switch orders packets to be forwarded through outgoing links. Every packet belongs to a client data stream, and each stream has an associated bandwidth reservation. A *virtual clock* is assigned to each stream, and each of its packets is stamped with its current virtual time upon arrival. With each arrival, the virtual clock advances by a *virtual tick* that is inversely proportional to the stream's reserved data rate. Using our stride-oriented terminology, a virtual tick is analogous to a *stride*, and a virtual clock is analogous to a *pass* value.

The VirtualClock algorithm is closely related to the *weighted fair queueing (WFQ)* algorithm developed by Demers, Keshav, and Shenker [Dem90], and Parekh and Gallager's equivalent *packet-by-packet generalized processor sharing (PGPS)* algorithm [Par93]. One difference that distinguishes WFQ and PGPS from VirtualClock is that they effectively maintain a *global* virtual clock. Arriving packets are stamped with their stream's virtual tick plus the *maximum* of their stream's virtual clock and the global virtual clock. Without this modification, an inactive stream can later monopolize a link as its virtual clock caught up to those of active streams; such behavior is possible under the VirtualClock algorithm [Par93].

Our stride scheduler's use of a *global_pass* variable is based on the global virtual clock employed by WFQ/PGPS, which follows an update rule that produces a smoothly varying global virtual time. Before we became aware of the WFQ/PGPS work, we used a simpler *global_pass* update rule: *global_pass* was set to the pass value of the client that currently owns the resource. To see the difference between these approaches, consider the set of minimum pass values over time in Figure 2. Although the *average* pass value increase per quantum is 1, the actual increases occur in non-uniform steps. We adopted the smoother WFQ/PGPS virtual time rule to improve the accuracy of pass updates associated with dynamic modifications.

To the best of our knowledge, our work on stride scheduling is the first cross-application of rate-based network flow control algorithms to scheduling other resources such as processor time. New techniques were required to support dynamic changes and higher-level abstractions such as ticket transfers and currencies. Our hierarchical stride scheduling algorithm is a novel recursive application of the basic technique that exhibits improved throughput accuracy and reduced response time variability compared to prior schemes.

7.2 Proportional-Share Schedulers

Several other deterministic approaches have recently been proposed for proportional-share processor scheduling [Fon95, Mah95, Sto95]. However, all require expensive operations to transform client state in response to dynamic changes. This makes them less attractive than stride scheduling for supporting dynamic or distributed environments. Moreover, although each algorithm is explicitly compared to lottery scheduling, none provides efficient support for the flexible resource management abstractions introduced with lottery scheduling.

Stoica and Abdel-Wahab have devised an interesting scheduler using a deterministic generator that employs

a bit-reversed counter in place of the random number generator used by lottery scheduling [Sto95]. Their algorithm results in an absolute error for throughput that is $O(\lg n_a)$, where n_a is the number of allocations. Allocations can be performed efficiently in $O(\lg n_c)$ time using a tree-based data structure, where n_c is the number of clients. However, dynamic modifications to the set of active clients or their allocations require executing a relatively complex "restart" operation with $O(n_c)$ time complexity. Also, no support is provided for fractional or nonuniform quanta.

Maheshwari has developed a deterministic *charge-based* proportional-share scheduler [Mah95]. Loosely based on an analogy to digitized line drawing, this scheme has a maximum relative throughput error of one quantum, and also supports fractional quanta. Although efficient in many cases, allocation has a worst-case $O(n_c)$ time complexity, where n_c is the number of clients. Dynamic modifications require executing a "refund" operation with $O(n_c)$ time complexity.

Fong and Squillante have introduced a general scheduling approach called *time-function scheduling (TFS)* [Fon95]. TFS is intended to provide differential treatment of job classes, where specific throughput ratios are specified across classes, while jobs within each class are scheduled in a FCFS manner. Time functions are used to compute dynamic job priorities as a function of the time each job has spent waiting since it was placed on the run queue. Linear functions result in proportional sharing: a job's value is equal to its waiting time multipled by its job-class slope, plus a job-class constant. An allocation is performed by selecting the job with the maximum time-function value. A naive implementation would be very expensive, but since jobs are grouped into classes, allocation can be performed in $O(n)$ time, where n is the number of distinct classes. If time-function values are updated infrequently compared to the scheduling quantum, then a priority queue can be used to reduce the allocation cost to $O(\lg n)$, with an $O(n \lg n)$ cost to rebuild the queue after each update.

When Fong and Squillante compared TFS to lottery scheduling, they found that although throughput accuracy was comparable, the waiting time variance of low-throughput tasks was often several orders of magnitude larger under lottery scheduling. This observation is consistent with our simulation results involving response time, presented in Section 5. TFS also offers the potential to specify performance goals that are more general than proportional sharing. However, when proportional sharing is the goal, stride scheduling has advantages in terms of efficiency and accuracy.

7.3 Priority Schedulers

Conventional operating systems typically employ *priority* schemes for scheduling processes [Dei90, Tan92]. Priority schedulers are not designed to provide proportional-share control over relative computation rates, and are often ad-hoc. Even popular priority-based approaches such as *decay-usage scheduling* are poorly understood, despite the fact that they are employed by numerous operating systems, including Unix [Hel93].

Fair share schedulers allocate resources so that users get fair machine shares over long periods of time [Hen84, Kay88, Hel93]. These schedulers are layered on top of conventional priority schedulers, and dynamically adjust priorities to push actual usage closer to entitled shares. The algorithms used by these systems are generally complex, requiring periodic usage monitoring, complicated dynamic priority adjustments, and administrative parameter setting to ensure fairness on a time scale of minutes.

7.4 Real-Time Schedulers

Real-time schedulers are designed for time-critical systems [Bur91]. In these systems, which include many aerospace and military applications, timing requirements impose absolute deadlines that must be met to ensure correctness and safety; a missed deadline may have dire consequences. One of the most widely used techniques in real-time systems is *rate-monotonic scheduling*, in which priorities are statically assigned as a monotonic function of the rate of periodic tasks [Liu73, Sha91]. The importance of a task is not reflected in its priority; tasks with shorter periods are simply assigned higher priorities. Bounds on total processor utilization (ranging from 69% to nearly 100%, depending on various assumptions) ensure that rate monotonic scheduling will meet all task deadlines. Another popular technique is *earliest deadline scheduling*, which always schedules the task with the closest deadline first. The earliest deadline approach permits high processor

utilization, but has increased runtime overhead due to the use of dynamic priorities; the task with the nearest deadline varies over time.

In general, real-time schedulers depend upon very restrictive assumptions, including precise static knowledge of task execution times and prohibitions on task interactions. In addition, limitations are placed on processor utilization, and even transient overloads are disallowed. In contrast, the proportional-share model used by stride scheduling and lottery scheduling is designed for more general-purpose environments. Task allocations degrade gracefully in overload situations, and active tasks proportionally benefit from extra resources when some allocations are not fully utilized. These properties facilitate adaptive applications that can respond to changes in resource availability.

Mercer, Savage, and Tokuda recently introduced a higher-level *processor capacity reserve* abstraction [Mer94] for measuring and controlling processor usage in a microkernel system with an underlying real-time scheduler. Reserves can be passed across protection boundaries during interprocess communication, with an effect similar to our use of ticket transfers. While this approach works well for many multimedia applications, its reliance on resource reservations and admission control is still more restrictive than the general-purpose model that we advocate.

7.5 Microeconomic Schedulers

Microeconomic schedulers are based on metaphors to resource allocation in real economic systems. *Money* encapsulates resource rights, and a *price* mechanism is used to allocate resources. Several microeconomic schedulers [Dre88, Mil88, Fer88, Fer89, Wal89, Wal92, Wel93] use auctions to determine prices and allocate resources among clients that bid monetary funds. Both the *escalator algorithm* proposed for uniprocessor scheduling [Dre88] and the distributed *Spawn* system [Wal92] rely upon auctions in which bidders increase their bids linearly over time. Since auction dynamics can be unexpectedly volatile, auction-based approaches sometimes fail to achieve resource allocations that are proportional to client funding. The overhead of bidding also limits the applicability of auctions to relatively coarse-grained tasks. Other market-based approaches that do not rely upon auctions have also been applied to managing processor and memory resources [Ell75, Har92, Che93].

Stride scheduling and lottery scheduling are compatible with a market-based resource management philosophy. Our mechanisms for proportional sharing provide a convenient substrate for pricing individual time-shared resources in a computational economy. For example, tickets are analogous to monetary income streams, and the number of tickets competing for a resource can be viewed as its price. Our currency abstraction for flexible resource management is also loosely borrowed from economics.

8 Conclusions

We have presented stride scheduling, a deterministic technique that provides accurate control over relative computation rates. Stride scheduling also efficiently supports the same flexible, modular resource management abstractions introduced by lottery scheduling. Compared to lottery scheduling, stride scheduling achieves significantly improved accuracy over relative throughput rates, with significantly less response time variability. However, lottery scheduling is conceptually simpler than stride scheduling. For example, stride scheduling requires careful state updates for dynamic changes, while lottery scheduling is effectively stateless.

The core allocation mechanism used by stride scheduling is based on rate-based flow-control algorithms for networks. One contribution of this paper is a cross-application of these algorithms to the domain of processor scheduling. New techniques were developed to support dynamic modifications to client allocations and resource right transfers between clients. We also introduced a new hierarchical stride scheduling algorithm that exhibits improved throughput accuracy and lower response time variability compared to prior schemes.

Acknowledgements

We would like to thank Kavita Bala, Dawson Engler, Paige Parsons, and Lyle Ramshaw for their many helpful comments. Thanks to Tom Rodeheffer for suggesting the connection between our work and rate-based flow-control algorithms in the networking literature. Special thanks to Paige for her help with the visual presentation of stride scheduling.

References

[Bur91] A. Burns. "Scheduling Hard Real-Time Systems: A Review," *Software Engineering Journal*, May 1991.

[Byt91] Byte Unix Benchmarks, Version 3, 1991. Available via Usenet and anonymous `ftp` from many locations, including `gatekeeper.dec.com`.

[Che93] D. R. Cheriton and K. Harty. "A Market Approach to Operating System Memory Allocation," Working Paper, Computer Science Department, Stanford University, June 1993.

[Cor90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[Dei90] H. M. Deitel. *Operating Systems*, Addison-Wesley, 1990.

[Dem90] A. Demers, S. Kehav, and S. Shenker. "Analysis and Simulation of a Fair Queueing Algorithm," *Internetworking: Research and Experience*, September 1990.

[Dre88] K. E. Drexler and M. S. Miller. "Incentive Engineering for Computational Resource Management" in *The Ecology of Computation*, B. Huberman (ed.), North-Holland, 1988.

[Ell75] C. M. Ellison. "The Utah TENEX Scheduler," *Proceedings of the IEEE*, June 1975.

[Fer88] D. Ferguson, Y. Yemini, and C. Nikolaou. "Microeconomic Algorithms for Load-Balancing in Distributed Computer Systems," *International Conference on Distributed Computer Systems*, 1988.

[Fer89] D. F. Ferguson. "The Application of Microeconomics to the Design of Resource Allocation and Control Algorithms," Ph.D. thesis, Columbia University, 1989.

[Fon95] L. L. Fong and M. S. Squillante. "Time-Functions: A General Approach to Controllable Resource Management," Working Draft, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, March 1995.

[Har92] K. Harty and D. R. Cheriton. "Application-Controlled Physical Memory using External Page-Cache Management," *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[Hel93] J. L. Hellerstein. "Achieving Service Rate Objectives with Decay Usage Scheduling," *IEEE Transactions on Software Engineering*, August 1993.

[Hen84] G. J. Henry. "The Fair Share Scheduler," *AT&T Bell Laboratories Technical Journal*, October 1984.

[Kay88] J. Kay and P. Lauder. "A Fair Share Scheduler," *Communications of the ACM*, January 1988.

[Liu73] C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, January 1973.

[Mah95] U. Maheshwari. "Charge-Based Proportional Scheduling," Working Draft, MIT Laboratory for Computer Science, Cambridge, MA, February 1995.

[Mer94] C. W. Mercer, S. Savage, and H. Tokuda. "Processor Capacity Reserves: Operating System Support for Multimedia Applications," *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[Mil88] M. S. Miller and K. E. Drexler. "Markets and Computation: Agoric Open Systems," in *The Ecology of Computation*, B. Huberman (ed.), North-Holland, 1988.

[Par93] A. K. Parekh and R. G. Gallager. "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case," *IEEE/ACM Transactions on Networking*, June 1993.

[Pug90] W. Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM*, June 1990.

[Sha91] L. Sha, M. H. Klein, and J. B. Goodenough. "Rate Monotonic Analysis for Real-Time Systems," in *Foundations of Real-Time Computing: Scheduling and Resource Management*, A. M. van Tilborg and G. M. Koob (eds.), Kluwer Academic Publishers, 1991.

[Sto95] I. Stoica and H. Abdel-Wahab. "A New Approach to Implement Proportional Share Resource Allocation," Technical Report 95-05, Department of Computer Science, Old Dominion University, Norfolk, VA, April 1995.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems*, Prentice Hall, 1992.

[TTC91] TTCP benchmarking tool. SGI version, 1991. Originally developed at the US Army Ballistics Research Lab (BRL). Available via anonymous `ftp` from many locations, including `ftp.sgi.com`.

[Wal89] C. A. Waldspurger. "A Distributed Computational Economy for Utilizing Idle Resources," Master's thesis, MIT, May 1989.

[Wal92] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. "Spawn: A Distributed Computational Economy," *IEEE Transactions on Software Engineering*, February 1992.

[Wal94] C. A. Waldspurger and W. E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[Wal95] C. A. Waldspurger. "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," Ph.D. thesis, MIT, 1995 (to appear).

[Wel93] M. P. Wellman. "A Market-Oriented Programming Environment and its Application to Distributed Multicommodity Flow Problems," *Journal of Artificial Intelligence Research*, August 1993.

[Zha91] L. Zhang. "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," *ACM Transactions on Computer Systems*, May 1991.

[ZhK91] H. Zhang and S. Kehav. "Comparison of Rate-Based Service Disciplines," *Proceedings of SIGCOMM '91*, September 1991.

A Fixed-Point Stride Representation

The precision of relative rates that can be achieved depends on both the value of $stride_1$ and the relative ratios of client ticket allocations. For example, with $stride_1 = 2^{20}$, and a maximum ticket allocation of 2^{10} tickets, ratios are represented with 10 bits of precision. Thus, ratios close to unity resulting from allocations that differ by only one part per thousand, such as $1001:1000$, can be supported.

Since $stride_1$ is a large integer, *stride* values will also be large for clients with small allocations. Since *pass* values are monotonically increasing, they will eventually overflow the machine word size after a large number of allocations. For a machine with 64-bit integers, this is not a practical problem. For example, with $stride_1 = 2^{20}$ and a worst-case client *tickets* = 1, approximately 2^{44} allocations can be performed before overflow occurs. At one allocation per millisecond, centuries of real time would elapse before an overflow.

For a machine with 32-bit integers, the *pass* values associated with all clients can be adjusted by subtracting the minimum pass value from all clients whenever an overflow is detected. Alternatively, such adjustments can periodically be made after a fixed number of allocations. For example, with $stride_1 = 2^{20}$, a conservative adjustment period would be a few thousand allocations. Perhaps the most straightforward approach is to simply use a 64-bit integer type if one is available. Our prototype implementation makes use of the 64-bit "`long long`" integer type provided by the GNU C compiler.

Parallel Database Systems: The Future of High Performance Database Systems

Highly parallel database systems are beginning to displace traditional mainframe computers for the largest database and transaction processing tasks. The success of these systems refutes a 1983 paper predicting the demise of database machines [3]. Ten years ago the future of highly parallel database machines seemed gloomy, even to their staunchest advocates. Most database machine research had focused on specialized, often trendy, hardware such as CCD memories, bubble memories, head-per-track disks, and optical disks. None of these technologies fulfilled their promises; so there was a sense that conventional CPUs, electronic RAM, and moving-head magnetic disks would dominate the scene for many years to come. At that time, disk throughput was predicted to double while processor speeds were predicted to increase by much larger factors. Consequently, critics predicted that multiprocessor systems would soon be I/O limited unless a solution to the I/O bottleneck was found.

While these predictions were fairly accurate about the future of hardware, the critics were certainly wrong about the overall future of parallel database systems. Over the last decade Teradata, Tandem, and a host of startup companies have successfully developed and marketed highly parallel machines.

David DeWitt and Jim Gray

Parallel

Why have parallel database systems become more than a research curiosity? One explanation is the widespread adoption of the relational data model. In 1983 relational database systems were just appearing in the marketplace; today they dominate it. Relational queries are ideally suited to parallel execution; they consist of uniform operations applied to uniform streams of data. Each operator produces a new relation, so the operators can be composed into highly parallel dataflow graphs. By streaming the output of one operator into the input of another operator, the two operators can work in series giving *pipelined parallelism*. By partitioning the input data among multiple processors and memories, an operator can often be split into many independent operators each working on a part of the data. This partitioned data and execution gives *partitioned parallelism* (Figure 1).

The dataflow approach to database system design needs a message-based client-server operating system to interconnect the parallel processes executing the relational operators. This in turn requires a high-speed network to interconnect the parallel processors. Such facilities seemed exotic a decade ago, but now they are the mainstream of computer architecture. The client-server paradigm using high-speed LANs is the basis for most PC, workstation, and workgroup software. Those same client-server mechanisms are an excellent basis for distributed database technology.

Mainframe designers have found it difficult to build machines powerful enough to meet the CPU and I/O demands of relational databases serving large numbers of simultaneous users or searching terabyte databases. Meanwhile, multiprocessors based on fast and inexpensive microprocessors have become widely available from vendors including Encore, Intel, NCR, nCUBE, Sequent, Tandem, Teradata, and Thinking Machines. These machines provide more total power than their mainframe coun-

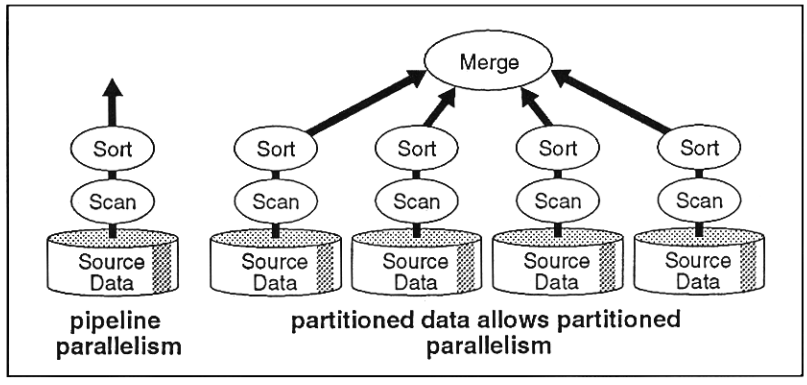


Figure 1.
The dataflow approach to relational operators gives both pipelined and partitioned parallelism. Relational data operators take relations (uniform sets of records) as input and produce relations as outputs. This allows them to be composed in dataflow graphs that allow *pipeline parallelism* (left) in which the computation of one operator proceeds in parallel with another, and *partitioned parallelism* in which operators (sort and scan in the diagram at the right) are replicated for each data source, and the replicas execute in parallel.

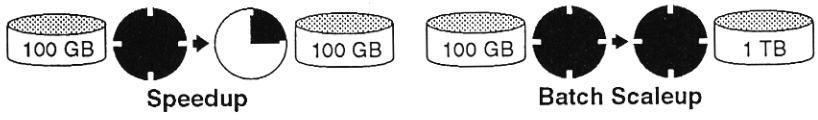


Figure 2.
Speedup and Scaleup. A speedup design performs a one-hour job four times faster when run on a four-times larger system. A scaleup design runs a ten-times bigger job is done in the same time by a ten-times bigger system.

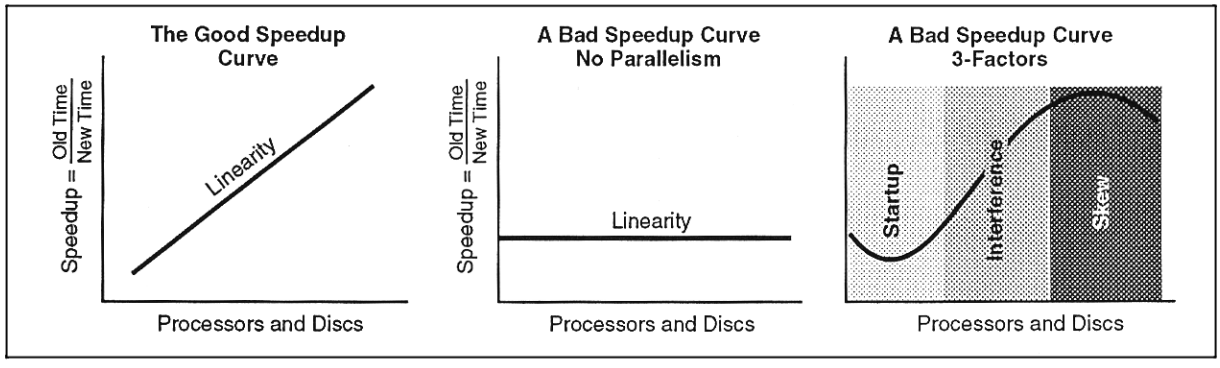


Figure 3.
Good and bad speedup curves. The standard speedup curves. The left curve is the ideal. The middle graph shows no speedup as hardware is added. The right curve shows the three threats to parallelism. Initial startup costs may dominate at first. As the number of processes increase, interference can increase. Ultimately, the job is divided so finely, that the variance in service times (skew) causes a slowdown.

terparts at a lower price. Their modular architectures enable systems to grow incrementally, adding MIPS, memory, and disks either to speedup the processing of a given job, or to scaleup the system to process a larger job in the same time.

In retrospect, special-purpose database machines have indeed failed; but parallel database systems are a big success. The successful parallel database systems are built from conventional processors, memories, and disks. They have emerged as major consumers of highly parallel architectures, and are in an excellent position to exploit massive numbers of fast-cheap commodity disks, processors, and memories promised by current technology forecasts.

A consensus on parallel and distributed database system architecture has emerged. This architecture is based on a *shared-nothing* hardware design [29] in which processors communicate with one another only by sending messages via an interconnection network. In such systems, tuples of each relation in the database are *partitioned (declustered)* across disk storage units[1] attached directly to each processor. Partitioning allows multiple processors to scan large relations in parallel without needing any exotic I/O devices. Such architectures were pioneered by Teradata in the late 1970s and by several research projects. This design is now used by Teradata, Tandem, NCR, OracleNCUBE, and several other products currently under development. The research community has also embraced this shared-nothing dataflow architecture in systems like Arbre, Bubba, and Gamma.

The remainder of this article is organized as follows: The next section describes the basic architectural concepts used in these parallel database systems. This is followed by a brief presentation of the unique features of the Teradata, Tandem, Bubba, and Gamma systems in the following section, entitled "The State of the Art." Several areas for future research are described in "Future Directions and Research Problems" prior to the conclusion of this article.

Basic Techniques for Parallel Database Machine Implementation

Parallelism Goals and Metrics: Speedup and Scaleup

The ideal parallel system demonstrates two key properties: (1) *linear speedup:* Twice as much hardware can perform the task in half the elapsed time, and (2) *linear scaleup:* Twice as much hardware can perform twice as large a task in the same elapsed time (see Figures 2 and 3).

More formally, given a fixed job run on a small system, and then run on a larger system, the *speedup* given by the larger system is measured as:

$$Speedup = \frac{small_system_elapsed_time}{big_system_elapsed_time}$$

Speedup is said to be linear, if an N-times large or more expensive system yields a speedup of N.

Speedup holds the problem size constant, and grows the system. Scaleup measures the ability to grow both the system and the problem. *Scaleup* is defined as the ability of an N-times larger system to perform an N-times larger job in the same elapsed time as the original system. The scaleup metric is:

$$Scaleup = \frac{small_system_elapsed_time_on_small_problem}{big_system_elapsed_time_on_big_problem}$$

If this scaleup equation evaluates to 1, then the scaleup is said to be linear[2]. There are two distinct kinds of scaleup, batch and transactional. If the job consists of performing many small independent requests submitted by many clients and operating on a shared database, then scaleup consists of N-times as many clients, submitting N-times as many requests against an N-times larger database. This is the scaleup typically found in transaction processing systems and timesharing systems. This form of scaleup is used by the Transaction Processing Performance Council to scaleup their transaction processing benchmarks [36]. Consequently, it is called *transaction scaleup.* Transaction scaleup is ideally suited to parallel systems since each transaction is typically a small independent job that can be run on a separate processor.

A second form of scaleup, called *batch scaleup,* arises when the scaleup task is presented as a single large job. This is typical of database queries and is also typical of scientific simulations. In these cases, scaleup consists of using an N-times larger computer to solve an N-times larger problem. For database systems batch scaleup translates to the same query on an N-times larger database; for scientific problems, batch scaleup translates to the same calculation on an N-times finer grid or on an N-times longer simulation.

The generic barriers to linear speedup and linear scaleup are the triple threats of:

startup: The time needed to start a parallel operation. If thousands of processes must be started, this can easily dominate the actual computation time.

interference: The slowdown each new process imposes on all others when accessing shared resources.

skew: As the number of parallel steps increases, the average size of each step decreases, but the variance can well exceed the mean. The service time of a job is the service time of the slowest step of the job. When the variance dominates the mean, increased parallelism improves elapsed time only slightly.

The subsection "A Parallel

[1]The term disk is used here as a shorthand for disk or other nonvolatile storage media. As the decade proceeds, nonvolatile electronic storage or some other media may replace or augment disks.

[2]The execution cost of some operators increases super-linearly. For example, the cost of sorting n-tuples increases as $nlog(n)$. When n is in the billions, scaling up by a factor of a thousand, causes $nlog(n)$ to increase by 3,000. This 30% deviation from linearity in a three-orders-of-magnitude scaleup justifies the use of the term *near-linear* scaleup.

Dataflow Approach to SQL Software" describes several basic techniques widely used in the design of shared-nothing parallel database machines to overcome these barriers. These techniques often achieve linear speedup and scaleup on relational operators.

Hardware Architecture, the Trend to Shared-Nothing Machines

The ideal database machine would have a single infinitely fast processor with an infinite memory with infinite bandwidth—and it would be infinitely cheap (free). Given such a machine, there would be no need for speedup, scaleup, or parallelism. Unfortunately, technology is not delivering such machines—but it is coming close. Technology is promising to deliver fast one-chip processors, fast high-capacity disks, and high-capacity electronic RAM. It also promises that each of these devices will be very inexpensive by today's standards, costing only hundreds of dollars each.

So, the challenge is to build an infinitely fast processor out of infinitely many processors of finite speed, and to build an infinitely large memory with infinite memory bandwidth from infinitely many storage units of finite speed. This sounds trivial mathematically; but in practice, when a new processor is added to most computer designs, it slows every other computer down just a little bit. If this slowdown (interference) is 1%, then the maximum speedup is 37 and a 1,000-processor system has 4% of the effective power of a single-processor system.

How can we build scaleable multiprocessor systems? Stonebraker suggested the following simple taxonomy for the spectrum of designs (see Figures 4 and 5) [29][3]:

shared-memory: All processors share direct access to a common global memory and to all disks. The IBM/370, Digital VAX, and Sequent Symmetry multiprocessors typify this design.

shared-disks: Each processor has a private memory but has direct access to all disks. The IBM Sysplex and original Digital VAXcluster typify this design.

shared-nothing: Each memory and disk is owned by some processor that acts as a server for that data. Mass storage in such an architecture is distributed among the processors by connecting one or more disks. The Teradata, Tandem, and nCUBE machines typify this design.

Shared-nothing architectures minimize interference by minimizing resource sharing. They also exploit commodity processors and memory without needing an incredibly powerful interconnection network. As Figure 5 suggests, the other architectures move large quantities of data through the interconnection network. The shared-nothing design moves only questions and answers through the network. Raw memory accesses and raw disk accesses are performed locally in a processor, and only the filtered (reduced) data is passed to the client program. This allows a more scaleable design by minimizing traffic on the interconnection network.

Shared-nothing characterizes the database systems being used by Teradata [33], Gamma [8, 9], Tandem [32], Bubba [1], Arbre [21], and nCUBE [13]. Significantly, Digital's VAXcluster has evolved to this design. DOS and UNIX workgroup systems from 3com, Borland, Digital, HP, Novell, Microsoft, and Sun also adopt a shared-nothing client-server architecture.

The actual interconnection networks used by these systems vary enormously. Teradata employs a redundant tree-structured communication network. Tandem uses a three-level duplexed network, two levels within a cluster, and rings connecting the clusters. Arbre, Bubba, and Gamma are independent of the underlying interconnection network, requiring only that the network allow any two nodes to communicate with one another. Gamma operates on an Intel Hypercube. The Arbre prototype was implemented using IBM 4381 processors connected to one another in a point-to-point network. Workgroup systems are currently making a transition from Ethernet to higher speed local networks.

The main advantage of shared-nothing multiprocessors is that they can be scaled up to hundreds and probably thousands of processors that do not interfere with one another. Teradata, Tandem, and Intel have each shipped systems with more than 200 processors. Intel is implementing a 2,000-node hypercube. The largest shared-memory multiprocessors currently available are limited to about 32 processors.

These shared-nothing architectures achieve near-linear speedups and scaleups on complex relational queries and on on-line transaction processing workloads [9, 10, 32]. Given such results, database machine designers see little justification for the hardware and software complexity associated with shared-memory and shared-disk designs.

Shared-memory and shared-disk systems do not scale well on database applications. Interference is a major problem for shared-memory multiprocessors. The interconnection network must have the bandwidth of the sum of the processors and disks. It is difficult to build such networks that can scale to thousands of nodes. To reduce network traffic and to minimize latency, each processor is given a large private cache. Measurements of shared-memory multiprocessors running database workloads show that loading and flushing these caches considerably degrades processor performance [35]. As parallelism increases, interference on shared resources limits performance. Multiprocessor systems

[3]Single Instruction stream, Multiple Data stream (SIMD) machines such as ILLIAC IV and its derivatives like MASSPAR and the "old" Connection Machine are ignored here because to date they have few successes in the database area. SIMD machines seem to have application in simulation, pattern matching, and mathematical search, but they do not seem to be appropriate for the multiuser, I/O intensive, and dataflow paradigm of database systems.

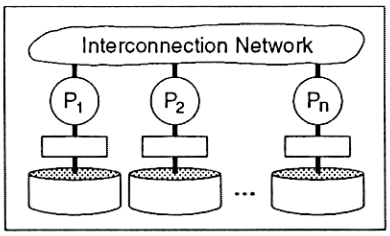


Figure 4.
The basic shared-nothing design. Each processor has a private memory and one or more disks. Processors communicate via a high-speed interconnect network. Teradata, Tandem, nCUBE, and the newer VAXclusters typify this design.

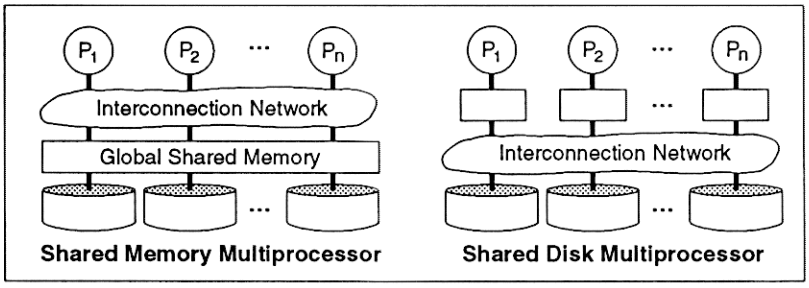


Figure 5.
The shared-memory and shared-disk designs. A shared-memory multiprocessor connects all processors to a globally shared memory. Multiprocessor IBM/370, VAX, and Sequent computers are typical examples of shared-memory designs. Shared-disk systems give each processor a private memory, but all the processors can directly address all the disks. Digital's VAXcluster and IBM's Sysplex typify this design.

Figure 6.
Example of a scan of a telephone relation to find the phone numbers of all people named Smith.

```
SELECT  telephone_number       /* the output attribute(s) */
FROM    telephone_book         /* the input relation      */
WHERE   last_name = 'Smith';   /* the predicate           */
```

often use an affinity scheduling mechanism to reduce this interference; giving each process an affinity to a particular processor. This is a form of data partitioning; it represents an evolutionary step toward the shared-nothing design. Partitioning a shared-memory system creates many of the skew and load balancing problems faced by a shared-nothing machine; but reaps none of the simpler hardware interconnect benefits. Based on this experience, we believe high-performance shared-memory machines will not economically scale beyond a few processors when running database applications.

To ameliorate the interference problem, most shared-memory multiprocessors have adopted a shared-disk architecture. This is the logical consequence of affinity scheduling. If the disk interconnection network can scale to thousands of disks and processors, then a shared-disk design is adequate for large read-only databases and for databases where there is no concurrent sharing. The shared-disk architecture is not very effective for database applications that read and write a shared database. A processor wanting to update some data must first obtain the current copy of that data. Since others might be updating the same data concurrently, the processor must declare its intention to update the data. Once this declaration has been honored and acknowledged by all the other processors, the updator can read the shared data from disk and update it. The processor must then write the shared data to disk so that subsequent readers and writers will be aware of the update. There are many optimizations of this protocol, but they all end up exchanging reservation messages and exchanging large physical data pages. This creates processor interference and delays. It creates heavy traffic on the shared interconnection network.

For shared database applications, the shared-disk approach is much more expensive than the shared-nothing approach of exchanging small high-level logical questions and answers among clients and servers. One solution to this interference has been to give data a processor affinity; other processors wanting to access the data send messages to the server managing the data. This has emerged as a major application of transaction processing monitors that partition the load among partitioned servers, and is also a major application for remote procedure calls. Again, this trend toward the partitioned data model and shared-nothing architecture on a shared-disk system reduces interference. Since the shared-disk system interconnection network is difficult to scale to thousands of processors and disks, many conclude that it would be better to adopt the shared-nothing architecture from the start.

Given the shortcomings of shared-disk and shared-memory architectures, why have computer architects been slow to adopt the shared-nothing approach? The first answer is simple, high-performance, low-cost commodity components have only recently become available. Traditionally, commodity components provided relatively low performance and low quality.

Today, old software is the most significant barrier to the use of parallelism. Old software written for uniprocessors gets no speedup or scaleup when put on any kind of multiprocessor. It must be rewritten to benefit from parallel processing and multiple disks. Database applications are a unique exception to this. Today, most database programs are written in the relational language SQL that has been standardized by both ANSI and ISO. It is possible to take standard SQL applications written for uniprocessor systems and execute them in parallel on shared-nothing database

machines. Database systems can automatically distribute data among multiple processors. Teradata and Tandem routinely port SQL applications to their system and demonstrate near-linear speedups and scaleups. The following subsection explains the basic techniques used by such parallel database systems.

A Parallel Dataflow Approach to SQL Software

Terabyte on-line databases, consisting of billions of records, are becoming common as the price of on-line storage decreases. These databases are often represented and manipulated using the SQL relational model. The next few paragraphs give a rudimentary introduction to relational model concepts needed to understand the remainder of this article.

A relational database consists of *relations* (*files* in COBOL terminology) that in turn contain *tuples* (*records* in COBOL terminology). All the tuples in a relation have the same set of *attributes* (*fields* in COBOL terminology).

Relations are created, updated, and queried by writing SQL statements. These statements are syntactic sugar for a simple set of operators chosen from the relational algebra. *Select-project,* here called *scan,* is the simplest and most common operator—it produces a row-and-column subset of a relational table. A scan of relation R using predicate P and attribute list L produces a relational data stream as output. The scan reads each tuple, t, of R and applies the predicate P to it. If $P(t)$ is true, the scan discards any attributes of t not in L and inserts the resulting tuple in the scan output stream. Expressed in SQL, a scan of a telephone book relation to find the phone numbers of all people named Smith would be written as shown in Figure 6. A scan's output stream can be sent to another relational operator, returned to an application, displayed on a terminal, or printed in a report. Therein lies the beauty and utility of the relational model. The uniformity of the data and operators allow them to be arbitrarily composed into dataflow graphs.

The output of a scan may be sent to a *sort* operator that will reorder the tuples based on an attribute sort criteria, optionally eliminating duplicates. SQL defines several *aggregate* operators to summarize attributes into a single value, for example, taking the sum, min, or max of an attribute, or counting the number of distinct values of the attribute. The *insert* operator adds tuples from a stream to an existing relation. The *update* and *delete* operators alter and delete tuples in a relation matching a scan stream.

The relational model defines several operators to combine and compare two or more relations. It provides the usual set operators *union, intersection, difference,* and some more exotic ones like *join* and *division.* Discussion here will focus on the *equi-join* operator (here called *join*). The join operator composes two relations, A and B, on some attribute to produce a third relation. For each tuple, ta, in A, the join finds all tuples, tb, in B whose attribute values are equal to that of ta. For each matching pair of tuples, the join operator inserts into the output stream a tuple built by concatenating the pair.

Codd, in a classic paper, showed that the relational data model can represent any form of data, and that these operators are complete [5]. Today, SQL applications are typically a combination of conventional programs and SQL statements. The programs interact with clients, perform data display, and provide high-level direction of the SQL dataflow.

The SQL data model was originally proposed to improve programmer productivity by offering a nonprocedural database language. Data independence was an additional benefit; since the programs do not specify how the query is to be executed, SQL programs continue to operate as the logical and physical database schema evolves.

Parallelism is an unanticipated benefit of the relational model. Since relational queries are really just relational operators applied to very large collections of data, they offer many opportunities for parallelism. Since the queries are presented in a nonprocedural language, they offer considerable latitude in executing the queries.

Relational queries can be executed as a dataflow graph. As mentioned in the first section of this article, these graphs can use both pipelined parallelism and partitioned parallelism. If one operator sends its output to another, the two operators can execute in parallel giving potential speedup of two.

The benefits of pipeline parallelism are limited because of three factors: (1) Relational pipelines are rarely very long—a chain of length ten is unusual. (2) Some relational operators do not emit their first output until they have consumed all their inputs. Aggregate and sort operators have this property. One cannot pipeline these operators. (3) Often, the execution cost of one operator is much greater than the others (this is an example of skew). In such cases, the speedup obtained by pipelining will be very limited.

Partitioned execution offers much better opportunities for speedup and scaleup. By taking the large relational operators and partitioning their inputs and outputs, it is possible to use divide-and-conquer to turn one big job into many independent little ones. This is an ideal situation for speedup and scaleup. Partitioned data is the key to partitioned execution.

Data Partitioning. Partitioning a relation involves distributing its tuples over several disks. Data partitioning has its origins in centralized systems that had to partition files, either because the file was too big for one disk, or because the file access rate could not be supported by a single disk. Distributed databases use data partitioning when they place relation fragments at different network sites [23]. Data par-

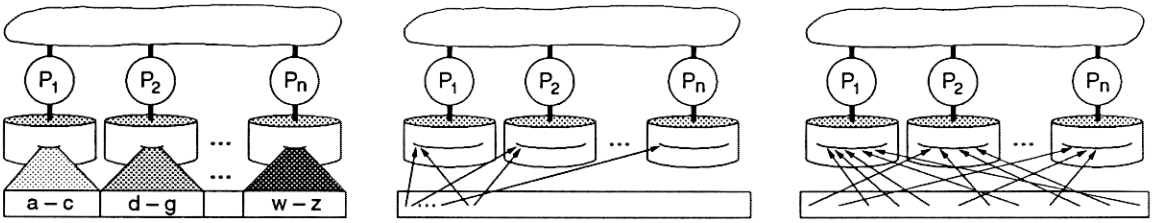


Figure 7.
The three basic partitioning schemes. Range partitioning maps contiguous attribute ranges of a relation to various disks. Round-robin partitioning maps the i'th tuple to disk i **mod** n. Hashed partitioning maps each tuple to a disk location based on a hash function. Each of these schemes spreads data among a collection of disks, allowing parallel disk access and parallel processing.

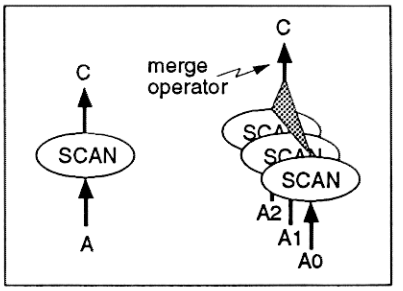


Figure 8.
Partitioned data parallelism. A simple relational dataflow graph showing a relational scan (project and select) decomposed into three scans on three partitions of the input stream or relation. These three scans send their output to a merge node that produces a single data stream.

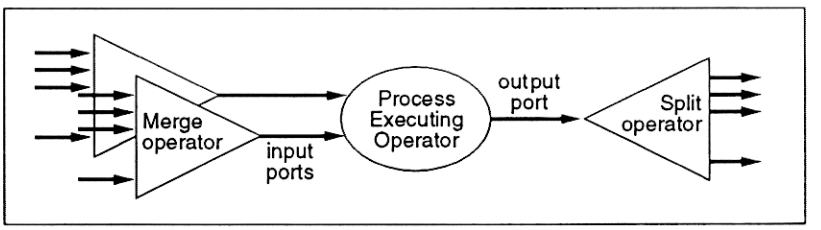


Figure 9.
Merging the inputs and partitioning the output of an operator. A relational dataflow graph showing a relational operator's inputs being merged to a sequential stream per port. The operator's output is being decomposed by a split operator in several independent streams. Each stream may be a duplicate or a partitioning of the operator output stream into many disjoint streams. With the split and merge operators, a web of simple sequential dataflow nodes can be connected to form a parallel execution plan.

titioning allows parallel database systems to exploit the I/O bandwidth of multiple disks by reading and writing them in parallel. This approach provides I/O bandwidth superior to RAID-style systems without needing any specialized hardware [22, 24].

The simplest partitioning strategy distributes tuples among the fragments in a *round-robin* fashion. This is the partitioned version of the classic entry-sequence file. Round-robin partitioning is excellent if all applications want to access the relation by sequentially scanning all of it on each query. The problem with round-robin partitioning is that applications frequently want to associatively access tuples, meaning that the application wants to find all the tuples having a particular attribute value. The SQL query looking for the Smiths in the phone book shown in Figure 6 is an example of an associative search.

Hash partitioning is ideally suited for applications that want only sequential and associative access to the data. Tuples are placed by applying a *hashing* function to an attribute of each tuple. The function specifies the placement of the tuple on a particular disk. Associative access to the tuples with a specific attribute value can be directed to a single disk, avoiding the overhead of starting queries on multiple disks. Hash partitioning mechanisms are provided by Arbre, Bubba, Gamma, and Teradata.

Database systems pay considerable attention to clustering related data together in physical storage. If a set of tuples is routinely accessed together, the database system attempts to store them on the same physical page. For example, if the Smiths of the phone book are routinely accessed in alphabetical order, then they should be stored on pages in that order, these pages should be clustered together on disk to allow sequential prefetching and other optimizations. Clustering is very application-specific. For example, tuples describing nearby streets should be clustered together in geographic databases, tuples describing the line items of an invoice should be clustered with the invoice tuple in an inventory control application.

Hashing tends to randomize data rather than cluster it. *Range partitioning* clusters tuples with similar attributes together in the same partition. It is good for sequential and associative access, and is also good for clustering data. Figure 7 shows range partitioning based on lexicographic order, but any clustering algorithm is possible. Range partitioning derives its name from the typical SQL range queries such as

```
latitude BETWEEN 38° AND 39°
```

Arbre, Bubba, Gamma, Oracle, and Tandem provide range partitioning.

The problem with range partitioning is that it risks *data skew*, where all the data is placed in one partition, and *execution skew* in which all the execution occurs in one partition. Hashing and round-robin are less susceptible to these skew problems. Range partitioning can minimize skew by picking non-uniformly-distributed partitioning criteria. Bubba uses this concept by considering the access frequency *(heat)* of each tuple when creating partitions of a relation; the goal being to balance the frequency with which each partition is accessed (its *temperature*) rather than the actual number of tuples on each disk (its volume) [6].

While partitioning is a simple concept that is easy to implement, it raises several new physical database design issues. Each relation must now have a partitioning strategy and a set of disk fragments. Increasing the degree of partitioning usually reduces the response time for an individual query and increases the overall throughput of the system. For sequential scans, the response time decreases because more processors and disks are used to execute the query. For associative scans, the response time improves because fewer tuples are stored at each node and hence the size of the index that must be searched decreases.

There is a point beyond which further partitioning actually increases the response time of a query. This point occurs when the cost of starting a query on a node becomes a significant fraction of the actual execution time [6, 11].

Table 1.
Sample Split Operators.
Each split operator maps tuples to a set of output streams (ports of other processes) depending on the range value (predicate) of the input tuple. The split operator on the left is for the relation A scan in Figure 10, while the table on the right is for the relation B scan. The tables partition the tuples among three data streams.

Relation A Scan Split Operator		Relation B Scan Split Operator	
Predicate	**Destination Process**	**Predicate**	**Destination Process**
"A-H"	(CPU #5, Process #3, Port #0)	"A-H"	(CPU #5, Process #3, Port #1)
"I-Q"	(CPU #7, Process #8, Port #0)	"I-Q"	(CPU #7, Process #8, Port #1)
"R-Z"	(CPU #2, Process #2, Port #0)	"R-Z"	(CPU #2, Process #2, Port #1)

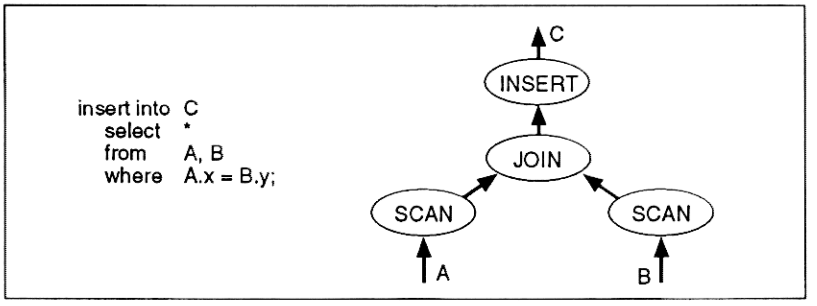


Figure 10.
A simple SQL query and the associated relational query graph. The query specifies that a join is to be performed between relations A and B by comparing the x attribute of each tuple from the A relation with the y attribute value of each tuple of the B relation. For each pair of tuples that satisfy the predicate, a result tuple is formed from all the attributes of both tuples. This result tuple is then added to the result relation C. The associated logical query graph (as might be produced by a query optimizer) shows a tree of operators, one for the join, one for the insert, and one for scanning each input relation.

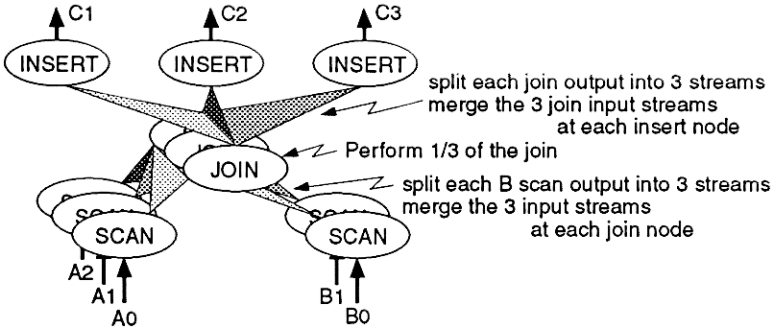


Figure 11.
A simple relational dataflow graph. It shows two relational scans (project and select) consuming two input relations, A and B and feeding their outputs to a join operator that in turn produces a data stream C.

Parallelism Within Relational Operators. Data partitioning is the first step in partitioned execution of relational dataflow graphs. The basic idea is to use parallel data streams instead of writing new parallel operators (programs). This approach enables the use of unmodified, existing sequential routines to execute the relational operators in parallel. Each relational operator has a set of *input ports* on which input tuples arrive and an *output* port to which the operator's output stream is sent. The parallel dataflow works by partitioning and merging data streams into these sequential ports. This approach allows the use of existing sequential relational operators to execute in parallel.

Consider a scan of a relation, A, that has been partitioned across three disks into fragments A0, A1, and A2. This scan can be imple-

mented as three scan operators that send their output to a common merge operator. The merge operator produces a single output data stream to the application or to the next relational operator. The parallel query executor creates the three scan processes shown in Figure 8 and directs them to take their inputs from three different sequential input streams (A0, A1, A2). It also directs them to send their outputs to a common merge node. Each scan can run on an independent processor and disk. So the first basic parallelizing operator is a *merge* that can combine several parallel data streams into a single sequential stream.

The merge operator tends to focus data on one spot. If a multistage parallel operation is to be done in parallel, a single data stream must be split into several independent streams. A *split operator* is used to partition or replicate the stream of tuples produced by a relational operator. A split operator defines a mapping from one or more attribute values of the output tuples to a set of destination processes (see Figure 9).

As an example, consider the two split operators shown in Table 1 in conjunction with the SQL query shown in Figure 10. Assume that three processes are used to execute the join operator, and that five other processes execute the two scan operators—three scanning partitions of relation A while two scan partitions of relation B. Each of the three relation A scan nodes will have the same split operator, sending all tuples between "A-H" to port 1 of join process 0, all between "I-Q" to port 1 of join process 1, and all between "R-Z" to port 1 of join process 2. Similarly the two relation B scan nodes have the same split operator except that their outputs are merged by port 1 (not port 0) of each join process. Each join process sees a sequential input stream of A tuples from the port 0 merge (the left-scan nodes) and another sequential stream of B tuples from the port 1 merge (the right-scan nodes). The outputs of each join are, in turn, split into three streams based on the partitioning criterion of relation C.

To clarify this example, consider the first join process in Figure 11 (processor 5, process 3, ports 0 and 1 in Table 1). It will receive all the relation A "A-H" tuples from the three relation A scan operators merged as a single stream on port 0, and will get all the "A-H" tuples from relation B merged as a single stream on port 1. It will join them using a hash-join, sort-merge join, or even a nested join if the tuples arrive in the proper order.

If each of these processes is on an independent processor with an independent disk, there will be little interference among them. Such dataflow designs are a natural application for shared-nothing machine architectures.

The split operator in Table 1 is just an example. Other split operators might duplicate the input stream, or partition it round-robin, or partition it by hash. The partitioning function can be an arbitrary program. Gamma, Volcano, and Tandem use this approach [14]. It has several advantages including the automatic parallelism of any new operator added to the system, plus support for many kinds of parallelism.

The split and merge operators have flow control and buffering built into them. This prevents one operator from getting too far ahead in the computation. When a split-operator's output buffers fill, it stalls the relational operator until the data target requests more output.

For simplicity, these examples have been stated in terms of an operator per process. But it is entirely possible to place several operators within a process to get coarser grained parallelism. The fundamental idea though is to build a self-pacing dataflow graph and distribute it in a shared-nothing machine in a way that minimizes interference.

Specialized Parallel Relational Operators. Some algorithms for relational operators are especially appropriate for parallel execution, either because they minimize data flow, or because they better tolerate data and execution skew. Improved algorithms have been found for most of the relational operators. The evolution of join operator algorithms is sketched here as an example of these improved algorithms.

Recall that the join operator combines two relations, A and B, to produce a third relation containing all tuple pairs from A and B with matching attribute values. The conventional way of computing the join is to sort both A and B into new relations ordered by the join attribute. These two intermediate relations are then compared in sorted order, and matching tuples are inserted in the output stream. This algorithm is called *sort-merge* join.

Many optimizations of sort-merge join are possible, but since sort has execution cost *nlog(n)*, sort-merge join has an *nlog(n)* execution cost. Sort-merge join works well in a parallel dataflow environment unless there is data skew. In case of data skew, some sort partitions may be much larger than others. This in turn creates execution skew and limits speedup and scaleup. These skew problems do not appear in centralized sort-merge joins.

Hash-join is an alternative to sort-merge join. It has linear execution cost rather than *nlog(n)* execution cost, and it is more resistant to data skew. It is superior to sort-merge join unless the input streams are already in sorted order. Hash join works as follows. Each of the relations A and B are first hash partitioned on the join attribute. A hash partition of relation A is hashed into memory. The corresponding partition of table relation B is scanned, and each tuple is compared against the main-memory hash table for the A partition. If there is a match, the pair of tuples are sent to the output stream. Each pair of hash partitions is compared

in this way.

The hash join algorithm breaks a big join into many little joins. If the hash function is good and if the data skew is not too bad, then there will be little variance in the hash bucket size. In these cases hash-join is a linear-time join algorithm with linear speedup and scaleup. Many optimizations of the parallel hash-join algorithm have been discovered over the last decade. In pathological skew cases, when many or all tuples have the same attribute value, one bucket may contain all the tuples. In these cases no algorithm is known to speedup or scaleup.

The hash-join example shows that new parallel algorithms can improve the performance of relational operators. This is a fruitful research area [4, 8, 18, 20, 25, 26, 38, 39]. Although parallelism can be obtained from conventional sequential relational algorithms by using split and merge operators, we expect that many new algorithms will be discovered in the future.

The State of the Art

Teradata

Teradata quietly pioneered many of the ideas presented in this article. Since 1978 they have been building shared-nothing highly-parallel SQL systems based on commodity microprocessors, disks, and memories. Teradata systems act as SQL servers to client programs operating on conventional computers.

Teradata systems may have over 1,000 processors and many thousands of disks. The Teradata processors are functionally divided into two groups: Interface Processors (IFPs) and Access Module Processors (AMPs). The IFPs handle communication with the host, query parsing and optimization, and coordination of AMPs during query execution. The AMPs are responsible for executing queries. Each AMP typically has several disks and a large memory cache. IFPs and AMPs are interconnected by a dual redundant, tree-shaped interconnect called the Y-net [33].

Each relation is hash partitioned over a subset of the AMPs. When a tuple is inserted into a relation, a hash function is applied to the primary key of the tuple to select an AMP for storage. Once a tuple arrives at an AMP, a second hash function determines the tuple's placement in its fragment of the relation. The tuples in each fragment are in hash-key order. Given a value for the key attribute, it is possible to locate the tuple in a single AMP. The AMP examines its cache, and if the tuple is not present, fetches it in a single disk read. Hash secondary indices are also supported.

Hashing is used to split the outputs of relational operators into intermediate relations. Join operators are executed using a parallel sort-merge algorithm. Rather than using pipelined parallel execution, during the execution of a query, each operator is run to completion on all participating nodes before the next operator is initiated.

Teradata has installed many systems containing over 100 processors and hundreds of disks. These systems demonstrate near-linear speedup and scaleup on relational queries, and far exceed the speed of traditional mainframes in their ability to process large (terabyte) databases.

Tandem NonStop SQL

The Tandem NonStop SQL system is composed of processor clusters interconnected via 4-plexed fiber-optic rings. Unlike most other systems discussed in this article, the Tandem systems run the applications on the same processors and operating system as the database servers. There is no front-end/back-end distinction between programs and machines. The systems are configured at a disk per MIPS, so each 10-MIPS processor has about 10 disks. Disks are typically duplexed [2]. Each disk is served by a set of processes managing a large shared RAM cache, a set of locks, and log records for the data on that disk pair. Considerable effort is spent on optimizing sequential scans by prefetching large units, and by filtering and manipulating the tuples with SQL predicates at these disk servers. This minimizes traffic on the shared interconnection network.

Relations may be range partitioned across multiple disks. Entry-sequenced, relative, and B-tree organizations are supported. Only B-tree secondary indices are supported. Nested join, sort-merge join, and hash join algorithms are provided. Parallelization of operators in a query plan is achieved by inserting split and merge operators between operator nodes in the query tree. Scans, aggregates, joins, updates, and deletes are executed in parallel. In addition, several utilities use parallelism (e.g., load, reorganize, . . .) [31, 39].

Tandem systems are primary designed for on-line transaction processing (OLTP)—running many simple transactions against a large shared database. Beyond the parallelism inherent in running many independent transactions in parallel, the main parallelism feature for OLTP is parallel index update. SQL relations typically have five indices on them, although it is not uncommon to see 10 indices on a relation. These indices speed reads, but slow down inserts, updates, and deletes. By doing the index maintenance in parallel, the maintenance time for multiple indices can be held almost constant if the indices are spread among many processors and disks.

Overall, the Tandem systems demonstrate near-linear scaleup on transaction processing workloads, and near-linear speedup and scaleup on large relational queries [10, 31].

Gamma

The current version of Gamma runs on a 32-node Intel iPSC/2 Hypercube with a disk attached to each node. In addition to round-robin, range and hash partitioning, Gamma also provides hybrid-range

partitioning that combines the best features of the hash and range partitioning strategies [12]. Once a relation has been partitioned, Gamma provides both clustered and nonclustered indices on either the partitioning or nonpartitioning attributes. The indices are implemented as B-trees or hash tables.

Gamma uses split and merge operators to execute relational algebra operators using both parallelism and pipelining [9]. Sort-merge and three different hash join methods are supported [7]. Near-linear speedup and scaleup for relational queries has been measured on this architecture [9, 25, 26].

The Super Database Computer

The Super Database Computer (SDC) project at the University of Tokyo presents an interesting contrast to other database systems [16, 20]. SDC takes a combined hardware and software approach to the performance problem. The basic unit, called a processing module (PM), consists of one or more processors on a shared memory. These processors are augmented by a special-purpose sorting engine that sorts at high speed (3MB/second at present), and by a disk subsystem [19]. Clusters of processing modules are connected via an omega network that provides both nonblocking NxN interconnect and some dynamic routing minimize skewed data distribution during hash joins. The SDC is designed to scale to thousands of PMs, and so considerable attention is paid to the problem of data skew.

Data is partitioned among the PMs by hashing. The SDC software includes a unique operating system, and a relational database query executor. The SDC is a shared-nothing design with a software dataflow architecture. This is consistent with our assertion that current parallel database machines systems use conventional hardware. But the special-purpose design of the omega network and of the hardware sorter clearly contradict the thesis that special-purpose hardware is not a good investment of development resources. Time will tell whether these special-purpose components offer better price performance or peak performance than shared-nothing designs built of conventional hardware.

Bubba

The Bubba prototype was implemented using a 40-node FLEX/32 multiprocessor with 40 disks [4]. Although this is a shared-memory multiprocessor, Bubba was designed as a shared-nothing system and the shared-memory is only used for message passing. Nodes are divided into three groups: Interface Processors for communicating with external host processors and coordinating query execution; Intelligent Repositories for data storage and query execution; and Checkpoint/Logging Repositories. While Bubba also uses partitioning as a storage mechanism (both range and hash partitioning mechanisms are provided) and dataflow processing mechanisms, Bubba is unique in several ways. First, Bubba uses FAD rather than SQL as its interface language. FAD is an extended-relational persistent programming language. FAD provides support for complex objects via several type constructors including shared subobjects, set-oriented data manipulation primitives, and more traditional language constructs. The FAD compiler is responsible for detecting operations that can be executed in parallel according to how the data objects being accessed are partitioned. Program execution is performed using a dataflow execution paradigm. The task of compiling and parallelizing a FAD program is significantly more difficult than parallelizing a relational query. Another Bubba feature is its use of a single-level store mechanism in which the persistent database at each node is mapped to the virtual memory address space of each process executing at the node. This is in contrast to the traditional approach of files and pages. Similar mechanisms are used in IBM's AS400 mapping of SQL databases into virtual memory, HP's mapping of the Image Database into the operating system virtual address space, and Mach's mapped file [34] mechanism. This approach simplified the implementation of the upper levels of the Bubba software.

Other Systems

Other parallel database system prototypes include XPRS [30], Volcano [14], Arbre [21], and the PERSIST project under development at IBM Research Labs in Hawthorne and Almaden. While both Volcano and XPRS are implemented on shared-memory multiprocessors, XPRS is unique in its exploitation of the availability of massive shared-memory in its design. In addition, XPRS is based on several innovative techniques for obtaining extremely high performance and availability.

Recently, the Oracle database system has been implemented atop a 64-node nCUBE shared-nothing system. The resulting system is the first to demonstrate more than 1,000 transactions per second on the industry-standard TPC-B benchmark. This is far in excess of Oracle's performance on conventional mainframe systems—both in peak performance and in price/performance [13].

The NCR Corporation has announced the 3600 and 3700 product lines that employ shared-nothing architectures running System V R4 of Unix on Intel 486 and 586 processors. The interconnection network for the 3600 product line uses an enhanced Y-Net licensed from Teradata while the 3700 is based on a new multistage interconnection network being developed jointly by NCR and Teradata. Two software offerings have been announced. The first, a port of the Teradata software to a Unix environment, is targeted toward the decision-support marketplace. The second, based on a parallelization of the Sybase DBMS, is intended primarily for transaction processing workloads.

Database Machines and Grosch's Law

Today shared-nothing database machines have the best peak performance and best price performance available. When compared to traditional mainframes, the Tandem system scales linearly well beyond the largest reported mainframes on the TPC-A transaction processing benchmark. Its price/performance on these benchmarks is three times cheaper than the comparable mainframe numbers. Oracle on an nCUBE has the highest reported TPC-B numbers, and has very competitive price performance [13, 36]. These benchmarks demonstrate linear scaleup on transaction processing benchmarks.

Gamma, Tandem, and Teradata have demonstrated linear speedup and scaleup on complex relational database benchmarks. They scale well beyond the size of the largest mainframes. Their performance and price performance is generally superior to mainframe systems.

These observations defy Grosch's law. In the 1960s, Herb Grosch observed that there is an economy-of-scale in computing. At that time, expensive computers were much more powerful than inexpensive computers. This gave rise to super-linear speedups and scaleups. The current pricing of mainframes at $25,000/MIPS and $1,000/MB of RAM reflects this view. Meanwhile, microprocessors are selling for $250/MIPS and $100/MB of RAM.

By combining hundreds or thousands of these small systems, one can build an incredibly powerful database machine for much less money than the cost of a modest mainframe. For database problems, the near-linear speedup and scaleup of these shared-nothing machines allows them to outperform current shared-memory and shared disk mainframes.

Grosch's law no longer applies to database and transaction processing problems. There is no economy of scale. At best, one can expect linear speedup and scaleup of performance and price/performance. Fortunately, shared-nothing database architectures achieve this near-linear performance.

Future Directions and Research Problems

Mixing Batch and OLTP Queries

The second section of this article, "Basic Techniques for Parallel Database Machine Implementation", concentrated on the basic techniques used for processing complex relational queries in a parallel database system. Concurrently running a mix of both simple and complex queries concurrently presents several unsolved problems.

One problem is that large relational queries tend to acquire many locks and tend to hold them for a relatively long time. This prevents concurrent updates of the data by simple on-line transactions. Two solutions are currently offered: give the ad-hoc queries a fuzzy picture of the database, not locking any data as they browse it. Such a "dirty-read" solution is not acceptable for some applications. Several systems offer a versioning mechanism that gives readers a consistent (old) version of the database while updators are allowed to create newer versions of objects. Other, perhaps better, solutions for this problem may also exist.

Priority scheduling is another mixed-workload problem. Batch jobs have a tendency to monopolize the processor, flood the memory cache, and make large demands on the I/O subsystem. It is up to the underlying operating system to quantize and limit the resources used by such batch jobs to ensure short response times and low variance in response times for short transactions. A particularly difficult problem is the *priority inversion problem,* in which a low-priority client makes a request to a high-priority server. The server must run at high priority because it is managing critical resources. Given this, the work of the low-priority client is effectively promoted to high priority when the low-priority request is serviced by the high-priority server. There have been several ad-hoc attempts at solving this problem, but considerably more work is needed.

Parallel Query Optimization

Current database query optimizers do not consider all possible plans when optimizing a relational query. While cost models for relational queries running on a single processor are now well-understood [27] they still depend on cost estimators that are a guess at best. Some dynamically select from among several plans at run time depending on, for example, the amount of physical memory actually available and the cardinalities of the intermediate results [15]. To date, no query optimizers consider all the parallel algorithms for each operator and all the query tree organizations. More work is needed in this area.

Another optimization problem relates to highly skewed value distributions. Data skew can lead to high variance in the size of intermediate relations, leading to both poor query plan cost estimates and sub-linear speedup. Solutions to this problem are an area of active research [17, 20, 37, 38].

Application Program Parallelism

The parallel database systems offer parallelism within the database system. Missing are tools to structure application programs to take advantage of parallelism inherent in these parallel systems. While automatic parallelization of applications programs written in COBOL may not be feasible, library packages to facilitate explicitly parallel application programs are needed. Ideally the SPLIT and MERGE operators could be packaged so that applications could benefit from them.

Physical Database Design

For a given database and workload there are many possible indexing and partitioning combinations. Database design tools are needed to help the database administrator select among these many design options. Such tools might accept as input a description of the queries comprising the workload, their frequency of execution, statistical information about the relations in the database, and a description of the

processors and disks. The resulting output would suggest a partitioning strategy for each relation plus the indices to be created on each relation. Steps in this direction are beginning to appear.

Current algorithms partition relations using the values of a single attribute. For example, geographic records could be partitioned by longitude or latitude. Partitioning on longitude allows selections for a longitude range to be localized to a limited number of nodes, selections on latitude must be sent to all the nodes. While this is acceptable in a small configuration, it is not acceptable in a system with thousands of processors. Additional research is needed on multidimensional partitioning and search algorithms.

On-line Data Reorganization and Utilities

Loading, reorganizing, or dumping a terabyte database at a megabyte per second takes over 12 days and nights. Clearly parallelism is needed if utilities are to complete within a few hours or days. Even then, it will be essential that the data be available while the utilities are operating. In the SQL world, typical utilities create indices, add or drop attributes, add constraints, and physically reorganize the data, changing its clustering.

One unexplored and difficult problem is how to process database utility commands while the system remains operational and the data remains available for concurrent reads and writes by others. The fundamental properties of such algorithms are that they must be *on-line* (operate without making data unavailable), *incremental* (operate on parts of a large database), *parallel* (exploit parallel processors), and *recoverable* (allow the operation to be canceled and return to the old state).

Summary and Conclusions

Like most applications, database systems want cheap, fast hardware. Today that means commodity processors, memories, and disks. Consequently, the hardware concept of a *database machine* built of exotic hardware is inappropriate for current technology. On the other hand, the availability of fast microprocessors, and small inexpensive disks packaged as standard inexpensive but fast computers is an ideal platform for *parallel database systems*. A shared-nothing architecture is relatively straightforward to implement and, more importantly, has demonstrated both speedup and scaleup to hundreds of processors. Furthermore, shared-nothing architectures actually simplify the software implementation. If the software techniques of data partitioning, dataflow, and intra-operator parallelism are employed, the task of converting an existing database management system to a highly parallel one becomes relatively straightforward. Finally, there are certain applications (e.g., data mining in terabyte databases) that require the computational and I/O resources available only from a parallel architecture.

While the successes of both commercial products and prototypes demonstrate the viability of highly parallel database machines, several research issues remain unsolved including techniques for mixing ad-hoc queries with on-line transaction processing without seriously limiting transaction throughput, improved optimizers for parallel queries, tools for physical database design, on-line database reorganization, and algorithms for handling relations with highly skewed data distributions. Some application domains are not well supported by the relational data model. It appears that a new class of database systems based on an object-oriented data model is needed. Such systems pose a host of interesting research problems that require further examination. ■

References

1. Alexander, W., et al. Process and dataflow control in distributed data-intensive systems. In *Proceedings of ACM SIGMOD Conference* (Chicago, Ill., June 1988) ACM, NY, 1988.
2. Bitton, D. and Gray, J. Disk shadowing. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases* (Los Angeles, Calif., August, 1988).
3. Boral, H. and DeWitt, D. Database machines: An idea whose time has passed? A critique of the future of database machines. In *Proceedings of the 1983 Workshop on Database Machines*. H.-O. Leilich and M. Missikoff, Eds., Springer-Verlag, 1983.
4. Boral, H. et al. Prototyping Bubba: A highly parallel database system. *IEEE Knowl. Data Eng. 2,* 1, (Mar. 1990).
5. Codd, E.F. A relational model of data for large shared databanks. *Commun. ACM 13,* 6 (June 1970).
6. Copeland, G., Alexander, W., Boughter, E., and Keller, T. Data placement in Bubba. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Chicago, May 1988).
7. DeWitt, D.J., Katz, R., Olken, F., Shapiro, D., Stonebraker, M. and Wood, D. Implementation techniques for main memory database systems. In *Proceedings of the 1984 SIGMOD Conference,* (Boston, Mass., June, 1984).
8. DeWitt, D., et al. GAMMA—A high performance dataflow database machine. In *Proceedings of the 1986 VLDB Conference* (Japan, August 1986).
9. DeWitt, D., et al. The Gamma database machine project. *IEEE Knowl. Data Eng. 2,* 1 (Mar. 1990).
10. Engelbert, S, Gray, J., Kocher, T., and Stah, P. A benchmark of nonstop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases. Tandem Computers, Technical Report 89.4, Tandem Part No. 27469, May 1989.
11. Ghandeharizadeh, S., and DeWitt, D.J. Performance analysis of alternative declustering strategies. In *Proceedings of the Sixth International Conference on Data Engineering* (Feb. 1990).
12. Ghandeharizadeh, S., and Dewitt, D.J. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of the Sixth International Conference on Very Large Data Bases,* (Melbourne, Australia, Aug. 1990).
13. Gibbs, J. Massively parallel systems, rethinking computing for business and science. *Oracle 6,* 1 (Dec. 1991).
14. Graefe, G. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of 1990*

ACM-SIGMOD International Conference on Management of Data (May 1990).

15. Graefe, G., and Ward, K. Dynamic query evaluation plans. In *Proceedings of the 1989 SIGMOD Conference*, (Portland, Ore., June 1989).
16. Hirano, M.S. et al. Architecture of SDC, the super database computer. In *Proceedings of JSPP '90*. 1990.
17. Hua, K.A. and Lee, C. Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*. (Barcelona, Spain, Sept. 1991).
18. Kitsuregawa, M., Tanaka, H., and Moto-oka, T. Application of hash to data base machine and its architecture. *New Generation Computing 1*, 1 (1983).
19. Kitsuregawa, M., Yang, W., and Fushimi, S. Evaluation of 18-stage pipeline hardware sorter. In *Proceedings of the Third International Conference on Data Engineering* (Feb. 1987).
20. Kitsuregawa, M., and Ogawa, Y. A new parallel hash join method with robustness for data skew in super database computer (SDC). In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*. (Melbourne, Australia, Aug. 1990).
21. Lorie, R., Daudenarde, J., Hallmark, G., Stamos, J., and Young, H. Adding intra-transaction parallelism to an existing DBMS: Early experience. *IEEE Data Engineering Newsletter 12*, 1 (Mar. 1989).
22. Patterson, D. A., Gibson, G. and Katz, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*. (Chicago, May 1988).
23. Ries, D. and Epstein, R. Evaluation of distribution criteria for distributed database systems. UBC/ERL Technical Report M78/22, UC Berkeley, May, 1978.
24. Salem, K. and Garcia-Molina, H. Disk-striping. Department of Computer Science, Princeton University Technical Report EEDS-TR-322-84, Princeton, N.J., Dec. 1984.
25. Schneider, D. and DeWitt, D. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 SIGMOD Conference* (Portland, Ore., June 1989).
26. Schneider, D. and DeWitt, D. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*. (Melbourne, Australia, Aug., 1990).
27. Selinger P. G., et al. Access path selection in a relational database management system. In *Proceedings of the 1979 SIGMOD Conference* (Boston, Mass., May 1979).
28. Stonebraker, M. Muffin: A distributed database machine. ERL Technical Report UCB/ERL M79/28, University of California at Berkeley, May 1979.
29. Stonebraker, M. The case for shared nothing. *Database Eng. 9*, 1 (1986).
30. Stonebraker, M., Katz, R., Patterson, D., and Ousterhout, J. The design of XPRS. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*. (Los Angeles, Calif., Aug. 1988).
31. Tandem Database Group. NonStop SQL, a distributed, high-performance, high-reliability implementation of SQL. Workshop on High Performance Transaction Systems, Asilomar, CA, Sept. 1987.
32. Tandem Performance Group. A benchmark of non-stop SQL on the debit credit transaction. In *Proceedings of the 1988 SIGMOD Conference* (Chicago, Ill., June 1988).
33. Teradata Corporation. DBC/1012 Data Base Computer Concepts & Facilities. Document No. C02-0001-00, 1983.
34. Tevanian, A., et al. A Unix interface for shared memory and memory mapped files under Mach. Dept. of Computer Science Technical Report, Carnegie Mellon University, July, 1987.
35. Thakkar, S.S. and Sweiger, M. Performance of an OLTP application on symmetry multiprocessor system. In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*. (Seattle, Wash., May, 1990).
36. *The Performance Handbook for Database and Transaction Processing Systems*. J. Gray, Ed., Morgan Kaufmann, San Mateo, Ca., 1991.
37. Walton, C.B., Dale, A.G., and Jenevein, R.M. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*. (Barcelona, Spain, Sept. 1991).
38. Wolf, J.L., Dias, D.M., and Yu, P.S. An effective algorithm for parallelizing sort-merge joins in the presence of data skew. In *Proceedings of the Second International Symposium on Parallel and Distributed Systems*. (Dublin, Ireland, July, 1990).
39. Zeller, H.J. and Gray, J. Adaptive hash joins for a multiprogramming environment. In *Proceedings of the 1990 VLDB Conference* (Australia, Aug. 1990).

CR Categories and Subject Descriptors: B.5.1 [**Register-Transfer-Level Implementation**]: Design-style (e.g., parallel, pipelined, special-purpose); C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Multiple Data Stream Architectures (Multiprocessors)*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism*; H.2.1 [**Information Systems**]: Database Management—*Logical design*; H.2.8 [**Information Systems**]: Database Management—*Database Applications*; H.3 [**Information Systems**]: Information Storage and Retrieval

General Terms: Design, Measurement

Additional Keywords and Phrases: Parallelism, parallel database systems, parallel processing systems.

About the Authors:

DAVID DEWITT is a professor in the Computer Sciences Department at the University of Wisconsin. His current research interests include parallel database systems, object-oriented database systems, and database performance evaluation. **Author's Present Address:** Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706; email: dewitt@cs.wisc.edu

JIM GRAY is a staff member with the Digital Equipment Corporation. His current research interests include databases, transaction processing, and computer architecture. **Author's Present Address:** San Francisco Systems Center, Digital Equipment Corporation, 455 Market Street—7th Floor, San Francisco, CA 94105-2403; email: gray@sfbay.enet.dec.com

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grant DCR-8512862, and by research grants from Digital Equipment Corporation, IBM, NCR, Tandem, and Intel Scientific Computers.

© ACM 0002-0782/92/0600-085 $1.50

Access Path Selection in a Relational Database Management System

P. Griffiths Selinger
M. M. Astrahan
D. D. Chamberlin
R. A. Lorie
T. G. Price

IBM Research Division, San Jose, California 95193

ABSTRACT: In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a user specification of desired data as a boolean expression of predicates. System R is an experimental database management system developed to carry out research on the relational model of data. System R was designed and built by members of the IBM San Jose Research Laboratory.

1. Introduction

System R is an experimental database management system based on the relational model of data which has been under development at the IBM San Jose Research Laboratory since 1975 <1>. The software was developed as a research vehicle in relational database, and is not generally available outside the IBM Research Division.

This paper assumes familiarity with relational data model terminology as described in Codd <7> and Date <8>. The user interface in System R is the unified query, data definition, and manipulation language SQL <5>. Statements in SQL can be issued both from an on-line casual-user-oriented terminal interface and from programming languages such as PL/I and COBOL.

In System R a user need not know how the tuples are physically stored and what access paths are available (e.g. which columns have indexes). SQL statements do not require the user to specify anything about the access path to be used for tuple retrieval. Nor does a user specify in what order joins are to be performed. The System R optimizer chooses both join order and an access path for each table in the SQL statement. Of the many possible choices, the optimizer chooses the one which minimizes "total access cost" for performing the entire statement.

This paper will address the issues of access path selection for queries. Retrieval for data manipulation (UPDATE, DELETE) is treated similarly. Section 2 will describe the place of the optimizer in the processing of a SQL statement, and section 3 will describe the storage component access paths that are available on a single physically stored table. In section 4 the optimizer cost formulas are introduced for single table queries, and section 5 discusses the joining of two or more tables, and their corresponding costs. Nested queries (queries in predicates) are covered in section 6.

2. Processing of an SQL statement

A SQL statement is subjected to four phases of processing. Depending on the origin and contents of the statement, these phases may be separated by arbitrary intervals of time. In System R, these arbitrary time intervals are transparent to the system components which process a SQL statement. These mechanisms and a description of the processing of SQL statements from both programs and terminals are further discussed in <2>. Only an overview of those processing steps that are relevant to access path selection will be discussed here.

The four phases of statement processing are parsing, optimization, code generation, and execution. Each SQL statement is sent to the parser. where it is checked for correct syntax. A query block is represented by a SELECT list, a FROM list, and a WHERE tree, containing, respectively the list of items to be retrieved, the table(s) referenced, and the boolean combination of simple predicates specified by the user. A single SQL statement may have many query blocks because a predicate may have one

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1979 ACM 0-89791-001-X/79/0500-0023 $00.75

operand which is itself a query.

If the parser returns without any errors detected, the OPTIMIZER component is called. The OPTIMIZER accumulates the names of tables and columns referenced in the query and looks them up in the System R catalogs to verify their existence and to retrieve information about them.

The catalog lookup portion of the OPTIMIZER also obtains statistics about the referenced relations, and the access paths available on each of them. These will be used later in access path selection. After catalog lookup has obtained the datatype and length of each column, the OPTIMIZER rescans the SELECT-list and WHERE-tree to check for semantic errors and type compatibility in both expressions and predicate comparisons.

Finally the OPTIMIZER performs access path selection. It first determines the evaluation order among the query blocks in the statement. Then for each query block, the relations in the FROM list are processed. If there is more than one relation in a block, permutations of the join order and of the method of joining are evaluated. The access paths that minimize total cost for the block are chosen from a tree of alternate path choices. This minimum cost solution is represented by a structural modification of the parse tree. The result is an execution plan in the Access Specification Language (ASL) <10>.

After a plan is chosen for each query block and represented in the parse tree, the CODE GENERATOR is called. The CODE GENERATOR is a table-driven program which translates ASL trees into machine language code to execute the plan chosen by the OPTIMIZER. In doing this it uses a relatively small number of code templates, one for each type of join method (including no join). Query blocks for nested queries are treated as "subroutines" which return values to the predicates in which they occur. The CODE GENERATOR is further described in <9>.

During code generation, the parse tree is replaced by executable machine code and its associated data structures. Either control is immediately transfered to this code or the code is stored away in the database for later execution, depending on the origin of the statement (program or terminal). In either case, when the code is ultimately executed, it calls upon the System R internal storage system (RSS) via the storage system interface (RSI) to scan each of the physically stored relations in the query. These scans are along the access paths chosen by the OPTIMIZER. The RSI commands that may be used by generated code are described in the next section.

3. The Research Storage System

The Research Storage System (RSS) is the storage subsystem of System R. It is responsible for maintaining physical storage of relations, access paths on these relations, locking (in a multi-user environment), and logging and recovery facilities. The RSS presents a tuple-oriented interface (RSI) to its users. Although the RSS may be used independently of System R, we are concerned here with its use for executing the code generated by the processing of SQL statements in System R, as described in the previous section. For a complete description of the RSS, see <1>.

Relations are stored in the RSS as a collection of tuples whose columns are physically contiguous. These tuples are stored on 4K byte pages; no tuple spans a page. Pages are organized into logical units called segments. Segments may contain one or more relations, but no relation may span a segment. Tuples from two or more relations may occur on the same page. Each tuple is tagged with the identification of the relation to which it belongs.

The primary way of accessing tuples in a relation is via an RSS scan. A scan returns a tuple at a time along a given access path. OPEN, NEXT, and CLOSE are the principal commands on a scan.

Two types of scans are currently available for SQL statements. The first type is a segment scan to find all the tuples of a given relation. A series of NEXTs on a segment scan simply examines all pages of the segment which contain tuples from <u>any</u> relation, and returns those tuples belonging to the given relation.

The second type of scan is an index scan. An index may be created by a System R user on one or more columns of a relation, and a relation may have any number (including zero) of indexes on it. These indexes are stored on separate pages from those containing the relation tuples. Indexes are implemented as B-trees <3>, whose leaves are pages containing sets of (key, identifiers of tuples which contain that key). Therefore a series of NEXTs on an index scan does a sequential read along the leaf pages of the index, obtaining the tuple identifiers matching a key, and using them to find and return the data tuples to the user in key value order. Index leaf pages are chained together so that NEXTs need not reference any upper level pages of the index.

In a segment scan, all the non-empty pages of a segment will be touched, regardless of whether there are any tuples from the desired relation on them. However, each page is touched only once. When an entire relation is examined via an index scan, each page of the index is touched

only once, but a data page may be examined more than once if it has two tuples on it which are not "close" in the index ordering. If the tuples are inserted into segment pages in the index ordering, and if this physical proximity corresponding to index key value is maintained, we say that the index is clustered. A clustered index has the property that not only each index page, but also each data page containing a tuple from that relation will be touched only once in a scan on that index.

An index scan need not scan the entire relation. Starting and stopping key values may be specified in order to scan only those tuples which have a key in a range of index values. Both index and segment scans may optionally take a set of predicates, called search arguments (or SARGS), which are applied to a tuple before it is returned to the RSI caller. If the tuple satisfies the predicates, it is returned; otherwise the scan continues until it either finds a tuple which satisfies the SARGS or exhausts the segment or the specified index value range. This reduces cost by eliminating the overhead of making RSI calls for tuples which can be efficiently rejected within the RSS. Not all predicates are of the form that can become SARGS. A sargable predicate is one of the form (or which can be put into the form) "column comparison-operator value". SARGS are expressed as a boolean expression of such predicates in disjunctive normal form.

4. Costs for single relation access paths

In the next several sections we will describe the process of choosing a plan for evaluating a query. We will first describe the simplest case, accessing a single relation, and show how it extends and generalizes to 2-way joins of relations, n-way joins, and finally multiple query blocks (nested queries).

The OPTIMIZER examines both the predicates in the query and the access paths available on the relations referenced by the query, and formulates a cost prediction for each access plan, using the following cost formula:

COST = PAGE FETCHES + W * (RSI CALLS).

This cost is a weighted measure of I/O (pages fetched) and CPU utilization (instructions executed). W is an adjustable weighting factor between I/O and CPU. RSI CALLS is the predicted number of tuples returned from the RSS. Since most of System R's CPU time is spent in the RSS, the number of RSI calls is a good approximation for CPU utilization. Thus the choice of a minimum cost path to process a query attempts to minimize total resources required.

During execution of the type-compatibility and semantic checking portion of the OPTIMIZER, each query block's WHERE tree of predicates is examined. The WHERE tree is considered to be in conjunctive normal form, and every conjunct is called a boolean factor. Boolean factors are notable because every tuple returned to the user must satisfy every boolean factor. An index is said to match a boolean factor if the boolean factor is a sargable predicate whose referenced column is the index key; e.g., an index on SALARY matches the predicate 'SALARY = 20000'. More precisely, we say that a predicate or set of predicates matches an index access path when the predicates are sargable and the columns mentioned in the predicate(s) are an initial substring of the set of columns of the index key. For example, a NAME, LOCATION index matches NAME = 'SMITH' AND LOCATION = 'SAN JOSE'. If an index matches a boolean factor, an access using that index is an efficient way to satisfy the boolean factor. Sargable boolean factors can also be efficiently satisfied if they are expressed as search arguments. Note that a boolean factor may be an entire tree of predicates headed by an OR.

During catalog lookup, the OPTIMIZER retrieves statistics on the relations in the query and on the access paths available on each relation. The statistics kept are the following:

For each relation T,
- NCARD(T), the cardinality of relation T.
- TCARD(T), the number of pages in the segment that hold tuples of relation T.
- P(T), the fraction of data pages in the segment that hold tuples of relation T.
  P(T) = TCARD(T) / (no. of non-empty pages in the segment).

For each index I on relation T,
- ICARD(I), number of distinct keys in index I.
- NINDX(I), the number of pages in index I.

These statistics are maintained in the System R catalogs, and come from several sources. Initial relation loading and index creation initialize these statistics. They are then updated periodically by an UPDATE STATISTICS command, which can be run by any user. System R does not update these statistics at every INSERT, DELETE, or UPDATE because of the extra database operations and the locking bottleneck this would create at the system catalogs. Dynamic updating of statistics would tend to serialize accesses that modify the relation contents.

Using these statistics, the OPTIMIZER assigns a selectivity factor 'F' for each boolean factor in the predicate list. This selectivity factor very roughly corresponds to the expected fraction of tuples which will satisfy the predicate. TABLE 1 gives the selectivity factors for different kinds of predicates. We assume that a lack of statistics implies that the relation is small, so an arbitrary factor is chosen.

TABLE 1 SELECTIVITY FACTORS

column = value

F = 1 / ICARD(column index) if there is an index on column

This assumes an even distribution of tuples among the index key values.

F = 1/10 otherwise

column1 = column2

F = 1/MAX(ICARD(column1 index), ICARD(column2 index))

if there are indexes on both column1 and column2

This assumes that each key value in the index with the smaller cardinality has a matching value in the other index.

F = 1/ICARD(column-i index) if there is only an index on column-i

F = 1/10 otherwise

column > value (or any other open-ended comparison)

F = (high key value - value) / (high key value - low key value)

Linear interpolation of the value within the range of key values yields F if the column is an arithmetic type and value is known at access path selection time.

F = 1/3 otherwise (i.e. column not arithmetic)

There is no significance to this number, other than the fact that it is less selective than the guesses for equal predicates for which there are no indexes, and that it is less than 1/2. We hypothesize that few queries use predicates that are satisfied by more than half the tuples.

column BETWEEN value1 AND value2

F = (value2 - value1) / (high key value - low key value)

A ratio of the BETWEEN value range to the entire key value range is used as the selectivity factor if column is arithmetic and both value1 and value2 are known at access path selection.

F = 1/4 otherwise

Again there is no significance to this choice except that it is between the default selectivity factors for an equal predicate and a range predicate.

column IN (list of values)

F = (number of items in list) * (selectivity factor for column = value)

This is allowed to be no more than 1/2.

columnA IN subquery

F = (expected cardinality of the subquery result) /

(product of the cardinalities of all the relations in the subquery's FROM-list).

The computation of query cardinality will be discussed below.

This formula is derived by the following argument:

Consider the simplest case, where subquery is of the form "SELECT columnB FROM relationC ...". Assume that the set of all columnB values in relationC contains the set of all columnA values. If all the tuples of relationC are selected by the subquery, then the predicate is always TRUE and F = 1. If the tuples of the subquery are restricted by a selectivity factor F', then assume that the set of unique values in the subquery result that match columnA values is proportionately restricted, i.e. the selectivity factor for the predicate should be F'. F' is the product of all the subquery's selectivity factors, namely (subquery cardinality) / (cardinality of all possible subquery answers). With a little optimism, we can extend this reasoning to include subqueries which are joins and subqueries in which columnB is replaced by an arithmetic expression involving column names. This leads to the formula given above.

(pred expression1) OR (pred expression2)

F = F(pred1) + F(pred2) - F(pred1) * F(pred2)

(pred1) AND (pred2)

F = F(pred1) * F(pred2)

Note that this assumes that column values are independent.

NOT pred

F = 1 - F(pred)

Query cardinality (QCARD) is the product of the cardinalities of every relation in the query block's FROM list times the product of all the selectivity factors of that query block's boolean factors. The number of expected RSI calls (RSICARD) is the product of the relation cardinalities times the selectivity factors of the sargable boolean factors, since the sargable boolean factors will be put into search arguments which will filter out tuples without returning across the RSS interface.

Choosing an optimal access path for a single relation consists of using these selectivity factors in formulas together with the statistics on available access paths. Before this process is described, a definition is needed. Using an index access path or sorting tuples produces tuples in the index value or sort key order. We say that a tuple order is an interesting order if that order is one specified by the query block's GROUP BY or ORDER BY clauses.

For single relations, the cheapest access path is obtained by evaluating the cost for each available access path (each index on the relation, plus a segment scan). The costs will be described below. For each such access path, a predicted cost is computed along with the ordering of the tuples it will produce. Scanning along the SALARY index in ascending order, for example, will produce some cost C and a tuple order of SALARY (ascending). To find the cheapest access plan for a single relation query, we need only to examine the cheapest access path which produces tuples in each "interesting" order and the cheapest "unordered" access path. Note that an "unordered" access path may in fact produce tuples in some order, but the order is not "interesting". If there are no GROUP BY or ORDER BY clauses on the query, then there will be no interesting orderings, and the cheapest access path is the one chosen. If there are GROUP BY or ORDER BY clauses, then the cost for producing that interesting ordering must be compared to the cost of the cheapest unordered path plus the cost of sorting QCARD tuples into the proper order. The cheapest of these alternatives is chosen as the plan for the query block.

The cost formulas for single relation access paths are given in TABLE 2. These formulas give index pages fetched plus data pages fetched plus the weighting factor times RSI tuple retrieval calls. W is the weighting factor between page fetches and RSI calls. Some situations give several alternative formulas depending on whether the set of tuples retrieved will fit entirely in the RSS buffer pool (or effective buffer pool per user). We assume for clustered indexes that a page remains in the buffer long enough for every tuple to be retrieved from it. For non-clustered indexes, it is assumed that for those relations not fitting in the buffer, the relation is sufficiently large with respect to the buffer size that a page fetch is required for every tuple retrieval.

TABLE 2 COST FORMULAS

SITUATION	COST (in pages)
Unique index matching an equal predicate	1 + 1 + W
Clustered index I matching one or more boolean factors	F(preds) * (NINDX(I) + TCARD) + W * RSICARD
Non-clustered index I matching one or more boolean factors	F(preds) * (NINDX(I) + NCARD) + W * RSICARD or F(preds) * (NINDX(I) + TCARD) + W * RSICARD if this number fits in the System R buffer
Clustered index I not matching any boolean factors	(NINDX(I) + TCARD) + W * RSICARD
Non-clustered index I not matching any boolean factors	(NINDX(I) + NCARD) + W * RSICARD or (NINDX(I) + TCARD) + W * RSICARD if this number fits in the System R buffer
Segment scan	TCARD/P + W * RSICARD

5. Access path selection for joins

In 1976, Blasgen and Eswaran <4> examined a number of methods for performing 2-way joins. The performance of each of these methods was analyzed under a variety of relation cardinalities. Their evidence indicates that for other than very small relations, one of two join methods were always optimal or near optimal. The System R optimizer chooses between these two methods. We first describe these methods, and then discuss how they are extended for n-way joins. Finally we specify how the join order (the order in which the relations are joined) is chosen. For joins involving two relations, the two relations are called the outer relation, from which a tuple will be retrieved first, and the inner relation, from which tuples will be retrieved, possibly depending on the values obtained in the outer relation tuple. A predicate which relates columns of two tables to be joined is called a join predicate. The columns referenced in a join predicate are called join columns.

The first join method, called the nested loops method, uses scans, in any order, on the outer and inner relations. The scan on the outer relation is opened and the first tuple is retrieved. For each outer relation tuple obtained, a scan is opened on the inner relation to retrieve, one at a time, all the tuples of the inner relation which satisfy the join predicate. The composite tuples formed by the outer-relation-tuple / inner-relation-tuple pairs comprise the result of this join.

The second join method, called merging scans, requires the outer and inner relations to be scanned in join column order. This implies that, along with the columns mentioned in ORDER BY and GROUP BY, columns of equi-join predicates (those of the form Table1.column1 = Table2.column2) also define "interesting" orders. If there is more than one join predicate, one of them is used as the join predicate and the others are treated as ordinary predicates. The merging scans method is only applied to equi-joins, although in principle it could be applied to other types of joins. If one or both of the relations to be joined has no indexes on the join column, it must be sorted into a temporary list which is ordered by the join column.

The more complex logic of the merging scan join method takes advantage of the ordering on join columns to avoid rescanning the entire inner relation (looking for a match) for each tuple of the outer relation. It does this by synchronizing the inner and outer scans by reference to matching join column values and by "remembering" where matching join groups are located. Further savings occur if the inner relation is clustered on the join column (as would be true if it is the output of a sort on the join column). "Clustering" on a column means that tuples which have the same value in that column are physically stored close to each other so that one page access will retrieve several tuples.

N-way joins can be visualized as a sequence of 2-way joins. In this visualization, two relations are joined together, the resulting composite relation is joined with the third relation, etc. At each step of the n-way join it is possible to identify the outer relation (which in general is composite) and the inner relation (the relation being added to the join). Thus the methods described above for two way joins are easily generalized to n-way joins. However, it should be emphasized that the first 2-way join does not have to be completed before the second 2-way join is started. As soon as we get a composite tuple for the first 2-way join, it can be joined with tuples of the third relation to form result tuples for the 3-way join, etc. Nested loop joins and merge scan joins may be mixed in the same query, e.g. the first two relations of a three-way join may be joined using merge scans and the composite result may be joined with the third relation using a nested loop join. The intermediate composite relations are physically stored only if a sort is required for the next join step. When a sort of the composite relation is not specified, the composite relation will be materialized one tuple at a time to participate in the next join.

We now consider the order in which the relations are chosen to be joined. It should be noted that although the cardinality of the join of n relations is the same regardless of join order, the cost of joining in different orders can be substantially different. If a query block has n relations in its FROM list, then there are n factorial permutations of relation join orders. The search space can be reduced by observing that that once the first k relations are joined, the method to join the composite to the k+1-st relation is independent of the order of joining the first k; i.e. the applicable predicates are the same, the set of interesting orderings is the same, the possible join methods are the same, etc. Using this property, an efficient way to organize the search is to find the best join order for successively larger subsets of tables.

A heuristic is used to reduce the join order permutations which are considered. When possible, the search is reduced by consideration only of join orders which have join predicates relating the inner relation to the other relations already participating in the join. This means that in joining relations t1,t2,...,tn only those orderings ti1,ti2,...,tin are examined in which for all j (j=2,...,n) either

(1) tij has at least one join predicate

with some relation tik, where k < j, or
(2) for all k > j, tik has no join predicate with ti1,ti2,...,or ti(j-1).
This means that all joins requiring Cartesian products are performed as late in the join sequence as possible. For example, if T1,T2,T3 are the three relations in a query block's FROM list, and there are join predicates between T1 and T2 and between T2 and T3 on different columns than the T1-T2 join, then the following permutations are not considered:

T1-T3-T2
T3-T1-T2

To find the optimal plan for joining n relations, a tree of possible solutions is constructed. As discussed above, the search is performed by finding the best way to join subsets of the relations. For each set of relations joined, the cardinality of the composite relation is estimated and saved. In addition, for the unordered join, and for each interesting order obtained by the join thus far, the cheapest solution for achieving that order and the cost of that solution are saved. A solution consists of an ordered list of the relations to be joined, the join method used for each join, and a plan indicating how each relation is to be accessed. If either the outer composite relation or the inner relation needs to be sorted before the join, then that is also included in the plan. As in the single relation case, "interesting" orders are those listed in the query block's GROUP BY or ORDER BY clause, if any. Also every join column defines an "interesting" order. To minimize the number of different interesting orders and hence the number of solutions in the tree, equivalence classes for interesting orders are computed and only the best solution for each equivalence class is saved. For example, if there is a join predicate E.DNO = D.DNO and another join predicate D.DNO = F.DNO, then all three of these columns belong to the same order equivalence class.

The search tree is constructed by iteration on the number of relations joined so far. First, the best way is found to access each single relation for each interesting tuple ordering and for the unordered case. Next, the best way of joining any relation to these is found, subject to the heuristics for join order. This produces solutions for joining pairs of relations. Then the best way to join sets of three relations is found by consideration of all sets of two relations and joining in each third relation permitted by the join order heuristic. For each plan to join a set of relations, the order of the composite result is kept in the tree. This allows consideration of a merge scan join which would not require sorting the composite. After the complete solutions (all of the relations joined together) have been found, the optimizer chooses the cheapest solution which gives the required order, if any was specified. Note that if a solution exists with the correct order, no sort is performed for ORDER BY or GROUP BY, unless the ordered solution is more expensive than the cheapest unordered solution plus the cost of sorting into the required order.

The number of solutions which must be stored is at most 2**n (the number of subsets of n tables) times the number of interesting result orders. The computation time to generate the tree is approximately proportional to the same number. This number is frequently reduced substantially by the join order heuristic. Our experience is that typical cases require only a few thousand bytes of storage and a few tenths of a second of 370/158 CPU time. Joins of 8 tables have been optimized in a few seconds.

Computation of costs

The costs for joins are computed from the costs of the scans on each of the relations and the cardinalities. The costs of the scans on each of the relations are computed using the cost formulas for single relation access paths presented in section 4.

Let C-outer(path1) be the cost of scanning the outer relation via path1, and N be the cardinality of the outer relation tuples which satisfy the applicable predicates. N is computed by:

N = (product of the cardinalities of all relations T of the join so far) * (product of the selectivity factors of all applicable predicates).

Let C-inner(path2) be the cost of scanning the inner relation, applying all applicable predicates. Note that in the merge scan join this means scanning the contiguous group of the inner relation which corresponds to one join column value in the outer relation. Then the cost of a nested loop join is

C-nested-loop-join(path1,path2)=
C-outer(path1) + N * C-inner(path2)

The cost of a merge scan join can be broken up into the cost of actually doing the merge plus the cost of sorting the outer or inner relations, if required. The cost of doing the merge is

C-merge(path1,path2)=
C-outer(path1) + N * C-inner(path2)

For the case where the inner relation is sorted into a temporary relation none of the single relation access path formulas in section 4 apply. In this case the inner scan is like a segment scan except that the merging scans method makes use of the fact that the inner relation is sorted so that it is not necessary to scan the entire inner relation looking for a match. For this case we use the following formula for the cost of the inner scan.

C-inner(sorted list) =
TEMPPAGES/N + W*RSICARD

where TEMPPAGES is the number of pages

required to hold the inner relation. This formula assumes that during the merge each page of the inner relation is fetched once.

It is interesting to observe that the cost formula for nested loop joins and the cost formula for merging scans are essentially the same. The reason that merging scans is sometimes better than nested loops is that the cost of the inner scan may be much less. After sorting, the inner relation is clustered on the join column which tends to minimize the number of pages fetched, and it is not necessary to scan the entire inner relation (looking for a match) for each tuple of the outer relation.

The cost of sorting a relation, C-sort(path), includes the cost of retrieving the data using the specified access path, sorting the data, which may involve several passes, and putting the results into a temporary list. Note that prior to sorting the inner table, only the local predicates can be applied. Also, if it is necessary to sort a composite result, the entire composite relation must be stored in a temporary relation before it can be sorted. The cost of inserting the composite tuples into a temporary relation before sorting is included in C-sort(path).

Example of tree

We now show how the search is done for the example join shown in Fig. 1. First we find all of the reasonable access paths for single relations with only their local predicates applied. The results for this example are shown in Fig. 2. There are three access paths for the EMP table: an index on DNO, an index on JOB, and a segment scan. The interesting orders are DNO and JOB. The index on DNO provides the tuples in DNO order and the index on JOB provides the tuples in JOB order. The segment scan access path is, for our purposes, unordered. For this example we assume that the index on JOB is the cheapest path, so the segment scan path is pruned. For the DEPT relation there are two access paths, an index on DNO and a segment scan. We assume that the index on DNO is cheaper so the segment scan path is pruned. For the JOB relation there are two access paths, an index on JOB and a segment scan. We assume that the segment scan path is cheaper, so both paths are saved. The results just described are saved in the search tree as shown in Fig. 3. In the figures, the notation C(EMP.DNO) or C(E.DNO) means the cost of scanning EMP via the DNO index, applying all predicates which are applicable given that tuples from the specified set of relations have already been fetched. The notation Ni is used to represent the cardinalities of the different partial results.

Next, solutions for pairs of relations are found by joining a second relation to

EMP

NAME	DNO	JOB	SAL
SMITH	50	12	8500
JONES	50	5	15000
DOE	51	5	9500

DEPT

DNO	DNAME	LOC
50	MFG	DENVER
51	BILLING	BOULDER
52	SHIPPING	DENVER

JOB

JOB	TITLE
5	CLERK
6	TYPIST
9	SALES
12	MECHANIC

```
SELECT  NAME, TITLE, SAL, DNAME
FROM    EMP, DEPT, JOB
WHERE   TITLE='CLERK'
AND     LOC='DENVER'
AND     EMP.DNO=DEPT.DNO
AND     EMP.JOB=JOB.JOB
```

"Retrieve the name, salary, job title, and department name of employees who are clerks and work for departments in Denver."

Figure 1. JOIN example

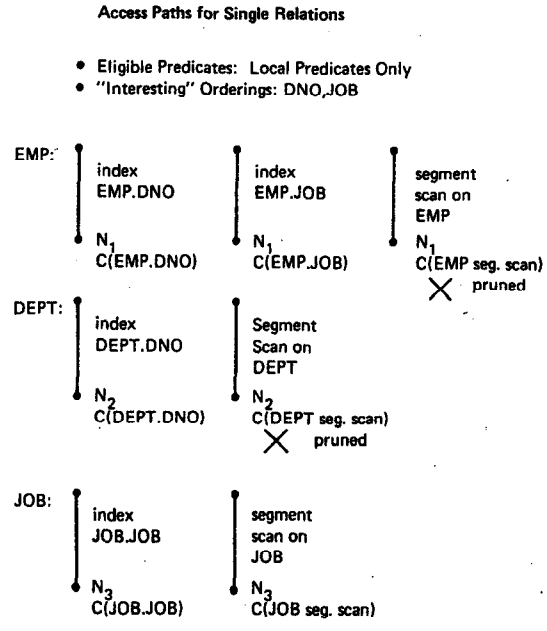


Figure 2.

the results for single relations shown in Fig. 3. For each single relation, we find access paths for joining in each second relation for which there exists a predicate connecting it to the first relation. First we consider access path selection for nested loop joins. In this example we assume that the EMP-JOB join is cheapest by accessing JOB on the JOB index. This is

likely since it can fetch directly the tuples with matching JOB (without having to scan the entire relation). In practice the cost of joining is estimated using the formulas given earlier and the cheapest path is chosen. For joining the EMP relation to the DEPT relation we assume that the DNO index is cheapest. The best access path for each second-level relation is combined with each of the plans in Fig. 3 to form the nested loop solutions shown in Fig. 4.

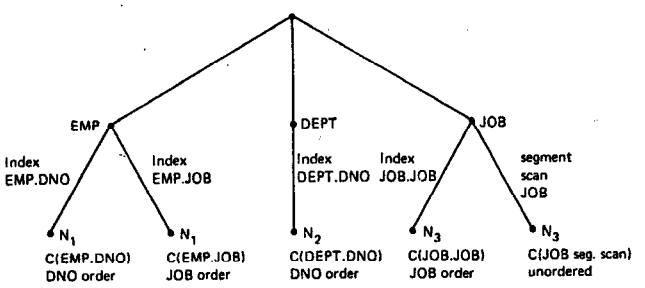


Figure 3. Search tree for single relations

Next we generate solutions using the merging scans method. As we see on the left side of Fig. 3, there is a scan on the EMP relation in DNO order, so it is possible to use this scan and the DNO scan on the DEPT relation to do a merging scans join, without any sorting. Although it is possible to do the merging join without sorting as just described, it might be cheaper to use the JOB index on EMP, sort on DNO, and then merge. Note that we never consider sorting the DEPT table because the cheapest scan on that table is already in DNO order.

For merging JOB with EMP, we only consider the JOB index on EMP since it is the cheapest access path for EMP regardless of order. Using the JOB index on JOB, we can merge without any sorting. However, it might be cheapter to sort JOB using a relation scan as input to the sort and then do the merge.

Referring to Fig. 3, we see that the access path chosen for the the DEPT relation is the DNO index. After accessing DEPT via this index, we can merge with EMP using the DNO index on EMP, again without any sorting. However, it might be cheaper to sort EMP first using the JOB index as input to the sort and then do the merge. Both of these cases are shown in Fig. 5.

As each of the costs shown in Figs. 4 and 5 are computed they are compared with the cheapest equivalent solution (same tables and same result order) found so far, and the cheapest solution is saved. After this pruning, solutions for all three relations are found. For each pair of relations, we find access paths for joining in the remaining third relation. As before we will extend the tree using nested loop joins and merging scans to join the third relation. The search tree for three relations is shown in Fig. 6. Note that in one case both the composite relation and the table being added (JOB) are sorted. Note also that for some of the cases no sorts are performed at all. In these cases, the composite result is materialized one tuple at a time and the intermediate composite relation is never stored. As before, as each of the costs are computed they are compared with the cheapest solu-

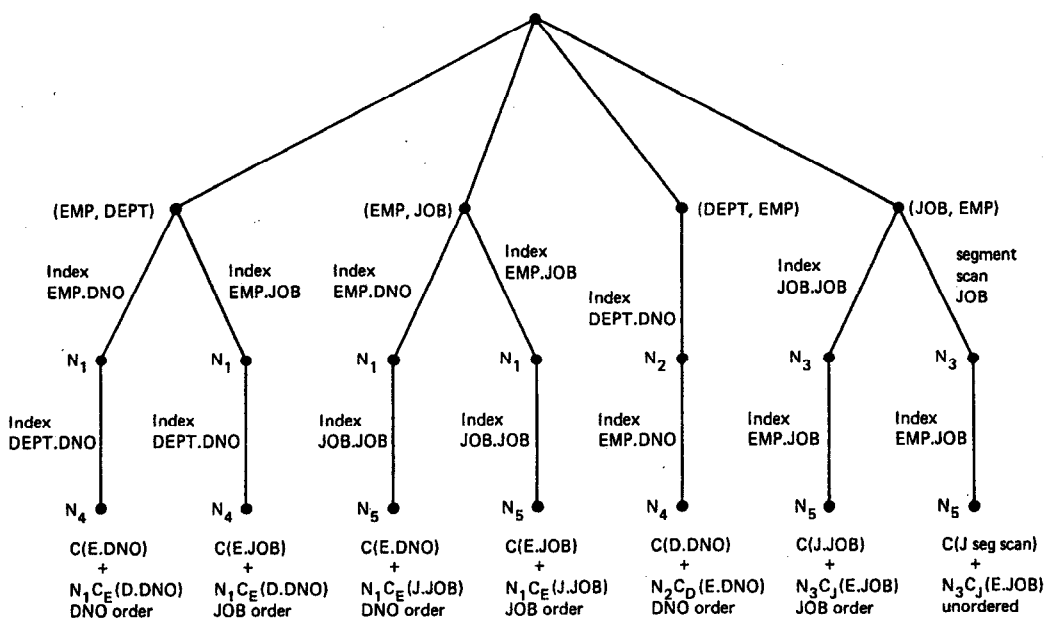


Figure 4. Extended search tree for second relation (nested loop join)

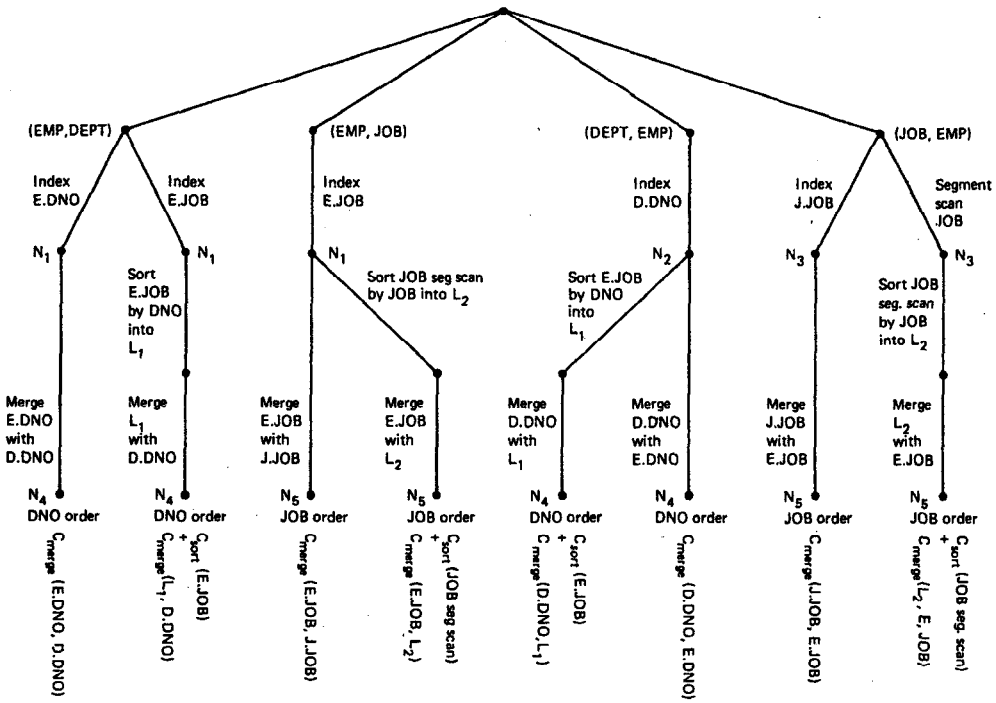


Figure 5. Extended search tree for second relation (merge join)

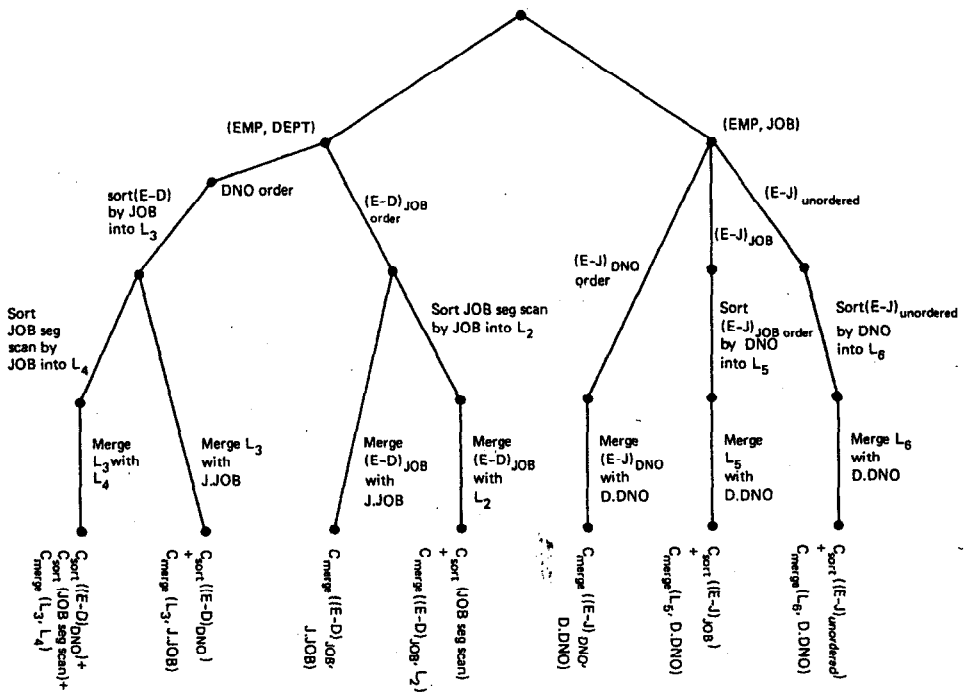


Figure 6. Extended search tree for third relation

6. Nested Queries

A query may appear as an operand of a predicate of the form "expression operator query". Such a query is called a Nested Query or a Subquery. If the operator is one of the six scalar comparisons (=, ¬=, >, >=, <, <=), then the subquery must return a single value. The following example using the "=" operator was given in section 2:

```
SELECT NAME
FROM EMPLOYEE
WHERE SALARY =
   (SELECT AVG(SALARY)
    FROM EMPLOYEE)
```

If the operator is IN or NOT IN then the subquery may return a set of values. For example:

```
SELECT NAME
FROM EMPLOYEE
WHERE DEPARTMENT_NUMBER IN
    (SELECT DEPARTMENT_NUMBER
     FROM DEPARTMENT
     WHERE LOCATION='DENVER')
```

In both examples, the subquery needs to be evaluated only once. The OPTIMIZER will arrange for the subquery to be evaluated before the top level query is evaluated. If a single value is returned, it is incorporated into the top level query as though it had been part of the original query statement; for example, if AVG(SAL) above evaluates to 15000 at execution time, then the predicate becomes "SALARY = 15000". If the subquery can return a set of values, they are returned in a temporary list, an internal form which is more efficient than a relation but which can only be accessed sequentially. In the example above, if the subquery returns the list (17,24) then the predicate is evaluated in a manner similar to the way in which it would have been evaluated if the original predicate had been DEPARTMENT_NUMBER IN (17,24).

A subquery may also contain a predicate with a subquery, down to a (theoretically) arbitrary level of nesting. When such subqueries do not reference columns from tables in higher level query blocks, they are all evaluated before the top level query is evaluated. In this case, the most deeply nested subqueries are evaluated first, since any subquery must be evaluated before its parent query can be evaluated.

A subquery may contain a reference to a value obtained from a candidate tuple of a higher level query block (see example below). Such a query is called a <u>correlation</u> subquery. A correlation subquery must in principle be re-evaluated for each candidate tuple from the referenced query block. This re-evaluation must be done before the correlation subquery's parent predicate in the higher level block can be tested for acceptance or rejection of the candidate tuple. As an example, consider the query:

```
SELECT NAME
FROM EMPLOYEE X
WHERE SALARY > (SELECT SALARY
                FROM EMPLOYEE
                WHERE EMPLOYEE_NUMBER=
                      X.MANAGER)
```

This selects names of EMPLOYEE's that earn more than their MANAGER. Here X identifies the query block and relation which furnishes the candidate tuple for the correlation. For each candidate tuple of the top level query block, the MANAGER value is used for evaluation of the subquery. The subquery result is then returned to the "SALARY >" predicate for testing acceptance of the candidate tuple.

If a correlation subquery is not directly below the query block it references but is separated from that block by one or more intermediate blocks, then the correlation subquery evaluation will be done before evaluation of the highest of the intermediate blocks. For example:

```
level 1   SELECT NAME
          FROM EMPLOYEE X
          WHERE SALARY >
level 2        (SELECT SALARY
                FROM EMPLOYEE
                WHERE EMPLOYEE_NUMBER =
level 3             (SELECT MANAGER
                     FROM EMPLOYEE
                     WHERE EMPLOYEE_NUMBER =
                           X.MANAGER))
```

This selects names of EMPLOYEE's that earn more than their MANAGER's MANAGER. As before, for each candidate tuple of the level-1 query block, the EMPLOYEE.MANAGER value is used for evaluation of the level-3 query block. In this case, because the level 3 subquery references a level 1 value but does not reference level 2 values, it is evaluated once for every new level 1 candidate tuple, but not for every level 2 candidate tuple.

If the value referenced by a correlation subquery (X.MANAGER above) is not unique in the set of candidate tuples (e.g., many employees have the same manager), the procedure given above will still cause the subquery to be re-evaluated for each occurrence of a replicated value. However, if the referenced relation is ordered on the referenced column, the re-evaluation can be made conditional, depending on a test of whether or not the current referenced value is the same as the one in the previous candidate tuple. If they are the same, the previous evaluation result can be used again. In some cases, it might even pay to sort the referenced relation on the referenced column in order to avoid re-evaluating subqueries unnecessarily. In order to determine whether or not the referenced column values are unique, the OPTIMIZER can use clues like NCARD > ICARD, where NCARD is the relation cardinality and ICARD is the index cardinality of an index on the referenced column.

7. Conclusion

The System R access path selection has been described for single table queries, joins, and nested queries. Evaluation work on comparing the choices made to the "right" choice is in progress, and will be described in a forthcoming paper. Preliminary results indicate that, although the costs predicted by the optimizer are often not accurate in absolute value, the true optimal path is selected in a large majority of cases. In many cases, the ordering among the estimated costs for all paths considered is precisely the same as that among the actual measured costs.

Furthermore, the cost of path selection is not overwhelming. For a two-way join, the cost of optimization is approximately equivalent to between 5 and 20 database retrievals. This number becomes even more insignificant when such a path selector is placed in an environment such as System R, where application programs are compiled once and run many times. The cost of optimization is amortized over many runs.

The key contributions of this path selector over other work in this area are the expanded use of statistics (index cardinality, for example), the inclusion of CPU utilization into the cost formulas, and the method of determining join order. Many queries are CPU-bound, particularly merge joins for which temporary relations are created and sorts performed. The concept of "selectivity factor" permits the optimizer to take advantage of as many of the query's restriction predicates as possible in the RSS search arguments and access paths. By remembering "interesting ordering" equivalence classes for joins and ORDER or GROUP specifications, the optimizer does more bookkeeping than most path selectors, but this additional work in many cases results in avoiding the storage and sorting of intermediate query results. Tree pruning and tree searching techniques allow this additional bookkeeping to be performed efficiently.

More work on validation of the optimizer cost formulas needs to be done, but we can conclude from this preliminary work that database management systems can support non-procedural query languages with performance comparable to those supporting the current more procedural languages.

Cited and General References

<1> Astrahan, M. M. et al. System R: Relational Approach to Database Management. ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976, pp. 97-137.

<2> Astrahan, M. M. et al. System R: A Relational Database Management System. To appear in Computer.

<3> Bayer, R. and McCreight, E. Organization and Maintenance of Large Ordered Indices. Acta Informatica, Vol. 1, 1972.

<4> Blasgen, M.W. and Eswaran, K.P. On the Evaluation of Queries in a Relational Data Base System. IBM Research Report RJ1745, April, 1976.

<5> Chamberlin, D.D., et al. SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control. IBM Journal of Research and Development, Vol. 20, No. 6, Nov. 1976, pp. 560-575.

<6> Chamberlin, D.D., Gray, J.N., and Traiger, I.L. Views, Authorization and Locking in a Relational Data Base System. ACM National Computer Conference Proceedings, 1975, pp. 425-430.

<7> Codd, E.F. A Relational Model of Data for Large Shared Data Banks. ACM Communications, Vol. 13, No. 6, June, 1970, pp. 377-387.

<8> Date, C.J. An Introduction to Data Base Systems, Addison-Wesley, 1975.

<9> Lorie, R.A. and Wade, B.W. The Compilation of a Very High Level Data Language. IBM Research Report RJ2008, May, 1977.

<10> Lorie, R.A. and Nilsson, J.F. An Access Specification Language for a Relational Data Base System. IBM Research Report RJ2218, April, 1978.

<11> Stonebraker, M.R., Wong, E., Kreps, P., and Held, G.D. The Design and Implementation of INGRES. ACM Trans. on Database Systems, Vol. 1, No. 3, September, 1976, pp. 189-222.

<12> Todd, S. PRTV: An Efficient Implementation for Large Relational Data Bases. Proc. International Conf. on Very Large Data Bases, Framingham, Mass., September, 1975.

<13> Wong, E., and Youssefi, K. Decomposition - A Strategy for Query Processing. ACM Transactions on Database Systems, Vol. 1, No. 3 (Sept. 1976) pp. 223-241.

<14> Zloof, M.M. Query by Example. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 431-437.

Grammar-like Functional Rules for Representing Query Optimization Alternatives

Guy M. Lohman
IBM Almaden Research Center
San Jose, CA 95120

Abstract

Extensible query optimization requires that the "repertoire" of alternative strategies for executing queries be represented as data, not embedded in the optimizer code Recognizing that query optimizers are essentially expert systems, several researchers have suggested using strategy rules to transform query execution plans into alternative or better plans Though extremely flexible, these systems can be very inefficient at any step in the processing, many rules may be eligible for application and complicated conditions must be tested to determine that eligibility during unification We present a constructive, "building blocks" approach to defining alternative plans, in which the rules defining alternatives are an extension of the productions of a grammar to resemble the definition of a function in mathematics The extensions permit each token of the grammar to be parametrized and each of its alternative definitions to have a complex condition The terminals of the grammar are base-level database operations on tables that are interpreted at run-time The non-terminals are defined declaratively by production rules that combine those operations into meaningful plans for execution Each production produces a set of alternative plans, each having a vector of properties, including the estimated cost of producing that plan Productions can require certain properties of their inputs, such as tuple order and location, and we describe a "glue" mechanism for augmenting plans to achieve the required properties We give detailed examples to illustrate the power and robustness of our rules and to contrast them with related ideas

1. Introduction

Ever since the first query optimizers [WONG 76, SELI 79] were built for relational databases, revising the "repertoire" of ways to construct a procedural execution plan from a non-procedural query has required complicated and costly changes to the optimizer code itself This has limited the repertoire of any one optimizer by discouraging or slowing experimentation with — and implementation of — all the new advances in relational technology, such as improved join methods [BABB 79, BRAT 84, DEWI 85], distributed query optimization [EPST 78, CHU 82, DANI 82, LOHM 85], semijoins [BERN 81], Bloomjoins [BABB 79, MACK 86], parallel joins on fragments [WONG 83], join indexes [HAER 78, VALD 87], dynamic creation of indexes [MACK 86], and many other variations of traditional processing strategies The recent surge in interest in extensible database systems [STON 86, CARE 86, SCHW 86, BATO 86] has only exacerbated the burden on optimizers, adding the need to customize a database system for a particular class of applications, such as geographic [LOHM 83], CAD/CAM, or expert systems Now optimizers must adapt to new access methods, storage managers, data types, user-defined functions, etc , all combined in novel ways Clearly the traditional specification of all feasible strategies in the optimizer code cannot support such fluidity

Perhaps the most challenging aspect of extensible query optimization is the representation of alternative execution strategies Ideally, this representation should be readily understood and modified by the Database Customizer (DBC)[1] Recognizing that query optimizers are expert systems, several authors have observed that rules show great promise for this purpose [ULLM 85, FREY 87, GRAE 87a] Rules provide a high-level, *declarative* (i e , non-procedural), and compact specification of legal alternatives, which may be input *as data* to the optimizer and traced to explain the origin of any execution plan This makes it easy to modify the strategies without impacting the optimizer, and to encapsulate the strategies executable by a particular processor in a heterogeneous network But how should rules represent alternative strategies? The EXODUS project [GRAE 87a, GRAE 87b] and Freytag [FREY 87] use rules to transform a given execution plan into other feasible plans The NAIL! project [ULLM 85, MORR 86] employs "capture rules" to determine which of a set of available plans can be used to execute a query

In this paper, we use rules to describe how to *construct* — rather than to *alter* or to *match* — plans Our rules "compose" low-level database operations on tables (such as ACCESS, JOIN, and SORT) into higher-level operations that can be re-used in other definitions These constructive, "building blocks" rules, which resemble the productions of a grammar, have two major advantages over plan transformation rules

- **They are more readily understood**, because they enable the DBC to build increasingly complex plans from common building blocks, the details of which may be transparent to him, and
- **They can be processed more efficiently** during optimization, by simply finding the definition of any building block that is referenced, using a simple dictionary search, much as is done in macro expanders By contrast, plan transformation rules usually must

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery To copy otherwise, or to republish, requires a fee and/or specific permission

Reproduced by consent of IBM

© 1988 ACM 0-89791-268-3/88/0006/0018 $1 50

1 We feel this term more accurately describes the role of adapting an implemented but extensible database system than does the term *Database Implementor (DBI)*, coined by Carey et al [CARE 86]

examine a large set of rules and apply complicated conditions on each of a large set of plans generated thus far, in order to determine if that plan matches the pattern to which that rule applies As new rules create new patterns, existing rules may have to add conditions that deal with those new patterns

Our grammar-like approach is founded upon a few fundamental observations about query optimization

- **All database operators consume and produce a common object — a table**, viewed as a stream of tuples that is generated by accessing a table [BATO 87a] The output of one operation becomes the input of the next Streams from individual tables are merged by joins, eventually into a single stream [FREY 87, GRAE 87a]

- **Optimizers construct legal sequences of such operators that are understood by an interpreter, the *query evaluator*** In other words, the repertoire of legal plans is a *language* that might well be defined by a grammar

- **Decisions made by the optimizer have an inherent sequence dependency** that limits the scope of subsequent decisions [BATO 87a, FREY 87] For example, for a given plan, the order in which a given set of tables are joined must be determined before the access path for any of those tables is chosen, because the table order determines which predicates are eligible and hence might be applied by the access path of any table (commonly referred to as "pushing down the selection") Thus, for any set of tables, the rules for ordering table accesses must precede those for choosing the access path of each table, and the former serve to limit significantly which of the latter rules are applicable

- **Alternative plans may incorporate the same plan fragment,** whose alternatives need be evaluated only once This further limits the rules generating alternatives to just the *new* portions of the plan

- Unlike the simple pattern-matching of tokens to determine the applicability of productions in grammars, in query optimization **specifying the *conditions* under which a rule is applicable is usually harder than specifying the rule's *transformation*** For example, a multi-column index can apply one or more predicates only if the columns referenced in the predicates form a prefix of the columns in the index Assigning the predicates to be applied by the index is far easier to express than the condition that permits that assignment

These observations prompted us to use "strategy" rules to construct legal nestings of database operators declaratively, much as the productions of a grammar construct legal sequences of tokens However, our rules resemble more the definition of a function in mathematics or a rule in Prolog, in that the "tokens" of our grammar may be parametrized and their definition alternatives may have complex conditions The reader is cautioned that the *application* — not the representation — is our claim to novelty Logic programming uses rules to construct new relations from base relations [ULLM 85], whereas we are using rules to construct new operators from base operators that operate on tables

Our approach is a general one, but we will present it in the context of its intended use the Starburst prototype extensible database system, which is under development at the IBM Almaden Research Center [SCHW 86, LIND 87]

The paper is organized as follows Section 2 first defines the end-product of optimization — plans We describe what they're made of, what they look like, how our rules are used to construct all of them for a query In Section 3, we associate properties with plans, and allow rules to impose requirements on the properties of their input plans A set of possible rules for joins is given in Section 4 to illustrate the power of our rules to specify some of the most complicated strategies of existing systems, including several not addressed by other authors Section 5 outlines how the DBC can make extensions to rules, properties, and database operators Having thoroughly described our approach, we contrast it with related work in Section 6, and conclude in Section 7

2. Plan Generation

In this section, we describe the form of our rules We must first define what we want to produce with these rules, namely a query evaluation plan, and its constituents

2.1. Plans

The basic object to be manipulated — and the class of "terminals" in our grammar — is a ***LOw-LEvel Plan OPerator (LOLEPOP)*** that will be interpreted by the query evaluator at run-time LOLEPOPs are a variation of the relational algrebra (e g , **JOIN, UNION,** etc), supplemented with low-level operators such as **ACCESS, SORT, SHIP,** etc [FREY 87] Each LOLEPOP is viewed as a function that operates on 1 or 2 tables[2], which are parameters to that function, and produces a single table as output A ***table*** can be either a table stored on disk or a "stream of tuples" in memory or a communication pipe The **ACCESS** LOLEPOP converts a stored table to a stream of tuples, and the **STORE** LOLEPOP does the reverse In addition to input tables, a LOLEPOP may have other parameters that control its operation For example, one parameter of the **SORT** LOLEPOP is the set of columns on which to sort Parameters may also specify a ***flavor*** of LOLEPOP For example, different join methods having the same input parameter structure are represented by different flavors of the **JOIN** LOLEPOP, differences in input parameters would necessitate a distinct LOLEPOP Parameters may be optional, for example, the **ACCESS** LOLEPOP may optionally apply a set of predicates

A ***query evaluation plan*** (***QEP***, or ***plan***) is a directed graph of LOLEPOPs An example plan is shown in Figure 1 Note that arrows point toward the source of the stream, not the direction in which tuples flow This plan shows a sort-merge **JOIN** of DEPT as the outer table and EMP as the inner table The DEPT stream is generated by an **ACCESS** to the stored table DEPT, then **SORT**ed into the order of column DNO for the merge-join The EMP stream is generated by an **ACCESS** to the stored index on column EMP DNO[3] that includes as one "column" the ***tuple identifier (TID)*** For each tuple in the stream, the **GET** LOLEPOP then uses the TID to get additional columns from its stored table columns NAME and ADDRESS from EMP in this example

Another way of representing this plan is as a nesting of functions [BATO 87a, FREY 87]

```
JOIN (sort merge, DEPT DNO=EMP DNO,
      SORT(ACCESS(DEPT, {DNO,MGR}, {MGR='Haas'}), DNO),
      GET( ACCESS(Index on EMP DNO, {TID, DNO}, φ),
           EMP, {NAME, ADDRESS}, φ                  )      )
```

This representation would be a lot more readable, and easier to construct, if we were to define intermediate functions D and E for the last two parameters to JOIN

```
JOIN (sort merge, D DNO=E DNO, D, E )
```

where

---

2 Nothing in the structure of our rules prevents LOLEPOPs from operating on any number of tables

3 Actually, ACCESSes to base tables and to access methods such as this index use different flavors of ACCESS

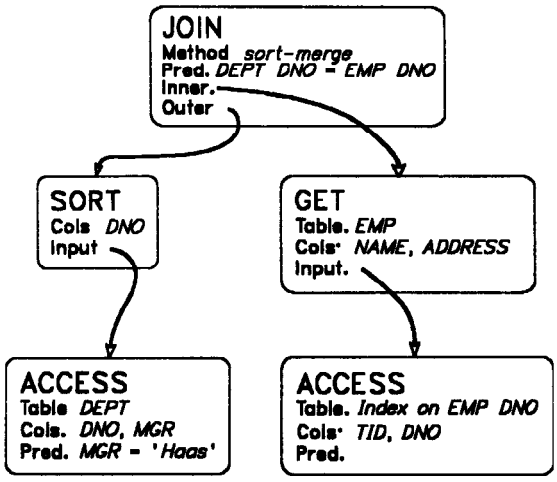


Figure 1 One potential query evaluation plan for the SQL query

```
SELECT NAME, ADDRESS
FROM EMP E, DEPT D
WHERE E DNO = D DNO AND MGR='Haas'
```

$$D = \mathbf{SORT}(\mathbf{ACCESS}(DEPT, \{DNO, MGR\}, \{MGR='Haas'\}), DNO)$$

and

$$E = \mathbf{GET}(\mathbf{ACCESS}(Index\ onEMP\ DNO, \{TID, DNO\}, \phi), EMP, \{NAME, ADDRESS\}, \phi)$$

If properly parametrized, these intermediate functions could be re-used for creating an ordered stream for *any* table, e g

$$\text{OrderedStream1}(T, C, P, order) = \mathbf{SORT}(\mathbf{ACCESS}(T, C, P), order)$$

and

$$\text{OrderedStream2}(T, C, P, order) = \mathbf{GET}(\mathbf{ACCESS}(a, \{TID\}, \phi), T, C, P) \quad \text{IF } order \sqsubseteq a$$

where T is the stored table (base table or base tables represented in a stored intermediate result) to be accessed, C is the set of columns to be accessed, P is the set of predicates to be applied, and "$order \sqsubseteq a$" means "the ordered list of columns of *order* are a prefix of those of access path a of T" Now it becomes apparent that OrderedStream1 and OrderedStream2 provide two alternative definitions for a single concept, an OrderedStream, in which the second definition depends upon the existence of a suitable access path

$$\text{OrderedStream}(T, C, P, order) = \begin{cases} \mathbf{SORT}(\mathbf{ACCESS}(T, C, P), order) \\ \mathbf{GET}(\mathbf{ACCESS}(a, \{TID\}, \phi), T, C, P) & \text{IF } order \sqsubseteq a \end{cases}$$

This higher-level construct can now be nested within other functions needing an ordered stream, without having to worry about the details of how the ordered stream was created [BATO 87a] It is precisely this train of reasoning that inspired the grammar-like design of our rules for constructing plans

2 2 Rules

Executable plans are defined using a grammar-like set of parametrized production rules called ***STrategy Alternative Rules (STARs)*** that define higher-level constructs from lower-level constructs, in a way resembling common mathematical functions or a functional programming language [BACK 78] A STAR defines a named, parametrized object (the "nonterminals" in our grammar) in terms of one or more ***alternative definitions***, each of which

- may have a ***condition of applicability***, and
- defines a plan by referencing one or more LOLEPOPs or other STARs, specifying ***arguments*** for their parameters

Arguments and conditions of applicability may reference constants, parameters of the STAR being defined, or other LOLEPOPs or STARs For example, the intermediate functions OrderedStream1 and OrderedStream2, defined above, are examples of STARs with only one alternative definition, but OrderedStream has two alternative definitions The first of these references the SORT LOLEPOP, whose first argument is a reference to the ACCESS LOLEPOP and whose second argument is the parameter *order* The conditions of applicability for all the alternatives may either overlap or be exclusive If they overlap, as they do for OrderedStream, then the STAR may return more than one plan

In addition, we may wish to apply the function to every element of a set For example, in OrderedStream2 above, any other index on EMP having DNO as its major column could achieve the desired order So we need a STAR to generate an ACCESS plan *for each index i in that set I*

$$\text{IndexAccess}(T) = \forall i \in I \quad \mathbf{ACCESS}(i, \{TID\}, \phi)$$

Using rule IndexAccess in rule OrderedStream2 as the first argument should apply the GET LOLEPOP to each such plan, i e , for each alternative plan returned by IndexAccess, the GET function will be referenced with that plan as its first argument So **GET** (IndexAccess(*EMP*), *C*, *P*) will also return multiple plans Therefore any STAR having overlapping conditions or referencing a multi-valued STAR will itself be multi-valued It is easiest to treat all STARs as operations on the abstract data type ***Set of Alternative Plans for a stream (SAP)***, which consume one or two SAPs and are mapped (in the LISP sense [FREY 87]) onto each element of those SAPs to produce an output SAP Set-valued parameters other than SAPs (such as the sets of columns *C* and predicates *P* above) are treated as a single parameter unless otherwise designated by the ∀ clause, as was done in the definition of IndexAccess

2.3. Use and Implementation

As our functional notation suggests, the rule mechanism starts with the ***root STAR***, which is the "starting state" of our grammar The root STAR has one or more alternative definitions, each of which may reference other STARs, which in turn may reference other STARs, and so on top down until a STAR is defined totally in terms of "terminals", i e LOLEPOPs operating on constants Each reference of a STAR is evaluated by replacing the reference with its alternative definitions that satisfy the condition of applicability, and replacing the parameters of those definitions with the arguments of the reference Unlike transformational rules, this substitution process is remarkably simple and fast, the fanout of any reference of a STAR is limited to just those STARs referenced in its definition, and alternative definitions may be evaluated in parallel Therein lies the real advantage of STARs over transformational rules The implementation of a prototype interpreter for STARs, including a very general mechanism for controlling the order in which STARs are evaluated, is described in [LEE 88]

Thus far in Starburst, we have sets of STARs for accessing individual tables and joins, but STARs may be defined for any new operation, e g outer join, and may reference any other STAR The root STAR for joins is called JoinRoot, a possible definition of which appears in Section "4 Example Join STARs", along with the STARs that it references Simplified definitions of the single-table access STARs are given in [LEE 88] For any given SQL query, we build plans bottom up, first referencing the AccessRoot STAR to build plans to access individual tables, and then repeatedly referencing the JoinRoot STAR to join plans that were generated earlier, until all tables have been joined What constitutes a joinable pair of streams depends upon a compile-time parameter The default is to give preference to those streams having an eligible join predicate linking them, as did System R and R*, but this can be overridden to also consider Cartesian products between two streams of small estimated cardinality In addition, in Starburst we exploit all predicates that reference more than one table as join predicates This generalization of System R's and R*'s "col1 = col2" join predicates, plus allowing plans to have composite inners (e g , (A*B)*(C*D)) and Cartesian products (when the appropriate parameters are specified), significantly complicates the generation of legal join pairs and increases their number However, a cheaper plan is more likely to be discovered among this expanded repertoire! We will address this aspect of query optimization in a forthcoming paper on join enumeration

3. Properties of Plans

The concept of cost has been generalized to include all properties a plan might have We next present how properties are defined and changed, and how they interact with STARs

3.1. Description

Every table (either base table or result of a plan) has a set of ***properties*** that summarize the work done on the table thus far (as in [GRAE 87b], [BATO 87a],and [ROSE 87]) and hence are important to the cost model These properties are of three types

relational the relational content of the plan, e g due to joins, projections, and selections

physical the physical aspects of the tuples, which affect the cost but not the relational content, e g the order of the tuples

estimated properties derived from the previous two as part of the cost model, e g estimated cardinality of the result and cost to produce it

Examples of these properties are summarized in Figure 2 All properties are handled uniformly as elements of a ***property vector***, which can easily be extended to add more properties (see section 5)

Initially, the properties of stored objects such as tables and access methods are determined from the system catalogs For example, for a table, the catalogs contain its constituent columns (COLS), the SITE at which it is stored [LOHM 85], and the access PATHS defined on it No predicates (PREDS) have been applied yet, it is not a TEMPorary table, and no COST has been incurred in the query The ORDER is "unknown" unless the table is known to store tuples in some order, in which case the order is defined by the ordered set of columns on which the tuples are ordered

Each LOLEPOP changes selected properties, including adding cost, in a way determined by the arguments of its reference and the properties of any arguments that are plans For example, **SORT**

- Relational (WHAT)

TABLES	Set of tables accessed
COLS	Set of columns accessed
PREDS	Set of predicates applied

- Physical (HOW)

ORDER	Ordering of tuples (an ordered list of columns)
SITE	Site to which tuples delivered
TEMP	"True" if materialized in a temporary table
PATHS	Set of available access paths on (set of) tables, each element an ordered list of columns

- Estimated (HOW MUCH)

CARD	Estimated number of tuples resulting
COST	Estimated cost (total resources, a linear combination of I/O, CPU, and communications costs [LOHM 85])

Figure 2 Example properties of a plan.

changes the ORDER of tuples to the order specified in a parameter **SHIP** changes the SITE property to the specified site Both LOLEPOPs add to the COST property of their input stream additional cost that depends upon the size of that stream, which is a function of its properties CARD and COLS **ACCESS** changes a stored table to a memory-resident stream of tuples, but optionally can also subset columns (relational project) and apply predicates (relational select) that may be enumerated as arguments The latter option will of course change the CARD property as well These changes, including the appropriate cost and cardinality estimates, are defined in Starburst by a ***property function*** for each LOLEPOP Each property function is passed the arguments of the LOLEPOP, including the property vector for arguments that are STARs or LOLEPOPs, and returns the revised property vector Thus, once STARs are reduced to LOLEPOPs, the cost of any plan can be assessed by invoking the property function for successive LOLEPOPs These cost functions are well established and validated [MACK 86], so will not be discussed further here

3.2. Required vs. Available Properties

A reference of a STAR or LOLEPOP, especially for certain join methods, may require certain properties for its arguments For example, the merge-join requires its input table streams to be ordered by the join columns, and the nested-loop join requires the inner table's access method to apply the join predicate as though it were a single-table predicate ("pushes the selection down") Dyadic LOLEPOPs such as **GET**, **JOIN**, and **UNION** require that the SITE of both input streams be the same

In the previous section, we constructed a STAR for an OrderedStream, where the desired order was a parameter of that STAR Clearly we could require a particular order by referencing OrderedStream with the required order as the corresponding argument The problem is that we may simultaneously require values for any of the 2^n combinations of n properties, and hence would have to have a differently-named STAR for each combination For example, if the sort-merge **JOIN** in the example is to take place

SITE =x then we need to define a SitedOrderedStream that has parameters for SITE and ORDER and references in its definition SHIP LOLEPOPs to send any stream to SITE x, as well as a SitedStream, an OrderedStream, and a STREAM Actually, SitedOrderedStream subsumes the others, since we can pass nulls for the properties not required But in general, every STAR will need this same capability to specify some or all of the properties that might be required by referencing STARs as parameters Much of the definition of each of these STARs would be redundant, because these properties really are orthogonal to what the stream produces In addition, we often want to find the *cheapest* plan that satisfies the required properties, even if there is a plan that naturally produces the required properties For example, even though there is an index EMP DNO by which we can access EMP in the required DNO order, it might be cheaper, if EMP were not ordered by DNO, to access EMP sequentially and sort it into DNO order

We therefore factor out a separate mechanism called ***Glue***, which can be referenced by any STAR and which

1 checks if any plans exist for the required relational properties (TABLES, COLS, and PREDS), referencing the top-most STAR with those parameters if not,
2 adds to any existing plan "Glue" operators as a "veneer" to achieve the required properties (for example, a SORT LOLEPOP can be added to change the tuple ORDER, or a SHIP LOLEPOP to change the SITE), and
3 either returns the cheapest plan satisfying the requirments or (optionally) all plans satisfying the requirements

In fact, Glue can be specified using STARs, and Glue operators can be STARs as well as LOLEPOPs, as described in [LEE 88]

Required properties in the STAR reference are enclosed in square brackets next to the affected SAP argument, to associate the required properties with the stream on which they are imposing requirements Different properties may be required by references in different STARs, the requirements are accumulated until Glue is referenced This will be illustrated in the next section.

An example of this Glue mechanism is shown in Figure 3 In this example, we assume that table DEPT is stored at SITE=N Y, but the STAR requires DEPT to be delivered to SITE=L.A in DNO order None of the available plans meets those requirements The first available plan must be augmented with a SHIP LOLEPOP to change the SITE property from N Y to L A The second plan, a simple ACCESS of DEPT, must be both SORTed and SHIPped The third plan, perhaps created by an earlier reference of Glue that didn't have the ORDER requirement, has already added a SHIP to plan 2 to get it to L A, but still needs a SORT to achieve the ORDER requirement

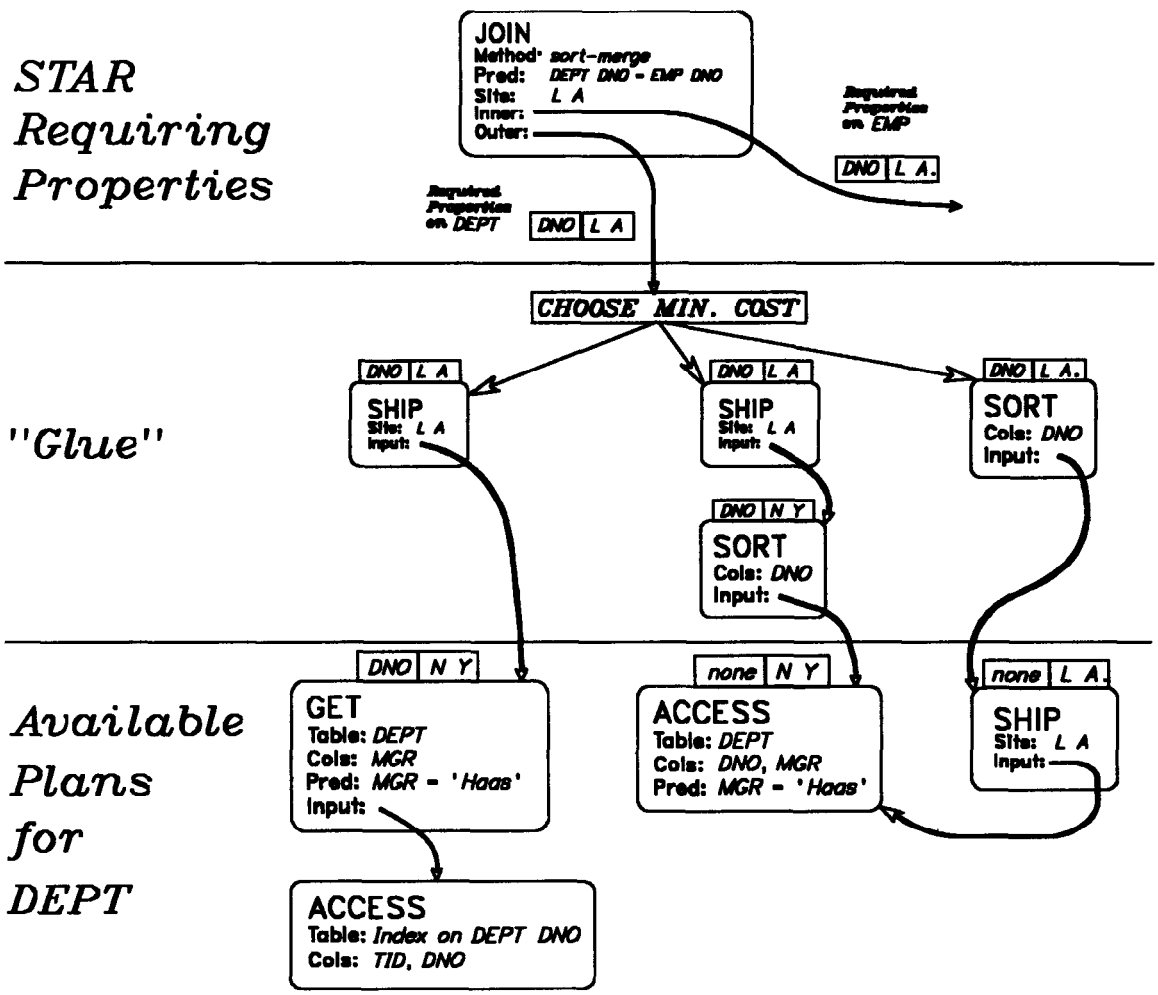


Figure 3 Example of "Glue" mechanism injecting "Glue" operators to match plans to required properties, and choosing the cheapest. Only two properties, order and site, are shown here, as "ears" on top of the top-most LOLEPOP for each plan.

4. Example: Join STARs

To illustrate the power of STARs in this section we discuss one possible set of STARs for generating the join strategies of the R* optimizer (in Sections 4 1 - 4 4), plus several additional strategies such as

- composite inners (Sections 4 1 and 4 3),
- new access methods (Section 4 5 2),
- new join methods (Section 4 4),
- dynamic creation of indexes on intermediate results (Section 4 5 3),
- materialization of inner streams of nested-loop joins to force projection (Section 4 5 2)

Although there may be better ways within our STAR structure to express the same set of strategies, the purpose of this section is to illustrate the full power of STARs Some of the strategies (e g, hash joins) have not yet been implemented in Starburst, they are included merely for illustrating what is involved in adding these strategies to the optimizer

These STARs are by no means complete we have intentionally simplified them by removing parameters and STARs that deal with subqueries treated as joins, for example The reader is cautioned against construing this omission as an inability to handle other cases, on the contrary, it illustrates the flexibility of STARs! We can construct, but have omitted for brevity, additional STARs for

- sorting TIDs taken from an unordered index in order to order I/O accesses to data pages,
- ANDing and ORing of multiple indexes for a single table,
- treating subqueries as joins having different quantifier types (i e, generalizing the predicate calculus quantifiers of ALL and EXISTS to include the FOR EACH quantifier for joins and the UNIQUE quantifier for scalar ("=") subqueries),
- filtration methods such as semi-joins and Bloom-joins

We believe that any desired strategy for non-recursive queries will be expressible using STARs, and are currently investigating what difficulties, if any, arise with recursive queries and multiple execution streams resulting from table partitioning [BATO 87a]

In these definitions, for readability we denote ***exclusive alternative definitions*** by a left *curly brace* and ***inclusive alternative definitions*** by a left *square bracket* In practice, no distinction is necessary In all examples, we will write non-terminals (STAR names) in RegularMixedCase, parameters in *italics* (those which may be sets are denoted by capital letters), and terminals in bold, with LOLEPOPs distinguished by **BOLD CAPITAL LETTERS** Required properties are written in **small bold letters** and surrounded by a pair of [square brackets] For brevity, we have had to shorten names, e g, "JMeth" should read "JoinMethod" The function "$\chi(\bullet)$" denotes "columns of $(\bullet)$", where $\bullet$ can be a set of tables, an index, etc We assume the existence of the basic set functions of $\in, \cap, \subseteq, -$ (set difference), etc

STARs are defined here top down (i e, a STAR referenced by any STAR is defined after its reference), which is also the order in which they will be referenced We start with the root STAR, JoinRoot, which is referenced for a given set of parameters

- table (quantifier) sets *T1* and *T2* (with no order implied)
- the set of (newly) eligible predicates, *P*

Suppose, for example, that plans for joining tables X and Y and for accessing table Z had already been generated, so we were ready to construct plans for joining X*Y with Z Then JoinRoot would be referenced with $T1 = \{X,Y\}$, $T2 = \{Z\}$, and $P = \{X.g = Z.m, Y.h = Z.n\}$

4.1. Join Permutation Alternatives

```
JoinRoot(T1, T2, P) = [PermutedJoin(T1, T2, P)
                       PermutedJoin(T2, T1, P)
```

The meaning of this STAR should be obvious either table-set *T1* or table-set *T2* can be the outer stream, with the other table-set as the inner stream Both are possible alternatives, denoted by an inclusive (square) bracket Note that we have no conditions on either alternative, to exclude a *composite inner* (i e, an inner that is itself the result of a join), we could add a condition restricting the inner table-set to be one table

This simple STAR fails to adequately tax the power of STARs, and thus resembles the comparable rule of transformational approaches However, note that since none of the STARs referenced by JoinRoot or any of its descendants will reference JoinRoot, there is no danger of this STAR being invoked again and "undoing" its effect, as there is in transformational rules [GRAE 87a]

4.2. Join-Site Alternatives

```
PermutedJoin(T1, T2, P) =
        { SitedJoin(T1, T2, P)              IF local query
        { ∀s ∈ σ  RemoteJoin(T1, T2, P, s)  OTHERWISE

RemoteJoin(T1, T2, P, s) =
                SitedJoin(T1[site = s], T2[site = s], P)

where
    σ ≡ set of sites at which tables of the query
          are stored, plus the query site
```

This STAR generates the same join-site alternatives as R* [LOHM 84], and illustrates the specification of a required property Note that Glue is not referenced yet, so the required **site** property accumulates on each alternative until it is The interpretation is

1. If all tables (of the query) are located at the query site, go on to SitedJoin, i e, bypass the RemoteJoin STAR which dictates the join site
2. Otherwise, require that the join take place at one of the sites at which tables are stored or the query originated

If a site with a particularly efficient join engine were available, then that site could easily be added to the definition of σ

4.3 Store Inner Stream?

```
SitedJoin(T1, T2, P) = { JMeth(T1, T2[temp], P)    IF C1
                       { JMeth(T1, T2, P)          OTHERWISE

where
    C1 ≡ IF |T2| > 1 OR T2[site] ≠ T2'[site]
```

Again, this simple STAR has an obvious interpretation, although the condition C1 is a bit complicated

1. IF the inner stream (*T2*) is a composite, or its site is not the same as its required site (T'[***site***]), then dictate that it be stored as a temp and call JMeth
2. OTHERWISE, reference JMeth with no additional requirements

Note that if the second disjunct of condition C1 were absent, there would be no reason that this STAR couldn't be the parent

(referencer) of the previous STAR, instead of vice versa As written, SitedJoin exploits decisions made in its parent STAR, PermutedJoin A transformational rule would either have to test if the site decision were made yet, or else inject the temp requirement redundantly in every transformation that dictated a site

4.4. Alternative Join Methods

```
JMeth(T1, T2, P) ≈
  JOIN(NL, Glue(T1, φ), Glue(T2, JP ∪ IP), JP, P−(JP ∪ IP))
  JOIN( MG, Glue(T1[order = χ(SP) ∩ χ(T1)], φ),
            Glue(T2[order = χ(SP) ∩ χ(T2)], IP),
            SP, P−(IP ∪ SP)                    ) IF SP≠φ
where
P  ≡  all eligible predicates
JP ≡  join predicates (multi-table, no ORs or
      subqueries, etc , but expressions OK)
SP ≡  sortable predicates
   =  { p∈JP of form 'col1 op col2', where
      col1∈χ(T1) & col2∈χ(T2) or vice versa }
IP ≡  predicates eligible on the inner only,
      i e , predicates p such that χ(p) ⊆ χ(T2)
```

This STAR references two alternative join methods, both represented as references of the **JOIN** LOLEPOP with different parameters

1 the join method (flavor of **JOIN**),
2 the outer stream and any required properties on that stream,
3 the inner stream and any required properties on that stream,
4 the join predicate(s) applicable by that join method (needed for the cost equations),
5 any residual predicates to apply *after* the join

The two join methods here are

1 **Nested-Loop (NL) Join,** which can always be done For each outer tuple instance, columns of the join predicates (JP) in the outer are instantiated to convert each JP to a single-table predicate on the inner stream[4] These and any predicates on just the inner (IP) are "pushed down" to be applied by the inner stream, if possible Any multi-table predicates that don't qualify as join predicates must be applied as residual predicates Note that the predicates to be applied by the inner stream are *parameters, not required attributes* This forces Glue to re-reference the single-table STARs to generate plans that *exploit* the converted JP predicates rather than *retrofitting* a **FILTER** LOLEPOP to existing plans that applied only the IP predicates

2 **Merge (MG) Join** If there are sortable predicates (SP), dictate that both inner and outer be sorted on their columns of SP Note that the merge join, unlike the nested-loop join, applies the sortable predicates as part of the join itself, pushing down to the inner stream only the single-table predicates on the inner (IP) The JOIN LOLEPOP in Figure 1, for example, would be generated by this alternative As before, remaining multi-table predicates must be applied by JOIN as residuals after the join

Glue will first reference the STARs for accessing the given table(s), applying the given predicate(s), if no plans exist for those parameters In Starburst, a data structure hashed on the tables and predicates facilitates finding all such plans, if they exist Glue then adds the necessary operators to each of these plans, as described in the previous section Simplified STARs for Glue, which this STAR references, and for accessing stored tables, which Glue references, are given in [LEE 88]

4.5. Additional Join Methods

Suppose now we wanted to augment the above alternatives with additional join methods All of the following alternative definitions would be added to the right-hand side of the above STAR (JMeth)

4 5 1 Hash Join Alternative

The hash join has shown promising performance [BABB 79, BRAT 84, DEWI 85] We assume here a hash-join flavor (HA) that atomically bucketizes both input streams and does the join on the buckets

```
JOIN(HA, Glue(T1, φ), Glue(T2, IP), HP, P−IP)    IF HP≠φ
 where
   HP ≡ hashable predicates
      ≡ { p∈JP of form 'expr(χ(T1)) = expr(χ(T2))'}
```

As in the merge join, only single-table predicates can be pushed down to the inner Note that all multi-table predicates (P-IP) — even the hashable predicates (HP) — remain as residual predicates, since there may be hash collisions Also note that the set of hashable predicates HP contains some predicates not in the set of sortable predicates SP (expressions on any number of columns in the same table), and vice versa (inequalities)

An alternate (and probably preferable) approach would be to add a **bucketized** property to the property vector and a LOLEPOP to achieve that property, so that any join method in the JMeth STAR could perform the join in parallel on each of the bucketized streams, with appropriate adjustments to its cost

4.5 2. Forcing Projection Alternative

To avoid expensive in-memory copying, tuples are normally retained as pages in the buffer just as they were **ACCESS**ed, until they are materialized as a temp or **SHIP**ped to another site Therefore, in nested-loop joins it may be advantageous to materialize (**STORE**) the selected and projected inner and re-**ACCESS** it before joining, whenever a very small percentage of the inner table results (i e , when the predicates on the inner table are quite selective and/or only a few columns are referenced) Batory suggests the same strategy whenever the inner "is generated by a complex expression" [BATO 87a] The following forces that alternative

```
JOIN( NL, Glue(T1, φ),
          TableAccess(Glue(T2[temp], IP), *, JP),
          JP, P−(IP ∪ JP)                        )
```

This JMeth alternative accesses the inner stream (*T2*), applying only the single-table predicates (IP), and forcing Glue to **STORE** the result in a temp (permanently stored tables are not considered temps initially) All columns (*) of the temp are then re-accessed, re-using the STAR for accessing any stored table, TableAccess Note that the STAR structure allows us to specify that the join predicates (JP) can be pushed down only to this access, to prevent the temp from being re-materialized *for each outer tuple!*

4 Ullman has coined the term "sideways information passing" [ULLM 85] for this conversion of join predicates to single-table predicates by instantiating one side of the predicate which was done in System R [SELI 79]

```
TableAccess(T, C, P) =

  { ACCESS(Heap, T, C, P)        IF StMgr(T) = 'heap'
  { ACCESS(BTree, T, C, P)       IF StMgr(T) = 'B-tree'
```

A TableAccess can be one (and only one) of the following flavors of ACCESS, depending upon the type of storage manager (StMgr) used, as described in [LIND 87]

1 A physically-sequential **ACCESS** of the pages of table *T*, if the storage manager type of *T* is 'heap', or
2 A B-Tree type **ACCESS** of table *T*, if the storage manager type of *T* is 'B-tree',

retrieving columns *C* and applying predicates *P* By now it should be apparent how easily alternatives for additional storage manager types could be added to this STAR alone, and affect all STARs that reference TableAccess

4 5 3 Dynamic Indexes Alternative

The nested-loop join works best when an index on the inner table can be used to limit the search of the inner to only those tuples satisfying the join and/or single-table predicates on the inner Such an index may not have been created by the user, or the inner may be an intermediate result, in which case no auxiliary access paths such as an index are normally created However, we can force Glue to create the index as another alternative Although this sounds more expensive than sorting for a merge join, it saves sorting the outer for a merge join, and will pay for itself when the join predicate is selective [MACK 86]

```
JOIN ( NL, Glue(T1, φ),
          Glue(T2[paths ≥ IX], XP∪IP), XP − IP, P−(XP ∪ IP) )
where
   XP ≡ indexable multi-table predicates
      = { p ∈ JP of form 'expr(χ(T1)) op T2.col'}
   IX ≡ columns of indexable predicates
      = (χ(IP) ∪ χ(XP)) ∩ χ(T2), '=' predicates first
```

This alternative forces Glue to make sure that the access paths property of the inner contains an index on the columns that have either single-table (IP) or indexable (XP) predicates, ordered so that those involved in equality predicates are applied first If this index needs to be created, the STARs implementing Glue will add [***order***] and [***temp***] requirements to ensure the creation of a compact index on a stored table As in the nested-loop alternative, the indexable multi-table predicates "pushed down" to the inner are effectively converted to single-table predicates that change for each outer tuple

5. Extensibility — What's Really Involved

Here we discuss briefly the steps required to change various aspects of the optimizer strategies, in order to demonstrate the extensibility and modularity of our STAR mechanism

Easiest to change are the STARs themselves, when an existing set of LOLEPOPs suffices If the STARs are treated as input data to a rule interpreter, then new STARs can be added to that file without impacting the Starburst system code at all [LEE 88] If STARs are compiled to generate an optimizer (as in [GRAE 87a, GRAE 87b]), then updates of the STARs would be followed by a re-generation of the optimizer In either case, any STAR having a condition not yet defined would require defining a C function for that condition, compiling that function, and relinking that part of the optimizer to Starburst Note that we assume that the DBC specifies the STARs correctly, i e without infinite cycles or meaningless sequences of LOLEPOPs An open issue is how to verify that any given set of STARs is correct

Less frequently, we may wish to add a new LOLEPOP, e g **OUTERJOIN** This necessitates defining and compiling two C functions a run-time execution routine that will be invoked by the query evaluator, and a property function for the optimizer to specify the changes to plan properties (including cost) made by that LOLEPOP In addition, STARs must be added and/or modified, as described above, to reference the LOLEPOP under the appropriate circumstances

Probably the least likely and most serious alterations occur when a property is added (or changed in any way) in the property vector Since the default action of any LOLEPOP on any property is to leave the input property unchanged, only those property functions that reference the new property would have to be updated, recompiled, and relinked to Starburst By representing the property vector as a self-defining record having a variable number of fields, each of which is a property, we can insulate unaffected property functions from any changes to the structure of the property vector STARs would be affected only if the new property were required or produced by that STAR

6. Related Work

Some aspects of our STARs resemble features of earlier work, but there are some important differences As we mentioned earlier, our STARs are inspired by functional programming concepts [BACK 78] A major difference is that our "functions" (STARs) can be multi-valued, i e a *set* of alternative objects (plans) The other major inspiration, a production of a grammar, does not permit a condition upon alternative expansions of a non-terminal it either matches or it doesn't (and the alternatives must be exclusive) Hoping to use a standard compiler generator to compile our STARs, we investigated the use of partially context-sensitive W-grammars [CLEA 77] for enforcing the "context" of required properties, but were discouraged by the same combinatorial explosion of productions described above when many properties are possible Koster [KOST 71] has solved this using a technique similar to ours, in which a predicate called an "affix" (comparable to our condition of applicability) may be associated with each alternative definition He has shown affix grammars to be Turing complete In addition, grammars are typically used in a parser to find *just one* expansion to terminals, whereas our goal is to construct *all* such expansions Although a grammar can be used to construct all legal sequences, this set may be infinite [ULLM 85]

The transformational approach of the EXODUS optimizer [GRAE 87a, GRAE 87b] uses C functions for the IF conditions and expresses the alternatives in rules, as do we, but then compiles those rules and conditions using an "optimizer generator" into executable code Given one initial plan, this code generates all legal variations of that plan using two kinds of rules transformation rules to define alternative transformations of a plan, and implementation rules to define alternative methods for implementing an operator (e g , nested-loop and sort-merge algorithms for implementing the JOIN operator) Our approach does not require an initial plan, and has only one type of rule, which permits us to express interactions between transformations and methods Our property functions are indistinguishable from Graefe's property and

si functions, although we have identified more properties than any other author to date Graefe does not deal with the need of some rules (e g merge join) to require certain properties, as discussed in Section 3 2 and illustrated in Sections 4 2 - 4 4, 4 5 2, and 4 5 3 Although Graefe re-uses common subplans in alternative plans, transformational rules may subsequently generate alternatives and pick a new optimal plan for the subplan, forcing re-estimation of the cost of every plan that has already incorporated that subplan Our building blocks approach avoids this problem by generating all plans for the subplan before incorporating that subplan in other plans, although Glue may generate some new plans having different properties and/or parameters And while the structure of our STARs does not preclude compilation by an optimizer generator, it also permits interpreting the STARs by a simple yet efficient interpreter during optimization, as was done in our prototype Interpretation saves re-compiling the optimizer component every time a strategy is added or changed, and also allows greater control of the order of evaluation For example, depending upon the value of a STAR's parameter, we may never have to construct entire subtrees within the decision tree, but a compiled optimizer must contain a completely general decision tree for all queries

Freytag [FREY 87] proposes a more LISP-like set of transformational rules that starts from a non-procedural set of parameters from the query, as do we, and transforms them into all alternative plans He points to the EXODUS optimizer generator as a possible implementation, but does not address several key implementation issues such as his ellipsis (" ") operator, which denotes any number of expressions, e g

$$((\text{JOIN } T_1(\ T_2\ )) \rightarrow (\text{JOIN } T_1(\quad)T_2))$$

And the ORDER and SITE properties (only) are expressed as functions, which presumably would have to be re-derived each time they were referenced in the conditions Freytag does not exploit the structure of query optimization to limit what rules are applicable at any time and to prevent re-application of the same rules to common subplans shared by two alternative plans, although he suggests the need to do so

Rosenthal and Helman [ROSE 87] suggest specifications for "well-formed" plans, so that transformational rules can be verified as valid if they transform well-formed plans to well-formed plans Like Graefe, they associate properties with plans, viewed as predicates that are true about the plan Alternative plans producing the same intermediate result with the same properties converge on "data nodes", on which "transformations that insert unary operators are more naturally applied" An operator is then well-formed if any input plan satisfying the required input properties produces an output plan that satisfies the output properties The paper emphasizes representations for verifiability and search issues, rather than detailing mechanisms (1) to construct well-formed transformations, (2) to match input data nodes to output data nodes (corresponding to our Glue), and (3) to recalculate the cost of all plans that share (through a common data node) a common subplan that is altered by a transformation

Probably the closest work to ours is Batory's "synthetic" architecture for the entire GENESIS extensible database system (not just the query optimizer [BATO 87b]), in which "atoms" of "primitive algorithms" are composed by functions into "molecules", in layers that successively add implementation details [BATO 87a] Developed concurrently and independently, Batory's functional notation closely resembles STARs, but is presented and implemented as rewrite (transformational) rules that are used to construct and compile the complete set of alternatives *a priori* for a given optimizer, after first selecting from a catalog of available algorithms those desired to implement operators for each layer At the highest layer, for example, the DBC chooses from many optimization algorithms (e g depth-first vs breadth-first), while the choices at the lowest layers correspond to our flavors of LOLEPOPs or Graefe's methods The functions that compose these operations do not explicitly permit conditions on the alternative definitions, as do we Batory considers them unnecessary when rules are constructed properly but alludes to them in comments next to some alternatives and in a footnote Inclusive alternatives automatically become arguments of a CHOOSE_CHEAPEST function during the composition process The rewrite rules include rules to match properties (which he calls characteristics) even if they are unneeded e g a SORT may be applied to a stream that is already ordered appropriately by an index, as well as rules to simplify the resulting compositions and eliminate any such unnecessary operations By treating the stored vs in-memory distinction as a property of streams, and by having a general-purpose Glue mechanism, we manage to factor out most of these redundancies in our STARs Although clearly relevant to query optimization, Batory s larger goal was to incorporate an encyclopedic array of known query processing algorithms within his framework, including operators for splitting, processing in parallel, and assembling horizontal partitions of tables

7. Conclusions

We have presented a grammar for specifying the set of legal strategies that can be executed by the query evaluator The grammar composes low-level database operators (LOLEPOPs) into higher-level constructs using rules (STARs) that resemble the definition of functions they may have alternative definitions that have IF conditions, and these alternative definitions may, in turn, reference other functions that have already been defined The functions are parametrized objects that produce one or more alternative plans Each plan has a vector of properties, including the cost to produce that plan, which may be altered only by LOLEPOPs When an alternative definition requires certain properties of an input, "Glue" can be referenced to do "impedance matching" between the plans created thus far and the required properties by injecting a veneer of Glue operators

We have shown the power of STARs by specifying some of the strategies considered by the R* system and several additional ones, and believe that any desired extension can be represented using STARs We find our constructive, "building-blocks" grammar to be a more natural paradigm for specifying the "language" of legal sequences of database operators than plan transformational rules, because they allow the DBC to build higher levels of abstraction from lower-level constructs, without having to be aware of how those lower-level constructs are defined And unlike plan transformational rules, which consider all rules applicable at every iteration and which must do complicated unification to determine applicability, referencing a STAR triggers in an obvious way only those STARs referenced in its definition, just like a macro expander This limited fanout of STARs should make it possible to achieve our goal of expressing alternative optimizer strategies as data and still use these rules to generate and evaluate the cost of a large number of plans within a reasonable amount of time

8. Acknowledgements

We wish to acknowledge the contributions to this work by several colleagues, especially the Starburst project team We particularly benefitted from lengthy discussions with — and suggestions by — Johann Christoph Freytag (now at the European Community Research Center in Munich), Laura Haas, and Kiyoshi Ono (visiting from the IBM Tokyo Research Laboratory) Laura Haas, Bruce Lindsay, Tim Malkemus (IBM Entry Systems Division in Austin, TX), John McPherson, Kiyoshi Ono, Hamid Pirahesh, Irv Traiger, and Paul Wilms constructively critiqued an earlier draft of this paper, improving its readability significantly We also thank the referees for their helpful suggestions

Bibliography

[BABB 79] E Babb, Implementing a Relational Database by Means of Specialized Hardware, *ACM Trans on Database Systems* **4,1** (1979) pp 1-29

[BATO 86] D S Batory et al, GENESIS An Extensible Database Management System, *Tech Report TR-86-07* (Dept of Comp Sci, Univ of Texas at To appear in IEEE Trans on Software Engineering

[BATO 87a] D S Batory, A Molecular Database Systems Technology, *Tech Report TR-87-23* (Dept of Comp Sci, Univ of Texas at

[BATO 87b] D Batory, Extensible Cost Models and Query Optimization in GENESIS, *IEEE Database Engineering* **10,4** (Nov 1987)

[BACK 78] J Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", *Comm ACM* **21,8** (Aug 1978)

[BERN 81] P Bernstein and D -H Chiu, Using Semi-Joins to Solve Relational Queries, *Journal ACM* **28,1** (Jan 1981) pp 25-40

[BRAT 84] K Bratbergsengen, Hashing Methods and Relational Algebra Operations, *Procs of the Tenth International Conf on Very Large Data Bases (Singapore),* **Morgan Kaufmann Publishers** (Los Altos, CA, 1984) pp 323-333

[CARE 86] M J Carey, D J DeWitt, D Frank, G Graefe, J E Richardson, E J Shekita, and M Muralikrishna, The Architecture of the EXODUS Extensible DBMS a Preliminary Report, *Procs of the International Workshop on Object-Oriented Database Systems* (Asilomar, CA, Sept 1986)

[CHU 82] W W Chu and P Hurley, Optimal Query Processing for Distributed Database Systems, *IEEE Trans on Computers* **C-31,9** (Sept 1982) pp 835-850

[CLEA 77] J C Cleaveland and R C Uzgalis, *Grammars for Programming Languages,* **Elsevier North-Holland** (New York, 1977)

[DANI 82] D Daniels, P G Selinger, L M Haas, B G Lindsay, C Mohan, A Walker, and P Wilms, An Introduction to Distributed Query Compilation in R*, *Procs Second International Conf on Distributed Databases* (Berlin, September 1982) Also available as IBM Research Report RJ3497, San Jose, CA, June 1982

[DEWI 85] D J DeWitt and R Gerber, Multiprocessor Hash-Based Join Algorithms, *Procs of the Eleventh International Conf on Very Large Data Bases (Stockholm, Sweden),* **Morgan Kaufmann Publishers** (Los Altos, CA, September 1985) pp 151-164

[EPST 78] R Epstein, M Stonebraker, and E Wong, Distributed Query Processing in a Relational Data Base System, *Procs of ACM-SIGMOD* (Austin, TX, May 1978) pp 169-180

[FREY 87] J C Freytag, A Rule-Based View of Query Optimization, *Procs of ACM-SIGMOD* (San Francisco, CA, May 1987) pp 173-180

[GRAE 87a] G Graefe and D J DeWitt, The EXODUS Optimizer Generator, *Procs of ACM-SIGMOD* (San Francisco, CA, May 1987) pp 160-172

[GRAE 87b] G Graefe, Software Modularization with the EXODUS Optimizer Generator, *IEEE Database Engineering* **10,4** (Nov 1987)

[HAER 78] T Haerder, Implementing a Generalized Access Path Structure for a Relational Database System, *ACM Trans on Database Systems* **3,3** (Sept 1978) pp 258-298

[KOST 71] C H A Koster, Affix Grammars, *ALGOL 68 Implementation* **Elsevier North-Holland** (J E L Peck (ed), Amsterdam, 1971) pp 95-109

[LEE 88] M K Lee, J C Freytag, and G M Lohman, Implementing an Interpreter for Functional Rules in a Query Optimizer, *IBM Research Report RJ6125* **IBM Almaden Research Center** (San Jose, CA, March 1988)

[LIND 87] B Lindsay, J McPherson, and H Pirahesh, A Data Management Extension Architecture, *Procs of ACM-SIGMOD* (San Francisco, CA, May 1987) pp 220-226 Also available as IBM Res Report RJ5436, San Jose, CA, Dec 1986

[LOHM 83] G M Lohman, J C Stoltzfus, A N Benson, M D Martin, and A F Cardenas, Remotely-Sensed Geophysical Databases Experience and Implications for Generalized DBMS, *Procs of ACM-SIGMOD* (San Jose, CA, May 1983) pp 146-160

[LOHM 84] G M Lohman, D Daniels, L M Haas, R Kistler, P G Selinger, Optimization of Nested Queries in a Distributed Relational Database, *Procs of the Tenth International Conf on Very Large Data Bases (Singapore),* **Morgan Kaufmann Publishers** (Los Altos, CA, 1984) pp 403-415 Also available as IBM Research Report RJ4260, San Jose, CA, April 1984

[LOHM 85] G M Lohman, C Mohan, L M Haas, B G Lindsay, P G Selinger, P F Wilms, and D Daniels, Query Processing in R*, *Query Processing in Database Systems, Springer-Verlag* (Kim, Batory, & Reiner (eds), 1985) pp 31-47 Also available as IBM Research Report RJ4272, San Jose, CA, April 1984

[MACK 86] L F Mackert and G M Lohman, R* Optimizer Validation and Performance Evaluation for Distributed Queries, *Procs of the Twelfth International Conference on Very Large Data Bases (Kyoto)* **Morgan Kaufmann Publishers** (Los Altos, CA, August 1986) pp 149-159 Also available as IBM Research Report RJ5050, San Jose, CA, April 1986

[MORR 86] K Morris, J D Ullman, and A Van Gelder, Design Overview of the NAIL! System, *Report No STAN-CS-86-1108* **Stanford University** (Stanford, CA, May 1986)

[ROSE 87] A Rosenthal and P Helman, Understanding and Extending Transformation-Based Optimizers, *IEEE Database Engineering* **10,4** (Nov 1987)

[SCHW 86] P M Schwarz, W Chang, J C Freytag, G M Lohman, J McPherson, C Mohan, and H Pirahesh, Extensibility in the Starburst Database System, *Procs of the International Workshop on Object-Oriented Database Systems (Asilomar, CA),* **IEEE** (Sept 1986)

[SELI 79] P G Selinger, M M Astrahan, D D Chamberlin, R A Lorie, and T G Price, Access Path Selection in a Relational Database Management System *Procs of ACM-SIGMOD* (May 1979) pp 23-34

[STON 86] M Stonebraker and L Rowe, The Design of Postgres *Procs of ACM-SIGMOD* (May 1986) pp 340-355

[ULLM 85] J D Ullman, Implementation of Logical Query Languages for Databases, *ACM Trans on Database Systems* **10,3** (September 1985) pp 289-321

[VALD 87] P Valduriez, Join Indices, *ACM Trans on Database Systems* **12,2** (June 1987) pp 219-246

[WONG 76] E Wong and K Youssefi, Decomposition — a Strategy for Query Processing, *ACM Trans on Database Systems* **1,3** (Sept 1976) pp 223-241

[WONG 83] E Wong and R Katz, Distributing a Database for Parallelism, *Procs of ACM-SIGMOD* (San Jose CA May 1983) pp 23-29

Eddies: Continuously Adaptive Query Processing

Ron Avnur Joseph M. Hellerstein
University of California, Berkeley
avnur@cohera.com, jmh@cs.berkeley.edu

Abstract

In large federated and shared-nothing databases, resources can exhibit widely fluctuating characteristics. Assumptions made at the time a query is submitted will rarely hold throughout the duration of query processing. As a result, traditional static query optimization and execution techniques are ineffective in these environments.

In this paper we introduce a query processing mechanism called an *eddy*, which continuously reorders operators in a query plan as it runs. We characterize the *moments of symmetry* during which pipelined joins can be easily reordered, and the *synchronization barriers* that require inputs from different sources to be coordinated. By combining eddies with appropriate join algorithms, we merge the optimization and execution phases of query processing, allowing each tuple to have a flexible ordering of the query operators. This flexibility is controlled by a combination of fluid dynamics and a simple learning algorithm. Our initial implementation demonstrates promising results, with eddies performing nearly as well as a static optimizer/executor in static scenarios, and providing dramatic improvements in dynamic execution environments.

1 Introduction

There is increasing interest in query engines that run at unprecedented scale, both for widely-distributed information resources, and for massively parallel database systems. We are building a system called Telegraph, which is intended to run queries over all the data available on line. A key requirement of a large-scale system like Telegraph is that it function robustly in an unpredictable and constantly fluctuating environment. This unpredictability is endemic in large-scale systems, because of increased complexity in a number of dimensions:

Hardware and Workload Complexity: In wide-area environments, variabilities are commonly observable in the bursty performance of servers and networks [UFA98]. These systems often serve large communities of users whose aggregate behavior can be hard to predict, and the hardware mix in the wide area is quite heterogeneous. Large clusters of computers can exhibit similar performance variations, due to a mix of user requests and heterogeneous hardware evolution. Even in totally homogeneous environments, hardware performance can be unpredictable: for example, the outer tracks of a disk can exhibit almost twice the bandwidth of inner tracks [Met97].

Data Complexity: Selectivity estimation for static alphanumeric data sets is fairly well understood, and there has been initial work on estimating statistical properties of static sets of data with complex types [Aok99] and methods [BO99]. But federated data often comes without any statistical summaries, and complex non-alphanumeric data types are now widely in use both in object-relational databases and on the web. In these scenarios – and even in traditional static relational databases – selectivity estimates are often quite inaccurate.

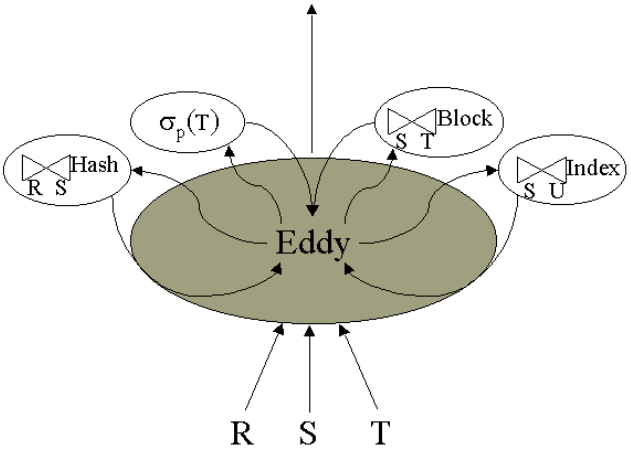


Figure 1: An eddy in a pipeline. Data flows into the eddy from input relations R, S and T. The eddy routes tuples to operators; the operators run as independent threads, returning tuples to the eddy. The eddy sends a tuple to the output only when it has been handled by all the operators. The eddy adaptively chooses an order to route each tuple through the operators.

User Interface Complexity: In large-scale systems, many queries can run for a very long time. As a result, there is interest in Online Aggregation and other techniques that allow users to "Control" properties of queries while they execute, based on refining approximate results [HAC+99].

For all of these reasons, we expect query processing parameters to change significantly over time in Telegraph, typically many times during a single query. As a result, it is not appropriate to use the traditional architecture of optimizing a query and then executing a static query plan: this approach does not adapt to intra-query fluctuations. Instead, for these environments we want query execution plans to be reoptimized regularly during the course of query processing, allowing the system to adapt dynamically to fluctuations in computing resources, data characteristics, and user preferences.

In this paper we present a query processing operator called an *eddy*, which continuously reorders the application of pipe-

lined operators in a query plan, on a tuple-by-tuple basis. An eddy is an n-ary tuple router interposed between n data sources and a set of query processing operators; the eddy encapsulates the ordering of the operators by routing tuples through them dynamically (Figure 1). Because the eddy observes tuples entering and exiting the pipelined operators, it can adaptively change its routing to effect different operator orderings. In this paper we present initial experimental results demonstrating the viability of eddies: they can indeed reorder effectively in the face of changing selectivities and costs, and provide benefits in the case of delayed data sources as well.

Reoptimizing a query execution pipeline on the fly requires significant care in maintaining query execution state. We highlight query processing stages called *moments of symmetry*, during which operators can be easily reordered. We also describe *synchronization barriers* in certain join algorithms that can restrict performance to the rate of the slower input. Join algorithms with frequent moments of symmetry and adaptive or non-existent barriers are thus especially attractive in the Telegraph environment. We observe that the Ripple Join family [HH99] provides efficiency, frequent moments of symmetry, and adaptive or nonexistent barriers for equijoins and non-equijoins alike.

The eddy architecture is quite simple, obviating the need for traditional cost and selectivity estimation, and simplifying the logic of plan enumeration. Eddies represent our first step in a larger attempt to do away with traditional optimizers entirely, in the hope of providing both run-time adaptivity and a reduction in code complexity. In this paper we focus on continuous operator reordering in a single-site query processor; we leave other optimization issues to our discussion of future work.

1.1 Run-Time Fluctuations

Three properties can vary during query processing: the costs of operators, their selectivities, and the rates at which tuples arrive from the inputs. The first and third issues commonly occur in wide area environments, as discussed in the literature [AFTU96, UFA98, IFF+99]. These issues may become more common in cluster (shared-nothing) systems as they "scale out" to thousands of nodes or more [Bar99].

Run-time variations in selectivity have not been widely discussed before, but occur quite naturally. They commonly arise due to correlations between predicates and the order of tuple delivery. For example, consider an employee table clustered by ascending age, and a selection `salary > 100000`; age and salary are often strongly correlated. Initially the selection will filter out most tuples delivered, but that selectivity rate will change as ever-older employees are scanned. Selectivity over time can also depend on performance fluctuations: e.g., in a parallel DBMS clustered relations are often horizontally partitioned across disks, and the rate of production from various partitions may change over time depending on performance characteristics and utilization of the different disks. Finally, Online Aggregation systems explicitly allow users to control the order in which tuples are delivered based on data preferences [RRH99], resulting in similar effects.

1.2 Architectural Assumptions

Telegraph is intended to efficiently and flexibly provide both distributed query processing across sites in the wide area, and parallel query processing in a large shared-nothing cluster. In this paper we narrow our focus somewhat to concentrate on the initial, already difficult problem of run-time operator reordering in a single-site query executor; that is, changing the effective order or "shape" of a pipelined query plan tree in the face of changes in performance.

In our discussion we will assume that some initial query plan tree will be constructed during parsing by a naive *pre-optimizer*. This optimizer need not exercise much judgement since we will be reordering the plan tree on the fly. However by constructing a query plan it must choose a spanning tree of the query graph (i.e. a set of table-pairs to join) [KBZ86], and algorithms for each of the joins. We will return to the choice of join algorithms in Section 2, and defer to Section 6 the discussion of changing the spanning tree and join algorithms during processing.

We study a standard single-node object-relational query processing system, with the added capability of opening scans and indexes from external data sets. This is becoming a very common base architecture, available in many of the commercial object-relational systems (e.g., IBM DB2 UDB [RPK+99], Informix Dynamic Server UDO [SBH98]) and in federated database systems (e.g., Cohera [HSC99]). We will refer to these non-resident tables as *external tables*. We make no assumptions limiting the scale of external sources, which may be arbitrarily large. External tables present many of the dynamic challenges described above: they can reside over a wide-area network, face bursty utilization, and offer very minimal information on costs and statistical properties.

1.3 Overview

Before introducing eddies, in Section 2 we discuss the properties of query processing algorithms that allow (or disallow) them to be frequently reordered. We then present the eddy architecture, and describe how it allows for extreme flexibility in operator ordering (Section 3). Section 4 discusses policies for controlling tuple flow in an eddy. A variety of experiments in Section 4 illustrate the robustness of eddies in both static and dynamic environments, and raise some questions for future work. We survey related work in Section 5, and in Section 6 lay out a research program to carry this work forward.

2 Reorderability of Plans

A basic challenge of run-time reoptimization is to reorder pipelined query processing operators while they are in flight. To change a query plan on the fly, a great deal of state in the various operators has to be considered, and arbitrary changes can require significant processing and code complexity to guarantee correct results. For example, the state maintained by an operator like hybrid hash join [DKO+84] can grow as large as the size of an input relation, and require modification or recomputation if the plan is reordered while the state is being constructed.

By constraining the scenarios in which we reorder operators, we can keep this work to a minimum. Before describing eddies, we study the state management of various join algorithms; this discussion motivates the eddy design, and forms the basis of our approach for reoptimizing cheaply and continuously. As a philosophy, *we favor adaptivity over best-case performance*. In a highly variable environment, the best-case scenario rarely exists for a significant length of time. So we

will sacrifice marginal improvements in idealized query processing algorithms when they prevent frequent, efficient reoptimization.

2.1 Synchronization Barriers

Binary operators like joins often capture significant state. A particular form of state used in such operators relates to the interleaving of requests for tuples from different inputs.

As an example, consider the case of a merge join on two sorted, duplicate-free inputs. During processing, the next tuple is always consumed from the relation whose last tuple had the lower value. This significantly constrains the order in which tuples can be consumed: as an extreme example, consider the case of a slowly-delivered external relation slowlow with many low values in its join column, and a high-bandwidth but large local relation fasthi with only high values in its join column – the processing of fasthi is postponed for a long time while consuming many tuples from slowlow. Using terminology from parallel programming, we describe this phenomenon as a *synchronization barrier*: one table-scan waits until the other table-scan produces a value larger than any seen before.

In general, barriers limit concurrency – and hence performance – when two tasks take different amounts of time to complete (i.e., to "arrive" at the barrier). Recall that concurrency arises even in single-site query engines, which can simultaneously carry out network I/O, disk I/O, and computation. Thus it is desirable to minimize the overhead of synchronization barriers in a dynamic (or even static but heterogeneous) performance environment. Two issues affect the overhead of barriers in a plan: the frequency of barriers, and the gap between arrival times of the two inputs at the barrier. We will see in upcoming discussion that barriers can often be avoided or tuned by using appropriate join algorithms.

2.2 Moments of Symmetry

Note that the synchronization barrier in merge join is stated in an order-independent manner: it does not distinguish between the inputs based on any property other than the data they deliver. Thus merge join is often described as a symmetric operator, since its two inputs are treated uniformly[1]. This is not the case for many other join algorithms. Consider the traditional nested-loops join, for example. The "outer" relation in a nested-loops join is synchronized with the "inner" relation, but not vice versa: after each tuple (or block of tuples) is consumed from the outer relation, a barrier is set until a full scan of the inner is completed. For asymmetric operators like nested-loops join, performance benefits can often be obtained by reordering the inputs.

When a join algorithm reaches a barrier, it has declared the end of a scheduling dependency between its two input relations. In such cases, the order of the inputs to the join can often be changed without modifying any state in the join; when this is true, we refer to the barrier as a *moment of symmetry*. Let us return to the example of a nested-loops join, with outer relation R and inner relation S. At a barrier, the join has completed a full inner loop, having joined each tuple in a subset of R with every tuple in S. Reordering the inputs at this point can be done without affecting the join algorithm, as long as

[1] If there are duplicates in a merge join, the duplicates are handled by an asymmetric but usually small nested loop. For purposes of exposition, we can ignore this detail here.

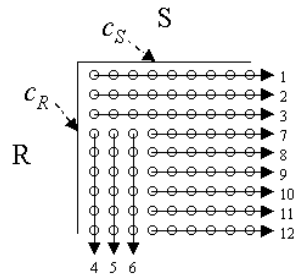


Figure 2: Tuples generated by a nested-loops join, reordered at two moments of symmetry. Each axis represents the tuples of the corresponding relation, in the order they are delivered by an access method. The dots represent tuples generated by the join, some of which may be eliminated by the join predicate. The numbers correspond to the barriers reached, in order. c_R and c_S are the cursor positions maintained by the corresponding inputs at the time of the reorderings.

the iterator producing R notes its current cursor position c_R. In that case, the new "outer" loop on S begins rescanning by fetching the first tuple of S, and R is scanned from c_R to the end. This can be repeated indefinitely, joining S tuples with all tuples in R from position c_R to the end. Alternatively, at the end of some loop over R (i.e. at a moment of symmetry), the order of inputs can be swapped again by remembering the current position of S, and repeatedly joining the next tuple in R (starting at c_R) with tuples from S between c_S and the end. Figure 2 depicts this scenario, with two changes of ordering. Some operators like the pipelined hash join of [WA91] have no barriers whatsoever. These operators are in constant symmetry, since the processing of the two inputs is totally decoupled.

Moments of symmetry allow reordering of the inputs to a single binary operator. But we can generalize this, by noting that since joins commute, a tree of $n-1$ binary joins can be viewed as a single n-ary join. One could easily implement a doubly-nested-loops join operator over relations R, S and T, and it would have moments of complete symmetry at the end of each loop of S. At that point, all three inputs could be reordered (say to T then R then S) with a straightforward extension to the discussion above: a cursor would be recorded for each input, and each loop would go from the recorded cursor position to the end of the input.

The same effect can be obtained in a binary implementation with two operators, by swapping the positions of binary operators: effectively the plan tree transformation would go in steps, from $(R \bowtie_1 S) \bowtie_2 T$ to $(R \bowtie_2 T) \bowtie_1 S$ and then to $(T \bowtie_2 R) \bowtie_1 S$. This approach treats an operator and its right-hand input as a unit (e.g., the unit $[\bowtie_2 T]$), and swaps units; the idea has been used previously in static query optimization schemes [IK84, KBZ86, Hel98]. Viewing the situation in this manner, we can naturally consider reordering multiple joins and their inputs, even if the join algorithms are different. In our query $(R \bowtie_1 S) \bowtie_2 T$, we need $[\bowtie_1 S]$ and $[\bowtie_2 T]$ to be mutually commutative, but do not require them to be the same join algorithm. We discuss the commutativity of join algorithms further in Section 2.2.2.

Note that the combination of commutativity and moments of symmetry allows for very aggressive reordering of a plan

tree. A single n-ary operator representing a reorderable plan tree is therefore an attractive abstraction, since it encapsulates any ordering that may be subject to change. We will exploit this abstraction directly, by interposing an n-ary tuple router (an "eddy") between the input tables and the join operators.

2.2.1 Joins and Indexes

Nested-loops joins can take advantage of indexes on the inner relation, resulting in a fairly efficient pipelining join algorithm. An index nested-loops join (henceforth an "index join") is inherently asymmetric, since one input relation has been pre-indexed. Even when indexes exist on both inputs, changing the choice of inner and outer relation "on the fly" is problematic[2]. Hence for the purposes of reordering, it is simpler to think of an index join as a kind of unary selection operator on the unindexed input (as in the join of S and U in Figure 1). The only distinction between an index join and a selection is that – with respect to the unindexed relation – the selectivity of the join node may be greater than 1. Although one cannot swap the inputs to a single index join, one can reorder an index join and its indexed relation as a unit among other operators in a plan tree. Note that the logic for indexes can be applied to external tables that require bindings to be passed; such tables may be gateways to, e.g., web pages with forms, GIS index systems, LDAP servers and so on [HKWY97, FMLS99].

2.2.2 Physical Properties, Predicates, Commutativity

Clearly, a pre-optimizer's choice of an index join algorithm constrains the possible join orderings. In the n-ary join view, an ordering constraint must be imposed so that the unindexed join input is ordered before (but not necessarily directly before) the indexed input. This constraint arises because of a *physical property* of an input relation: indexes can be probed but not scanned, and hence cannot appear before their corresponding probing tables. Similar but more complex constraints can arise in preserving the ordered inputs to a merge join (i.e., preserving "interesting orders").

The applicability of certain join algorithms raises additional constraints. Many join algorithms work only for equijoins, and will not work on other joins like Cartesian products. Such algorithms constrain reorderings on the plan tree as well, since they always require all relations mentioned in their equijoin predicates to be handled before them. In this paper, we consider ordering constraints to be an inviolable aspect of a plan tree, and we ensure that they always hold. In Section 6 we sketch initial ideas on relaxing this requirement, by considering multiple join algorithms and query graph spanning trees.

2.2.3 Join Algorithms and Reordering

In order for an eddy to be most effective, we favor join algorithms with frequent moments of symmetry, adaptive or non-existent barriers, and minimal ordering constraints: these algorithms offer the most opportunities for reoptimization. In [AH99] we summarize the salient properties of a variety of join algorithms. Our desire to avoid blocking rules out the use of hybrid hash join, and our desire to minimize ordering constraints and barriers excludes merge joins. Nested loops joins have infrequent moments of symmetry and imbalanced barriers, making them undesirable as well.

The other algorithms we consider are based on frequently-symmetric versions of traditional iteration, hashing and indexing schemes, i.e., the Ripple Joins [HH99]. Note that the original pipelined hash join of [WA91] is a constrained version of the hash ripple join. The external hashing extensions of [UF99, IFF+99] are directly applicable to the hash ripple join, and [HH99] treats index joins as a special case as well. For non-equijoins, the block ripple join algorithm is effective, having frequent moments of symmetry, particularly at the beginning of processing [HH99]. Figure 3 illustrates block, index and hash ripple joins; the reader is referred to [HH99, IFF+99, UF99] for detailed discussions of these algorithms and their variants. These algorithms are adaptive without sacrificing much performance: [UF99] and [IFF+99] demonstrate scalable versions of hash ripple join that perform competitively with hybrid hash join in the static case; [HH99] shows that while block ripple join can be less efficient than nested-loops join, it arrives at moments of symmetry much more frequently than nested-loops joins, especially in early stages of processing. In [AH99] we discuss the memory overheads of these adaptive algorithms, which can be larger than standard join algorithms.

Ripple joins have moments of symmetry at each "corner" of a rectangular ripple in Figure 3, i.e., whenever a prefix of the input stream R has been joined with all tuples in a prefix of input stream S and vice versa. For hash ripple joins and index joins, this scenario occurs between each consecutive tuple consumed from a scanned input. Thus ripple joins offer very frequent moments of symmetry.

Ripple joins are attractive with respect to barriers as well. Ripple joins were designed to allow changing rates for each input; this was originally used to *proactively* expend more processing on the input relation with more statistical influence on intermediate results. However, the same mechanism allows *reactive* adaptivity in the wide-area scenario: a barrier is reached at each corner, and the next corner can adaptively reflect the relative rates of the two inputs. For the block ripple join, the next corner is chosen upon reaching the previous corner; this can be done adaptively to reflect the relative rates of the two inputs over time.

The ripple join family offers attractive adaptivity features at a modest overhead in performance and memory footprint. Hence they fit well with our philosophy of sacrificing marginal speed for adaptability, and we focus on these algorithms in Telegraph.

3 Rivers and Eddies

The above discussion allows us to consider easily reordering query plans at moments of symmetry. In this section we proceed to describe the eddy mechanism for implementing reordering in a natural manner during query processing. The techniques we describe can be used with any operators, but algorithms with frequent moments of symmetry allow for more frequent reoptimization. Before discussing eddies, we first introduce our basic query processing environment.

3.1 River

We implemented eddies in the context of River [AAT+99], a shared-nothing parallel query processing framework that dy-

[2] In unclustered indexes, the index ordering is not the same as the scan ordering. Thus after a reordering of the inputs it is difficult to ensure that – using the terminology of Section 2.2 – lookups on the index of the new "inner" relation R produce only tuples between c_R and the end of R.

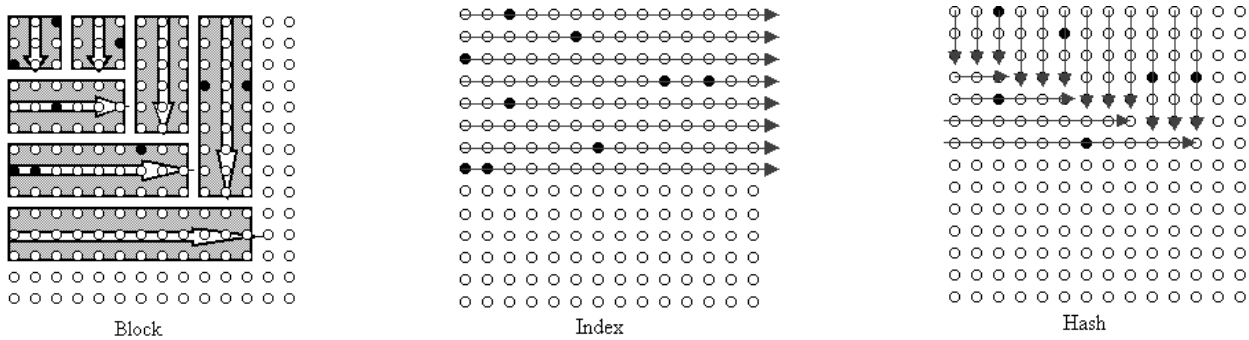


Figure 3: Tuples generated by block, index, and hash ripple join. In block ripple, all tuples are generated by the join, but some may be eliminated by the join predicate. The arrows for index and hash ripple join represent the *logical* portion of the cross-product space checked so far; these joins only expend work on tuples satisfying the join predicate (black dots). In the hash ripple diagram, one relation arrives 3× faster than the other.

namically adapts to fluctuations in performance and workload. River has been used to robustly produce near-record performance on I/O-intensive benchmarks like parallel sorting and hash joins, despite heterogeneities and dynamic variability in hardware and workloads across machines in a cluster. For more details on River's adaptivity and parallelism features, the interested reader is referred to the original paper on the topic [AAT+99]. In Telegraph, we intend to leverage the adaptability of River to allow for dynamic shifting of load (both query processing and data delivery) in a shared-nothing parallel environment. But in this paper we restrict ourselves to basic (single-site) features of eddies; discussions of eddies in parallel rivers are deferred to Section 6.

Since we do not discuss parallelism here, a very simple overview of the River framework suffices. River is a dataflow query engine, analogous in many ways to Gamma [DGS+90], Volcano [Gra90] and commercial parallel database engines, in which "iterator"-style *modules* (query operators) communicate via a fixed dataflow graph (a query plan). Each module runs as an independent thread, and the edges in the graph correspond to finite message queues. When a producer and consumer run at differing rates, the faster thread may block on the queue waiting for the slower thread to catch up. As in [UFA98], River is multi-threaded and can exploit barrier-free algorithms by reading from various inputs at independent rates. The River implementation we used derives from the work on Now-Sort [AAC+97], and features efficient I/O mechanisms including pre-fetching scans, avoidance of operating system buffering, and high-performance user-level networking.

3.1.1 Pre-Optimization

Although we will use eddies to reorder tables among joins, a heuristic pre-optimizer must choose how to initially pair off relations into joins, with the constraint that each relation participates in only one join. This corresponds to choosing a spanning tree of a query graph, in which nodes represent relations and edges represent binary joins [KBZ86]. One reasonable heuristic for picking a spanning tree forms a chain of cartesian products across any tables known to be very small (to handle "star schemas" when base-table cardinality statistics are available); it then picks arbitrary equijoin edges (on the assumption that they are relatively low selectivity), followed by as many arbitrary non-equijoin edges as required to complete a spanning tree.

Given a spanning tree of the query graph, the pre-optimizer needs to choose join algorithms for each edge. Along each equijoin edge it can use either an index join if an index is available, or a hash ripple join. Along each non-equijoin edge it can use a block ripple join.

These are simple heuristics that we use to allow us to focus on our initial eddy design; in Section 6 we present initial ideas on making spanning tree and algorithm decisions adaptively.

3.2 An Eddy in the River

An eddy is implemented via a module in a river containing an arbitrary number of input relations, a number of participating unary and binary modules, and a single output relation (Figure 1)[3]. An eddy encapsulates the scheduling of its participating operators; tuples entering the eddy can flow through its operators in a variety of orders.

In essence, an eddy explicitly merges multiple unary and binary operators into a single n-ary operator within a query plan, based on the intuition from Section 2.2 that symmetries can be easily captured in an n-ary operator. An eddy module maintains a fixed-sized buffer of tuples that are to be processed by one or more operators. Each operator participating in the eddy has one or two inputs that are fed tuples by the eddy, and an output stream that returns tuples to the eddy. Eddies are so named because of this circular data flow within a river.

A tuple entering an eddy is associated with a tuple descriptor containing a vector of *Ready* bits and *Done* bits, which indicate respectively those operators that are elgibile to process the tuple, and those that have already processed the tuple. The eddy module ships a tuple only to operators for which the corresponding Ready bit turned on. After processing the tuple, the operator returns it to the eddy, and the corresponding Done bit is turned on. If all the Done bits are on, the tuple is sent to the eddy's output; otherwise it is sent to another eligible operator for continued processing.

[3] Nothing prevents the use of n-ary operators with $n > 2$ in an eddy, but since implementations of these are atypical in database query processing we do not discuss them here.

When an eddy receives a tuple from one of its inputs, it zeroes the Done bits, and sets the Ready bits appropriately. In the simple case, the eddy sets all Ready bits on, signifying that any ordering of the operators is acceptable. When there are ordering constraints on the operators, the eddy turns on only the Ready bits corresponding to operators that can be executed initially. When an operator returns a tuple to the eddy, the eddy turns on the Ready bit of any operator eligible to process the tuple. Binary operators generate output tuples that correspond to combinations of input tuples; in these cases, the Done bits and Ready bits of the two input tuples are ORed. In this manner an eddy preserves the ordering constraints while maximizing opportunities for tuples to follow different possible orderings of the operators.

Two properties of eddies merit comment. First, note that eddies represent the full class of bushy trees corresponding to the set of join nodes – it is possible, for instance, that two pairs of tuples are combined independently by two different join modules, and then routed to a third join to perform the 4-way concatenation of the two binary records. Second, note that eddies do not constrain reordering to moments of symmetry across the eddy as a whole. A given operator must carefully refrain from fetching tuples from certain inputs until its next moment of symmetry – e.g., a nested-loops join would not fetch a new tuple from the current outer relation until it finished rescanning the inner. But there is no requirement that *all* operators in the eddy be at a moment of symmetry when this occurs; just the operator that is fetching a new tuple. Thus eddies are quite flexible both in the shapes of trees they can generate, and in the scenarios in which they can logically reorder operators.

4 Routing Tuples in Eddies

An eddy module directs the flow of tuples from the inputs through the various operators to the output, providing the flexibility to allow each tuple to be routed individually through the operators. The routing policy used in the eddy determines the efficiency of the system. In this section we study some promising initial policies; we believe that this is a rich area for future work. We outline some of the remaining questions in Section 6.

An eddy's tuple buffer is implemented as a priority queue with a flexible prioritization scheme. An operator is always given the highest-priority tuple in the buffer that has the corresponding Ready bit set. For simplicity, we start by considering a very simple priority scheme: tuples enter the eddy with low priority, and when they are returned to the eddy from an operator they are given high priority. This simple priority scheme ensures that tuples flow completely through the eddy before new tuples are consumed from the inputs, ensuring that the eddy does not become "clogged" with new tuples.

4.1 Experimental Setup

In order to illustrate how eddies work, we present some initial experiments in this section; we pause briefly here to describe our experimental setup. All our experiments were run on a single-processor Sun Ultra-1 workstation running Solaris 2.6, with 160 MB of RAM. We used the Euphrates implementation of River [AAT+99]. We synthetically generated relations as in Table 1, with 100 byte tuples in each relation.

To allow us to experiment with costs and selectivities of selections, our selection modules are (artificially) implemented as spin loops corresponding to their relative costs, followed by a randomized selection decision with the appropriate selectivity. We describe the relative costs of selections in terms of abstract "delay units"; for studying optimization, the absolute number of cycles through a spin loop are irrelevant. We implemented the simplest version of hash ripple join, identical to the original pipelining hash join [WA91]; our implementation here does not exert any statistically-motivated control over disk resource consumption (as in [HH99]). We simulated index joins by doing random I/Os within a file, returning on average the number of matches corresponding to a pre-programmed selectivity. The filesystem cache was allowed to absorb some of the index I/Os after warming up. In order to fairly compare eddies to static plans, we simulate static plans via eddies that enforce a static ordering on tuples (setting Ready bits in the correct order).

Table	Cardinality	values in column a
R	10,000	500 - 5500
S	80,000	0 - 5000
T	10,000	N/A
U	50,000	N/A

Table 1: Cardinalities of tables; values are uniformly distributed.

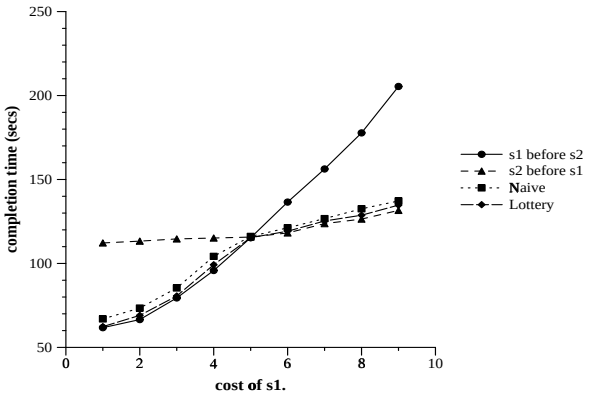


Figure 4: Performance of two 50% selections, $s2$ has cost 5, $s1$ varies across runs.

4.2 Naive Eddy: Fluid Dynamics and Operator Costs

To illustrate how an eddy works, we consider a very simple single-table query with two expensive selection predicates, under the traditional assumption that no performance or selectivity properties change during execution. Our SQL query is simply the following:

```
SELECT   *
  FROM   U
 WHERE   s1() AND s2();
```

In our first experiment, we wish to see how well a "naive" eddy can account for differences in costs among operators. We run the query multiple times, always setting the cost of $s2$ to 5 delay units, and the selectivities of both selections to 50%. In each run we use a different cost for $s1$, varying it between 1 and 9 delay units across runs. We compare a naive eddy of the two selections against both possible static orderings of

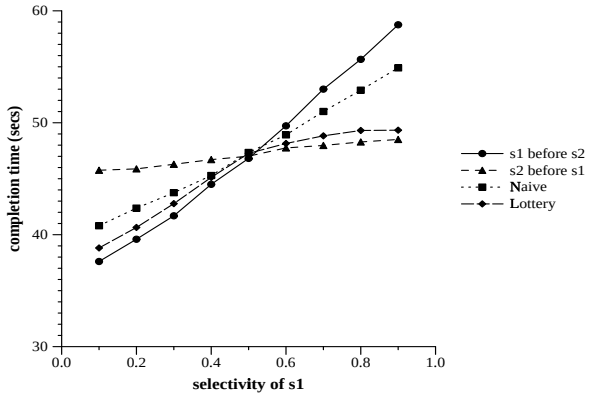


Figure 5: Performance of two selections of cost 5, $s2$ has 50% selectivity, $s1$ varies across runs.

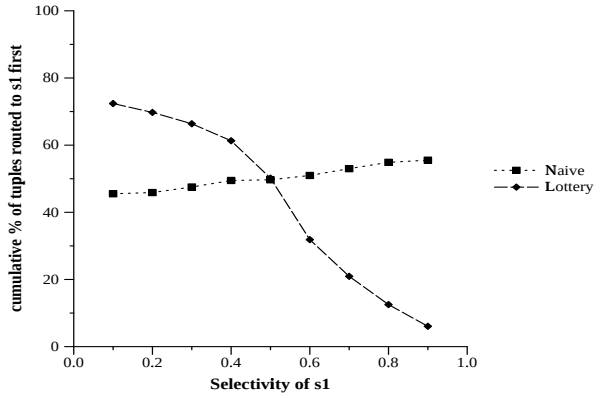


Figure 6: Tuple flow with lottery scheme for the variable-selectivity experiment(Figure 5).

the two selections (and against a "lottery"-based eddy, about which we will say more in Section 4.3.) One might imagine that the flexible routing in the naive eddy would deliver tuples to the two selections equally: half the tuples would flow to $s1$ before $s2$, and half to $s2$ before $s1$, resulting in middling performance over all. Figure 4 shows that this is not the case: the naive eddy nearly matches the better of the two orderings in all cases, without any explicit information about the operators' relative costs.

The naive eddy's effectiveness in this scenario is due to simple fluid dynamics, arising from the different rates of consumption by $s1$ and $s2$. Recall that edges in a River dataflow graph correspond to fixed-size queues. This limitation has the same effect as *back-pressure* in a fluid flow: production along the input to any edge is limited by the rate of consumption at the output. The lower-cost selection (e.g., $s1$ at the left of Figure 4) can consume tuples more quickly, since it spends less time per tuple; as a result the lower-cost operator exerts less back-pressure on the input table. At the same time, the high-cost operator *produces* tuples relatively slowly, so the low-cost operator will rarely be required to consume a high-priority, previously-seen tuple. Thus most tuples are routed to the low-cost operator first, even though the costs are not explicitly exposed or tracked in any way.

4.3 Fast Eddy: Learning Selectivities

The naive eddy works well for handling operators with different costs but equal selectivity. But we have not yet considered differences in selectivity. In our second experiment we keep the costs of the operators constant and equal (5 units), keep the selectivity of $s2$ fixed at 50%, and vary the selectivity of $s1$ across runs. The results in Figure 5 are less encouraging, showing the naive eddy performing as we originally expected, about half-way between the best and worst plans. Clearly our naive priority scheme and the resulting back-pressure are insufficient to capture differences in selectivity.

To resolve this dilemma, we would like our priority scheme to favor operators based on both their consumption and production rate. Note that the consumption (input) rate of an operator is determined by cost alone, while the production (output) rate is determined by a product of cost and selectivity. Since an operator's back-pressure on its input depends largely on its consumption rate, it is not surprising that our naive scheme does not capture differing selectivities.

To track both consumption and production over time, we enhance our priority scheme with a simple learning algorithm implemented via *Lottery Scheduling* [WW94]. Each time the eddy gives a tuple to an operator, it credits the operator one "ticket". Each time the operator returns a tuple to the eddy, one ticket is debited from the eddy's running count for that operator. When an eddy is ready to send a tuple to be processed, it "holds a lottery" among the operators eligible for receiving the tuple. (The interested reader is referred to [WW94] for a simple and efficient implementation of lottery scheduling.) An operator's chance of "winning the lottery" and receiving the tuple corresponds to the count of tickets for that operator, which in turn tracks the relative efficiency of the operator at draining tuples from the system. By routing tuples using this lottery scheme, the eddy tracks ("learns") an ordering of the operators that gives good overall efficiency.

The "lottery" curve in Figures 4 and 5 show the more intelligent lottery-based routing scheme compared to the naive back-pressure scheme and the two static orderings. The lottery scheme handles both scenarios effectively, slightly improving the eddy in the changing-cost experiment, and performing much better than naive in the changing-selectivity experiment.

To explain this a bit further, in Figure 6 we display the percent of tuples that followed the order $s1, s2$ (as opposed to $s2, s1$) in the two eddy schemes; this roughly represents the average ratio of lottery tickets possessed by $s1$ and $s2$ over time. Note that the naive back-pressure policy is barely sensitive to changes in selectivity, and in fact drifts slightly in the wrong direction as the selectivity of $s1$ is increased. By contrast, the lottery-based scheme adapts quite nicely as the selectivity is varied.

In both graphs one can see that when the costs and selectivities are close to equal ($s1 = s2 = 50\%$), the percentage of tuples following the cheaper order is close to 50%. This observation is intuitive, but quite significant. The lottery-based eddy approaches the *cost* of an optimal ordering, but does not concern itself about strictly observing the optimal ordering. Contrast this to earlier work on runtime reoptimization [KD98, UFA98, IFF+99], where a traditional query optimizer runs during processing to determine the optimal plan remnant. By focusing on overall cost rather than on finding

the optimal plan, the lottery scheme probabilistically provides nearly optimal performance with much less effort, allowing re-optimization to be done with an extremely lightweight technique that can be executed multiple times for every tuple.

A related observation is that the lottery algorithm gets closer to perfect routing ($y = 0\%$) on the right of Figure 6 than it does ($y = 100\%$) on the left. Yet in the corresponding performance graph (Figure 5), the differences between the lottery-based eddy and the optimal static ordering do not change much in the two settings. This phenomenon is explained by examining the "jeopardy" of making ordering errors in either case. Consider the left side of the graph, where the selectivity of $s1$ is 10%, $s2$ is 50%, and the costs of each are $c = 5$ delay units. Let e be the rate at which tuples are routed erroneously (to $s2$ before $s1$ in this case). Then the expected cost of the query is $(1 - e) \cdot 1.1c + e \cdot 1.5c = .4ec + 1.1c$. By contrast, in the second case where the selectivity of $s1$ is changed to 90%, the expected cost is $(1 - e) \cdot 1.5c + e \cdot 1.9c = .4ec + 1.5c$. Since the jeopardy is higher at 90% selectivity than at 10%, the lottery more aggressively favors the optimal ordering at 90% selectivity than at 10%.

4.4 Joins

We have discussed selections up to this point for ease of exposition, but of course joins are the more common expensive operator in query processing. In this section we study how eddies interact with the pipelining ripple join algorithms. For the moment, we continue to study a static performance environment, validating the ability of eddies to do well even in scenarios where static techniques are most effective.

We begin with a simple 3-table query:

```
SELECT   *
  FROM   R, S, T
 WHERE   R.a = S.a
   AND   S.b = T.b
```

In our experiment, we constructed a preoptimized plan with a hash ripple join between R and S, and an index join between S and T. Since our data is uniformly distributed, Table 1 indicates that the selectivity of the RS join is 1.8×10^{-4}; its selectivity *with respect to* S is 180% – i.e., each S tuple entering the join finds 1.8 matching R tuples on average [Hel98]. We artificially set the selectivity of the index join w.r.t. S to be 10% (overall selectivity 1×10^{-5}). Figure 7 shows the relative performance of our two eddy schemes and the two static join orderings. The results echo our results for selections, showing the lottery-based eddy performing nearly optimally, and the naive eddy performing in between the best and worst static plans.

As noted in Section 2.2.1, index joins are very analogous to selections. Hash joins have more complicated and symmetric behavior, and hence merit additional study. Figure 8 presents performance of two hash-ripple-only versions of this query. Our in-memory pipelined hash joins all have the same cost. We change the data in R, S and T so that the selectivity of the ST join w.r.t. S is 20% in one version, and 180% in the other. In all runs, the selectivity of the RS join predicate w.r.t. S is fixed at 100%. As the figure shows, the lottery-based eddy continues to perform nearly optimally.

Figure 9 shows the percent of tuples in the eddy that follow one order or the other in all four join experiments. While the eddy is not strict about following the optimal ordering, it is

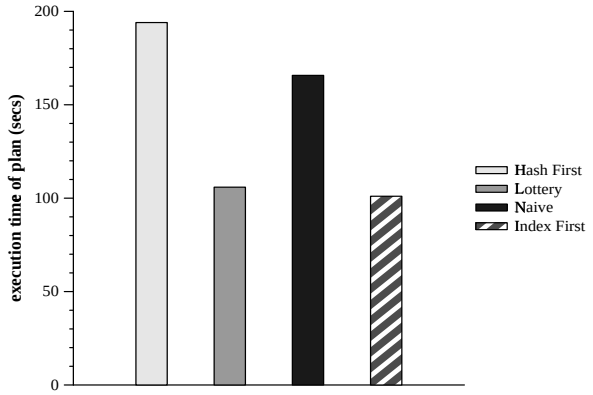


Figure 7: Performance of two joins: a selective Index Join and a Hash Join

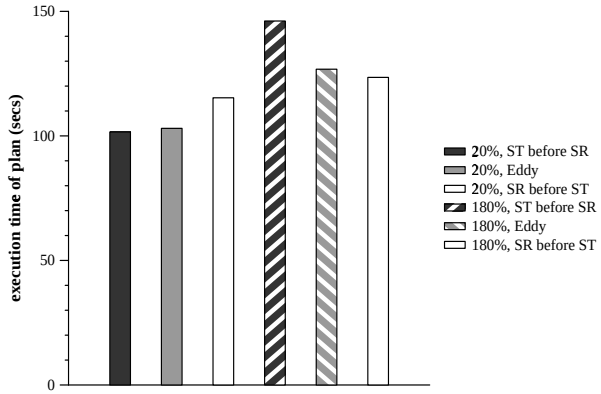


Figure 8: Performance of hash joins $R \bowtie S$ and $S \bowtie T$. $R \bowtie S$ has selectivity 100% w.r.t. S, the selectivity of $S \bowtie T$ w.r.t. S varies between 20% and 180% in the two runs.

quite close in the case of the experiment where the hash join should precede the index join. In this case, the relative cost of index join is so high that the jeopardy of choosing it first drives the hash join to nearly always win the lottery.

4.5 Responding to Dynamic Fluctuations

Eddies should adaptively react over time to the changes in performance and data characteristics described in Section 1.1. The routing schemes described up to this point have not considered how to achieve this. In particular, our lottery scheme weighs all experiences equally: observations from the distant past affect the lottery as much as recent observations. As a result, an operator that earns many tickets early in a query may become so wealthy that it will take a great deal of time for it to lose ground to the top achievers in recent history.

To avoid this, we need to modify our point scheme to forget history to some extent. One simple way to do this is to use a *window* scheme, in which time is partitioned into windows, and the eddy keeps track of two counts for each operator: a number of *banked* tickets, and a number of *escrow* tickets. Banked tickets are used when running a lottery. Escrow tickets are used to measure efficiency during the window. At the beginning of the window, the value of the es-

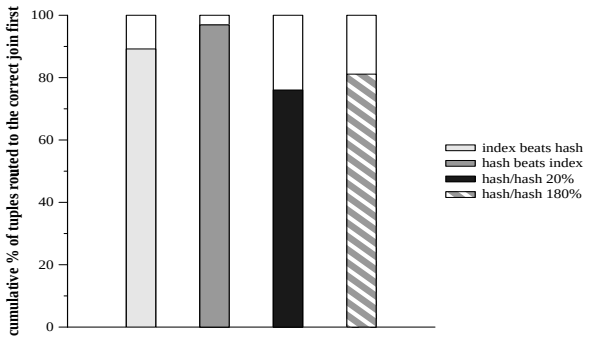


Figure 9: Percent of tuples routed in the optimal order in all of the join experiments.

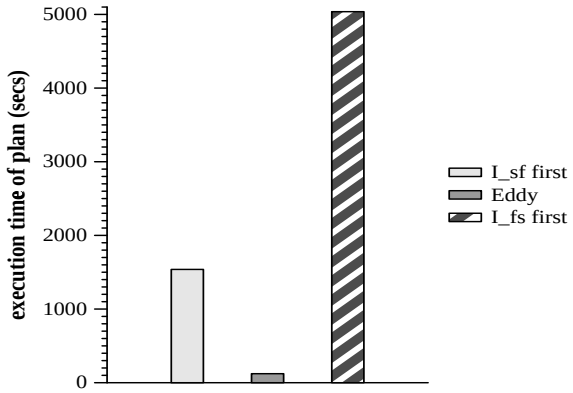


Figure 10: Adapting to changing join costs: performance.

crow account replaces the value of the banked account (i.e., `banked = escrow`), and the escrow account is reset (`escrow = 0`). This scheme ensures that operators "re-prove themselves" each window.

We consider a scenario of a 3-table equijoin query, where two of the tables are external and used as "inner" relations by index joins. Our third relation has 30,000 tuples. Since we assume that the index servers are remote, we implement the "cost" in our index module as a time delay (i.e., `while (gettimeofday() < x) ;`) rather than a spin loop; this better models the behavior of waiting on an external event like a network response. We have two phases in the experiment: initially, one index (call it I_{fs}) is fast (no time delay) and the other (I_{sf}) is slow (5 seconds per lookup). After 30 seconds we begin the second phase, in which the two indexes swap speeds: the I_{fs} index becomes slow, and I_{sf} becomes fast. Both indexes return a single matching tuple 1% of the time.

Figure 10 shows the performance of both possible static plans, compared with an eddy using a lottery with a window scheme. As we would hope, the eddy is much faster than either static plan. In the first static plan (I_{sf} before I_{fs}), the initial index join in the plan is slow in the first phase, processing only 6 tuples and discarding all of them. In the remainder of the run, the plan quickly discards 99% of the tuples, passing 300 to the (now) expensive second join. In the second static

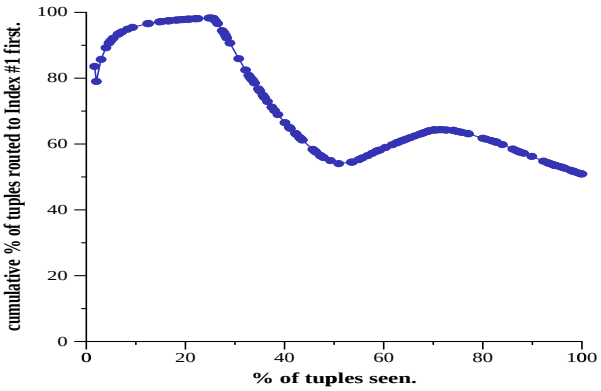


Figure 11: Adapting to changing join costs: tuple movement.

plan (I_{fs} before I_{sf}), the initial join begins fast, processing about 29,000 tuples, and passing about 290 of those to the second (slower) join. After 30 seconds, the second join becomes fast and handles the remainder of the 290 tuples quickly, while the first join slowly processes the remaining 1,000 tuples at 5 seconds per tuple. The eddy outdoes both static plans: in the first phase it behaves identically to the second static plan, consuming 29,000 tuples and queueing 290 for the eddy to pass to I_{sf}. Just after phase 2 begins, the eddy adapts its ordering and passes tuples to I_{sf} – the new fast join – first. As a result, the eddy spends 30 seconds in phase one, and in phase two it has less then 290 tuples queued at I_{sf} (now fast), and only 1,000 tuples to process, only about 10 of which are passed to I_{fs} (now slow).

A similar, more controlled experiment illustrates the eddy's adaptability more clearly. Again, we run a three-table join, with two external indexes that return a match 10% of the time. We read 4,000 tuples from the scanned table, and toggle costs between 1 and 100 cost units every 1000 tuples – i.e., three times during the experiment. Figure 11 shows that the eddy adapts correctly, switching orders when the operator costs switch. Since the cost differential is less dramatic here, the jeopardy is lower and the eddy takes a bit longer to adapt. Despite the learning time, the trends are clear – the eddy sends most of the first 1000 tuples to index #1 first, which starts off cheap. It sends most of the second 1000 tuples to index #2 first, causing the overall percentage of tuples to reach about 50%, as reflected by the near-linear drift toward 50% in the second quarter of the graph. This pattern repeats in the third and fourth quarters, with the eddy eventually displaying an even use of the two orderings over time – always favoring the best ordering.

For brevity, we omit here a similar experiment in which we fixed costs and modified selectivity over time. The results were similar, except that changing only the selectivity of two operators results in less dramatic benefits for an adaptive scheme. This can be seen analytically, for two operators of cost c whose selectivites are swapped from low to hi in a manner analogous to our previous experiment. To lower-bound the performance of either static ordering, selectivities should be toggled to their extremes (100% and 0%) for equal amounts of time – so that half the n tuples go through both operators. Either static plan thus takes $nc+1/2nc$ time, whereas an optimal

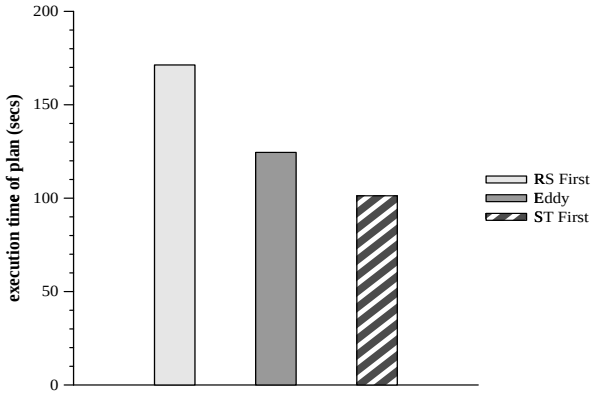


Figure 12: Adapting to an initial delay on R: performance

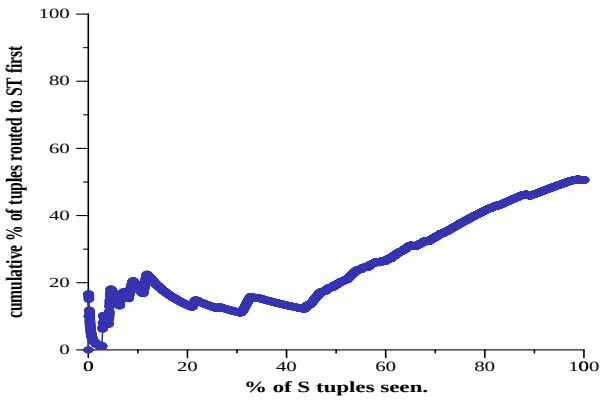


Figure 13: Adapting to an initial delay on R: tuple movement.

dynamic plan takes nc time, a ratio of only 3/2. With more operators, adaptivity to changes in selectivity can become more significant, however.

4.5.1 Delayed Delivery

As a final experiment, we study the case where an input relation suffers from an initial delay, as in [AFTU96, UFA98]. We return to the 3-table query shown in the left of Figure 8, with the RS selectivity at 100%, and the ST selectivity at 20%. We delay the delivery of R by 10 seconds; the results are shown in Figure 12. Unfortunately, we see here that our eddy – even with a lottery and a window-based forgetting scheme – does not adapt to initial delays of R as well as it could. Figure 13 tells some of the story: in the early part of processing, the eddy incorrectly favors the RS join, even though no R tuples are streaming in, and even though the RS join should appear second in a normal execution (Figure 8). The eddy does this because it observes that the RS join does not produce any output tuples when given S tuples. So the eddy awards most S tuples to the RS join initially, which places them in an internal hash table to be subsequently joined with R tuples when they arrive. The ST join is left to fetch and hash T tuples. This wastes resources that could have been spent joining S tuples with T tuples during the delay, and "primes" the RS join to produce a large number of tuples once the Rs begin appearing.

Note that the eddy does far better than pessimally: when R begins producing tuples (at 43.5 on the x axis of Figure 13), the S values bottled up in the RS join burst forth, and the eddy quickly throttles the RS join, allowing the ST join to process most tuples first. This scenario indicates two problems with our implementation. First, our ticket scheme does not capture the growing selectivity inherent in a join with a delayed input. Second, storing tuples inside the hash tables of a single join unnecessarily prevents other joins from processing them; it might be conceivable to hash input tuples within multiple joins, if care were taken to prevent duplicate results from being generated. A solution to the second problem might obviate the need to solve the first; we intend to explore these issues further in future work.

For brevity, we omit here a variation of this experiment, in which we delayed the delivery of S by 10 seconds instead of R. In this case, the delay of S affects both joins identically, and simply slows down the completion time of all plans by about 10 seconds.

5 Related Work

To our knowledge, this paper represents the first general query processing scheme for reordering in-flight operators within a pipeline, though [NWMN99] considers the special case of unary operators. Our characterization of barriers and moments of symmetry also appears to be new, arising as it does from our interest in reoptimizing general pipelines.

Recent papers consider reoptimizing queries at the ends of pipelines [UFA98, KD98, IFF+99], reordering operators only after temporary results are materialized. [IFF+99] observantly notes that this approach dates back to the original INGRES query decomposition scheme [SWK76]. These inter-pipeline techniques are not adaptive in the sense used in traditional control theory (e.g., [Son98]) or machine learning (e.g., [Mit97]); they make decisions without any ongoing feedback from the operations they are to optimize, instead performing static optimizations at coarse-grained intervals in the query plan. One can view these efforts as complementary to our work: eddies can be used to do tuple scheduling within pipelines, and techniques like those of [UFA98, KD98, IFF+99] can be used to reoptimize across pipelines. Of course such a marriage sacrifices the simplicity of eddies, requiring both the traditional complexity of cost estimation and plan enumeration along with the ideas of this paper. There are also significant questions on how best to combine these techniques – e.g., how many materialization operators to put in a plan, which operators to put in which eddy pipelines, etc.

DEC Rdb (subsequently Oracle Rdb) used competition to choose among different access methods [AZ96]. Rdb briefly observed the performance of alternative access methods at runtime, and then fixed a "winner" for the remainder of query execution. This bears a resemblance to sampling for cost estimation (see [BDF+97] for a survey). More distantly related is the work on "parameterized" or "dynamic" query plans, which postpone some optimization decisions until the beginning of query execution [INSS97, GC94].

The initial work on Query Scrambling [AFTU96] studied network unpredictabilities in processing queries over wide-area sources. This work materialized remote data while processing was blocked waiting for other sources, an idea that can be used in concert with eddies. Note that local materialization ameliorates but does not remove barriers: work to be

done locally after a barrier can still be quite significant. Later work focused on rescheduling runnable sub-plans during initial delays in delivery [UFA98], but did not attempt to reorder in-flight operators as we do here.

Two out-of-core versions of the pipelined hash join have been proposed recently [IFF+99, UF99]. The X-Join [UF99] enhances the pipelined hash join not only by handling the out-of-core case, but also by exploiting delay time to aggressively match previously-received (and spilled) tuples. We intend to experiment with X-Joins and eddies in future work.

The Control project [HAC+99] studies interactive analysis of massive data sets, using techniques like online aggregation, online reordering and ripple joins. There is a natural synergy between interactive and adaptive query processing; online techniques to pipeline best-effort answers are naturally adaptive to changing performance scenarios. The need for optimizing pipelines in the Control project initially motivated our work on eddies. The Control project [HAC+99] is not explicitly related to the field of control theory [Son98], though eddies appears to link the two in some regards.

The River project [AAT+99] was another main inspiration of this work. River allows modules to work as fast as they can, naturally balancing flow to whichever modules are faster. We carried the River philosophy into the intial back-pressure design of eddies, and intend to return to the parallel load-balancing aspects of the optimization problem in future work.

In addition to commercial projects like those in Section 1.2, there have been numerous research systems for heterogeneous data integration, e.g. [GMPQ+97, HKWY97, IFF+99], etc.

6 Conclusions and Future Work

Query optimization has traditionally been viewed as a coarse-grained, static problem. Eddies are a query processing mechanism that allow fine-grained, adaptive, online optimization. Eddies are particularly beneficial in the unpredictable query processing environments prevalent in massive-scale systems, and in interactive online query processing. They fit naturally with algorithms from the Ripple Join family, which have frequent moments of symmetry and adaptive or non-existent synchronization barriers. Eddies can be used as the sole optimization mechanism in a query processing system, obviating the need for much of the complex code required in a traditional query optimizer. Alternatively, eddies can be used in concert with traditional optimizers to improve adaptability within pipelines. Our initial results indicate that eddies perform well under a variety of circumstances, though some questions remain in improving reaction time and in adaptively choosing join orders with delayed sources. We are sufficiently encouraged by these early results that we are using eddies and rivers as the basis for query processing in the Telegraph system.

In order to focus our energies in this initial work, we have explicitly postponed a number of questions in understanding, tuning, and extending these results. One main challenge is to develop eddy "ticket" policies that can be formally proved to converge quickly to a near-optimal execution in static scenarios, and that adaptively converge when conditions change. This challenge is complicated by considering both selections and joins, including hash joins that "absorb" tuples into their hash tables as in Section 4.5.1. We intend to focus on multiple performance metrics, including time to completion, the rate of output from a plan, and the rate of refinement for online aggregation estimators. We have also begun studying schemes to allow eddies to effectively order dependent predicates, based on reinforcement learning [SB98]. In a related vein, we would like to automatically tune the aggressiveness with which we forget past observations, so that we avoid introducing a tuning knob to adjust window-length or some analogous constant (e.g., a hysteresis factor).

Another main goal is to attack the remaining static aspects of our scheme: the "pre-optimization" choices of spanning tree, join algorithms, and access methods. Following [AZ96], we believe that competition is key here: one can run multiple redundant joins, join algorithms, and access methods, and track their behavior in an eddy, adaptively choosing among them over time. The implementation challenge in that scenario relates to preventing duplicates from being generated, while the efficiency challenge comes in not wasting too many computing resources on unpromising alternatives.

A third major challenge is to harness the parallelism and adaptivity available to us in rivers. Massively parallel systems are reaching their limit of manageability, even as data sizes continue to grow very quickly. Adaptive techniques like eddies and rivers can significantly aid in the manageability of a new generation of massively parallel query processors. Rivers have been shown to adapt gracefully to performance changes in large clusters, spreading query processing load across nodes and spreading data delivery across data sources. Eddies face additional challenges to meet the promise of rivers: in particular, reoptimizing queries with intra-operator parallelism entails repartitioning data, which adds an expense to reordering that was not present in our single-site eddies. An additional complication arises when trying to adaptively adjust the degree of partitioning for each operator in a plan. On a similar note, we would like to explore enhancing eddies and rivers to tolerate failures of sources or of participants in parallel execution.

Finally, we are exploring the application of eddies and rivers to the generic space of dataflow programming, including applications such as multimedia analysis and transcoding, and the composition of scalable, reliable internet services [GWBC99]. Our intent is for rivers to serve as a generic parallel dataflow engine, and for eddies to be the main scheduling mechanism in that environment.

Acknowledgments

Vijayshankar Raman provided much assistance in the course of this work. Remzi Arpaci-Dusseau, Eric Anderson and Noah Treuhaft implemented Euphrates, and helped implement eddies. Mike Franklin asked hard questions and suggested directions for future work. Stuart Russell, Christos Papadimitriou, Alistair Sinclair, Kris Hildrum and Lakshminarayanan Subramanian all helped us focus on formal issues. Thanks to Navin Kabra and Mitch Cherniack for initial discussions on run-time reoptimization, and to the database group at Berkeley for feedback. Stuart Russell suggested the term "eddy".

This work was done while both authors were at UC Berkeley, supported by a grant from IBM Corporation, NSF grant IIS-9802051, and a Sloan Foundation Fellowship. Computing and network resources for this research were provided through NSF RI grant CDA-9401156.

References

[AAC+97] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-Performance Sorting on Networks of Workstations. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997.

[AAT+99] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, May 1999.

[AFTU96] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling Query Plans to Cope With Unexpected Delays. In *4th International Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, December 1996.

[AH99] R. Avnur and J. M. Hellerstein. Continuous query optimization. Technical Report CSD-99-1078, University of California, Berkeley, November 1999.

[Aok99] P. M. Aoki. How to Avoid Building DataBlades That Know the Value of Everything and the Cost of Nothing. In *11th International Conference on Scientific and Statistical Database Management*, Cleveland, July 1999.

[AZ96] G. Antoshenkov and M. Ziauddin. Query Processing and Optimization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.

[Bar99] R. Barnes. Scale Out. In *High Performance Transaction Processing Workshop (HPTS '99)*, Asilomar, September 1999.

[BDF+97] D. Barbara, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey Data Reduction Report. *IEEE Data Engineering Bulletin*, 20(4), December 1997.

[BO99] J. Boulos and K. Ono. Cost Estimation of User-Defined Methods in Object-Relational Database Systems. *SIGMOD Record*, 28(3):22–28, September 1999.

[DGS+90] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, Mar 1990.

[DKO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 1–8, Boston, June 1984.

[FMLS99] D. Florescu, I. Manolescu, A. Levy, and D. Suciu. Query Optimization in the Presence of Limited Access Patterns. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Phildelphia, June 1999.

[GC94] G. Graefe and R. Cole. Optimization of Dynamic Query Evaluation Plans. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Minneapolis, 1994.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[Gra90] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 102–111, Atlantic City, May 1990.

[GWBC99] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proceedings of the 1999 Usenix Annual Technical Conference*, Monterey, June 1999.

[HAC+99] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive Data Analysis: The Control Project. *IEEE Computer*, 32(8):51–59, August 1999.

[Hel98] J. M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. *ACM Transactions on Database Systems*, 23(2):113–157, 1998.

[HH99] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 287–298, Philadelphia, 1999.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proc. 23rd International Conference on Very Large Data Bases (VLDB)*, Athens, 1997.

[HSC99] J. M. Hellerstein, M. Stonebraker, and R. Caccia. Open, Independent Enterprise Data Integration. *IEEE Data Engineering Bulletin*, 22(1), March 1999. http://www.cohera.com.

[IFF+99] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Philadelphia, 1999.

[IK84] T. Ibaraki and T. Kameda. Optimal Nesting for Computing N-relational Joins. *ACM Transactions on Database Systems*, 9(3):482–502, October 1984.

[INSS97] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. *VLDB Journal*, 6(2):132–151, 1997.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. In *Proc. 12th International Conference on Very Large Databases (VLDB)*, pages 128–137, August 1986.

[KD98] N. Kabra and D. J. DeWitt. Efficient Mid-Query Reoptimization of Sub-Optimal Query Execution Plans. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 106–117, Seattle, 1998.

[Met97] R. Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the Usenix 1997 Technical Conference*, Anaheim, January 1997.

[Mit97] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[NWMN99] K. W. Ng, Z. Wang, R. R. Muntz, and S. Nittel. Dynamic Query Re-Optimization. In *11th International Conference on Scientific and Statistical Database Management*, Cleveland, July 1999.

[RPK+99] B. Reinwald, H. Pirahesh, G. Krishnamoorthy, G. Lapis, B. Tran, and S. Vora. Heterogeneous Query Processing Through SQL Table Functions. In *15th International Conference on Data Engineering*, pages 366–373, Sydney, March 1999.

[RRH99] V. Raman, B. Raman, and J. M. Hellerstein. Online Dynamic Reordering for Interactive Data Processing. In *Proc. 25th International Conference on Very Large Data Bases (VLDB)*, pages 709–720, Edinburgh, 1999.

[SB98] R. S. Sutton and A. G. Bartow. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.

[SBH98] M. Stonebraker, P. Brown, and M. Herbach. Interoperability, Distributed Applications, and Distributed Databases: The Virtual Table Interface. *IEEE Data Engineering Bulletin*, 21(3):25–34, September 1998.

[Son98] E. D. Sontag. *Mathematical Control Theory: Deterministic Finite-Dimensional Systems, Second Edition*. Number 6 in Texts in Applied Mathematics. Springer-Verlag, New York, 1998.

[SWK76] M. R. Stonebraker, E. Wong, and P. Kreps. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[UF99] T. Urhan and M. Franklin. XJoin: Getting Fast Answers From Slow and Bursty Networks. Technical Report CS-TR-3994, University of Maryland, February 1999.

[UFA98] T. Urhan, M. Franklin, and L. Amsaleg. Cost-Based Query Scrambling for Initial Delays. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Seattle, June 1998.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. First International Conference on Parallel and Distributed Info. Sys. (PDIS)*, pages 68–77, 1991.

[WW94] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 1–11, Monterey, CA, November 1994. USENIX Assoc.

The Click Modular Router

EDDIE KOHLER, ROBERT MORRIS, BENJIE CHEN, JOHN JANNOTTI,
and M. FRANS KAASHOEK
Laboratory for Computer Science, MIT

Click is a new software architecture for building flexible and configurable routers. A Click router is assembled from packet processing modules called *elements*. Individual elements implement simple router functions like packet classification, queueing, scheduling, and interfacing with network devices. A router configuration is a directed graph with elements at the vertices; packets flow along the edges of the graph. Several features make individual elements more powerful and complex configurations easier to write, including *pull connections*, which model packet flow driven by transmitting hardware devices, and *flow-based router context*, which helps an element locate other interesting elements.

Click configurations are modular and easy to extend. A standards-compliant Click IP router has sixteen elements on its forwarding path; some of its elements are also useful in Ethernet switches and IP tunneling configurations. Extending the IP router to support dropping policies, fairness among flows, or Differentiated Services simply requires adding a couple elements at the right place. On conventional PC hardware, the Click IP router achieves a maximum loss-free forwarding rate of 333,000 64-byte packets per second, demonstrating that Click's modular and flexible architecture is compatible with good performance.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Packet-switching networks*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms: Design, Management, Performance

Additional Key Words and Phrases: Routers, component systems, software router performance

An article describing a previous version of this system was published in *Operating Systems Review* **34**(5) (*Proceedings of the 17th Symposium on Operating Systems Principles*), pp 217–231, December 1999.
This research was supported by a National Science Foundation (NSF) Young Investigator Award and the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory under agreement number F30602-97-2-0288. In addition, Eddie Kohler was supported by a National Science Foundation Graduate Research Fellowship.
Address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139; email: {eddietwo; rtm; benjie; jj; kaashoek}@lcs.mit.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or `permissions@acm.org`.

1. INTRODUCTION

Routers are increasingly expected to do more than route packets. Boundary routers, which lie on the borders between organizations, must often prioritize traffic, translate network addresses, tunnel and filter packets, and act as firewalls, among other things. Furthermore, fundamental properties like packet dropping policies are still under active research [Floyd and Jacobson 1993; Lakshman et al. 1996; Cisco Corporation 1999], and initiatives like Differentiated Services [Blake et al. 1998] bring the need for flexibility close to the core of the Internet.

Unfortunately, most routers have closed, static, and inflexible designs. Network administrators may be able to turn router functions on or off, but they cannot easily specify or even identify the interactions of different functions. Furthermore, it is difficult for network administrators and third party software vendors to extend a router with new functions. Extensions require access to software interfaces in the forwarding path, but these often don't exist, don't exist at the right point, or aren't published.

This paper presents Click, a flexible, modular software architecture for creating routers. Click routers are built from fine-grained components; this supports fine-grained extensions throughout the forwarding path. The components are packet processing modules called *elements*. The basic element interface is narrow, consisting mostly of functions for initialization and packet handoff, but elements can extend it to support other functions (such as reporting queue lengths). To build a router configuration, the user chooses a collection of elements and connects them into a directed graph. The graph's edges, which are called *connections*, represent possible paths for packet handoff. To extend a configuration, the user can write new elements or compose existing elements in new ways, much as UNIX allows one to build complex applications directly or by composing simpler ones using pipes.

Several aspects of the Click architecture were directly inspired by properties of routers. First, packet handoff along a connection may be initiated by either the source end (*push processing*) or the destination end (*pull processing*). This cleanly models most router packet flow patterns, and pull processing makes it possible to write composable packet schedulers. Second, the *flow-based router context* mechanism lets an element automatically locate other elements on which it depends; it is based on the observation that relevant elements are often connected by the flow of packets. These features make individual elements more powerful and configurations easier to write. For example, in Section 4.2, we present an element that can implement four variants of the Random Early Detection dropping policy [Floyd and Jacobson 1993] depending on its context in the router: RED, RED over multiple queues, weighted RED, or drop-from-front RED. This would be difficult or impossible to achieve in previous modular networking systems.

We have implemented Click on general-purpose PC hardware as an extension to the Linux kernel. A modular Click IP router configuration has a maximum loss-free forwarding rate of 333,000 64-byte packets per second on a 700 MHz Pentium III. This is roughly four times the rate for a standard Linux router on

the same hardware, mostly due to device handling improvements inspired by previous work [Mogul and Ramakrishnan 1997; Druschel et al. 1994]. It is also slightly more than the rate for a modified Linux router that uses our network device extensions, demonstrating that Click's modularity and flexibility comes at an acceptable cost.

In the rest of this paper, we describe Click's architecture in detail, including the language used to describe configurations (Section 2), then present a standards-compliant Click IP router (Section 3) and some router extensions (Section 4). After summarizing Click's kernel environment (Section 5) and evaluating its performance (Section 6), we describe related work (Section 7) and summarize our conclusions (Section 8).

2. ARCHITECTURE

A Click *element* represents a unit of router processing. An element represents a conceptually simple computation, such as decrementing an IP packet's time-to-live field, rather than a large, complex computation, such as IP routing. A Click router configuration is a directed graph with elements at the vertices. An edge, or *connection*, between two elements represents a possible path for packet transfer. Every action performed by a Click router's software is encapsulated in an element, from device handling and routing table lookups to queueing and counting packets. The user determines what a Click router does by choosing the elements to be used and the connections among them. Inside a running router, each element is a C++ object that may maintain private state. Connections are represented as pointers to element objects, and passing a packet along a connection is implemented as a single virtual function call.

The most important properties of an element are:

—**Element class.** Each element belongs to one element class. This specifies the code that should be executed when the element processes a packet, as well as the element's initialization procedure and data layout.

—**Ports.** An element can have any number of input and output ports. Every connection goes from an output port on one element to an input port on another. Different ports can have different semantics; for example, second output ports are often used to emit erroneous packets.

—**Configuration string.** The optional configuration string contains additional arguments that are passed to the element at router initialization time. Many element classes use these arguments to set per-element state and fine-tune their behavior.

—**Method interfaces.** Each element supports one or more method interfaces. Every element supports the simple packet-transfer interface, but elements can create and export arbitrary additional interfaces; for example, a queue might export an interface that reports its length. Elements communicate at run time through these interfaces, which can contain both methods and data.

Figure 1 shows how we diagram these properties for a sample element, *Tee(2)*. '*Tee*' is the element class; a *Tee* copies each packet received on its single input port, sending one copy to each output port. Configuration strings are enclosed

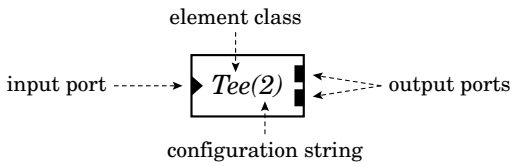


Fig. 1. A sample element. Triangular ports are inputs and rectangular ports are outputs.

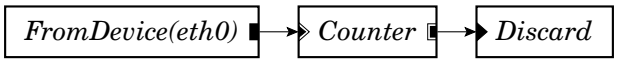


Fig. 2. A router configuration that throws away all packets.

in parentheses; the '*2*' in '*Tee(2)*' is interpreted by *Tee* as a request for two outputs. Method interfaces are not shown explicitly, as they are implied by the element class. Figure 2 shows several elements connected into a simple router that counts incoming packets, then throws them all away.

2.1 Push and Pull Connections

Click supports two kinds of connections, *push* and *pull*. On a push connection, packets start at the source element and are passed downstream to the destination element. This corresponds to the way packets move through most software routers. On a pull connection, in contrast, the destination element initiates packet transfer: it asks the source element to return a packet, or a null pointer if no packet is available. This is the dual of a push connection. (Clark called pull connections *upcalls* [Clark 1985].) Each of these forms of packet transfer is implemented by one virtual function call.

The processing type of a connection—whether it is push or pull—is determined by the ports at its endpoints. Each port in a running router is either push or pull; connections between two push ports are push, and connections between two pull ports are pull. Connections between a push port and a pull port are illegal. Elements set their ports' types as the router is initialized. They may also create *agnostic ports*, which behave as push when connected to push ports and pull when connected to pull ports. When a router is initialized, the system propagates constraints until every agnostic port has been assigned to either push or pull.[1] In our configuration diagrams, black ports are push and white ports are pull; agnostic ports are shown as push or pull ports with a double outline. Figure 3 shows how push and pull work in a simple router.

Push processing is appropriate when unsolicited packets arrive at a Click router—for example, when packets arrive from a device. The router must handle such packets as they arrive, if only to queue them for later consideration. Pull processing is appropriate when the Click router needs to control the timing of packet processing. For example, a router may transmit a packet only when the transmitting device is ready. In Click, transmitting devices are elements

[1]The simplest way of creating an agnostic port causes each packet handoff to that port to take two virtual function calls rather than one. The first calls a general `push` or `pull` method, which is a wrapper that calls the element's "`agnostic`" method.

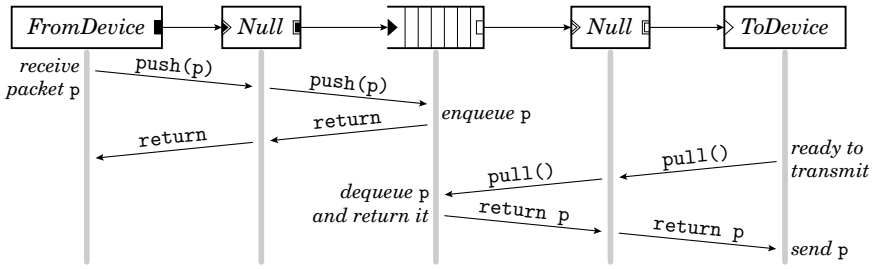


Fig. 3. Push and pull control flow. This diagram shows functions called as a packet moves through a simple router; time moves downwards. The central element is a *Queue*. During the push, control flow moves forward through the element graph starting at the receiving device; during the pull, control flow moves backward through the graph, starting at the transmitting device. The packet p always moves forward.

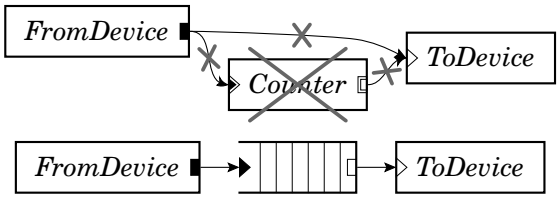


Fig. 4. Some push and pull violations. The top configuration has four errors: (1) *FromDevice*'s push output connects to *ToDevice*'s pull input; (2) more than one connection to *FromDevice*'s push output; (3) more than one connection to *ToDevice*'s pull input; and (4) an agnostic element, *Counter*, in a mixed push/pull context. The bottom configuration, which includes a *Queue*, is legal. In a properly configured router, the port colors on either end of each connection will match.

with one pull input; they therefore initiate packet transfer, and can ask for packets only when they are ready.

Pull processing also models the scheduling decision inherent in choosing the next packet to send. A Click packet scheduler is simply an element with one pull output and multiple pull inputs. Such an element responds to a pull request by choosing one of its inputs, making a pull request to that input, and returning the packet it receives. (If it receives a null pointer, it will generally try another input.) These elements make only local decisions: different scheduling behaviors correspond to different algorithms for choosing an input. Thus, they are easily composable. Section 4.1 discusses this further.

The following properties hold for all correctly configured routers: Push outputs must be connected to push inputs, and pull outputs must be connected to pull inputs. Each agnostic port must be used as push or pull exclusively. Furthermore, if packets arriving on an agnostic input might be emitted immediately on one of that element's agnostic outputs, then both input and output must be used in the same way (either push or pull). Finally, push outputs and pull inputs must be connected exactly once. This ensures that each packet transfer request—either pushing to an output port or pulling from an input port—is along a unique connection. These properties are automatically checked by the system during router initialization. Figure 4 demonstrates some property violations.

These properties are designed to catch intuitively invalid configurations. For example, the connection in Figure 4 from *FromDevice* to *ToDevice* is illegal because *FromDevice*'s output is push while *ToDevice*'s input is pull. But this connection is intuitively illegal, since it would mean that *ToDevice* might receive packets when it was not ready to send them. The *Queue* element, which converts from push to pull, also provides the temporary packet storage this configuration requires.

2.2 Packet Storage

Click elements do not have implicit queues on their input and output ports, or the associated performance and complexity costs. Instead, queues in Click are explicit objects, implemented by a separate *Queue* element. This gives the router designer explicit control over an important router property, namely how packets are stored. It also enables valuable configurations that are difficult to arrange otherwise—for example, a single queue feeding multiple devices, or a queue feeding a traffic shaper on the way to a device. Explicit queues necessitate both push and pull connections. A *Queue* has a push input port and a pull output port; the input port responds to pushed packets by enqueueing them, and the output port responds to pull requests by dequeueing packets and returning them.

2.3 CPU Scheduling

Click schedules the router's CPU with a task queue; the router's driver is a loop that processes the task queue one element at a time. (The task queue is currently scheduled with the flexible and lightweight stride scheduling algorithm [Waldspurger and Weihl 1995].) Each task is simply an element that would like special access to CPU time. Thus, the element is Click's unit of CPU scheduling as well as its unit of packet processing. A task can initiate an arbitrary sequence of push and pull requests. However, most elements are never placed on the task queue; they are implicitly scheduled when their `push` or `pull` methods are called. An element should be on the task queue if it frequently initiates push or pull requests without receiving a corresponding request. This includes device elements—for example, *FromDevice* initiates a push request when it receives a packet from a device—as well as elements that ship packets between *Queue*s inside the router configuration.

Another task structure handles timer events. Each element can have any number of active timers, where each timer calls an arbitrary method when it fires.

Click currently runs in a single thread. Thus, any packet transfer method (`push` or `pull`) must return to its caller before another task can begin. The router will continue to process each pushed packet, following it from element to element along a path in the router graph, until it is explicitly stored or dropped (and similarly for pull requests). Therefore, the placement of *Queue*s in a configuration graph determines how CPU scheduling may be performed. For example, if *Queue*s are late in the graph, the router commits to a lot of processing on each input packet before processing the next input packet—specifically, it commits to the processing required to push the packet to a *Queue*.

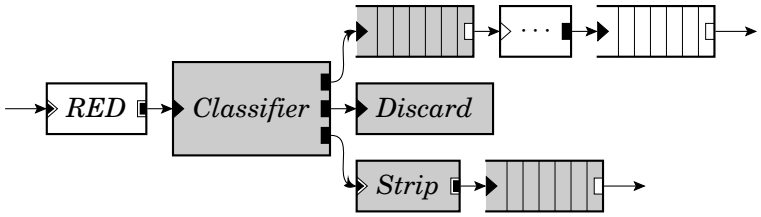


Fig. 5. Flow-based router context. A packet starting at *RED* and stopping at the first *Queue* it encountered might pass through any of the grey elements.

2.4 Flow-Based Router Context

If an element A wants to use a method interface of another element B, it must first locate B. Connections solve this problem for packet transfer, but not for other method interface interactions. Instead, A can refer to B by name (for example, A's configuration string could contain the string "B"), or it can use an automatic mechanism called *flow-based router context*.

Flow-based router context describes where packets starting at a given element might end up after several transfers, or where packets arriving at that element might have originated. This generalizes connections, which specify where a packet might travel in exactly one transfer. In practical terms, elements may ask the system questions such as "If I were to emit a packet on my second output, where might it go?"; the answer is computed using flow-based router context. The element may restrict this question by method interface; when searching for *Queue*s, it might ask "If I were to emit a packet on my second output, which *Queue*s might it encounter?" It may further restrict the answer to the closest method interfaces: "If I were to emit a packet on my second output, and it stopped at the first *Queue* it encountered, where might it stop?" This occupies a useful middle ground between purely local information (connections) and purely global information (the entire router). It can be more robust than naming elements explicitly, since it automatically adapts to changes in the router configuration. It also captures a fundamental router property: if two elements interact, then packets can usually pass from one to the other.

For example, each *Queue* exports its current length using a method interface. Elements such as *RED* (a dropping policy element described further in Section 4.2) are interested in this information, but before finding a *Queue*'s length, *RED* must find the correct *Queue*. It can do so with flow-based router context. Figure 5 shows the router context downstream of *RED*. Every element in the figure is downstream of *RED*, but only the grey elements are found if the search stops at the closest *Queue*s. Thus, *RED*'s flow-based router context search will return the two grey *Queue*s.

Flow-based router context is calculated by a simple data-flow algorithm that is robust in the presence of cycles in the configuration graph. Elements generally ask for flow-based router context once, at router initialization time, and save its results for quick reference as the router runs. Any element that uses flow-based router context must be prepared to handle zero, one, two, or more

```
// Declare three elements ...
src :: FromDevice(eth0);
ctr :: Counter;
sink :: Discard;
// ... and connect them together
src -> ctr;
ctr -> sink;

// Alternate definition using syntactic sugar
FromDevice(eth0) -> Counter -> Discard;
```

Fig. 6. Two Click-language definitions for the trivial router of Figure 2.

result elements, possibly by reporting an error if there are too many or too few results. Section 4.2 demonstrates the flexibility benefits of supporting variable numbers of results.

2.5 Language

Click configurations are written in a simple language with two important constructs: *declarations* create elements, and *connections* say how they should be connected. Its syntax is easy enough to learn from an example; Figure 6 uses it to define a trivial router.

The language contains an abstraction mechanism called *compound elements* that lets users define their own element classes. A compound element is a router configuration fragment that behaves like an element class. A user could define a compound element consisting of a *Queue* followed by a *Shaper*, and call the resulting element class *ShapedQueue*; the rest of the configuration could then use *ShapedQueue* as if it were a primitive Click element class. Compound elements can have multiple ports with arbitrary push/pull characteristics. At initialization time, each use of a compound element is compiled into the corresponding collection of simple elements.

The language is wholly declarative: it specifies what elements to create and how they should be connected, not how to process packets procedurally. This makes configurations easy to manipulate—processing a language file loses no information, and there is a canonical language file for any configuration. Router manipulation tools can take advantage of these properties to optimize router configurations off line or prove simple properties about them.

Language extensions are generally implemented through special element classes rather than new syntactic constructs. For example, the language has no special syntax for specifying CPU scheduling priorities. Instead, the user writes this information in the configuration strings of one or more *ScheduleInfo* elements. These elements have no input or output ports, and packets do not pass through them; they are used only at initialization time, for the data in their configuration strings. This design keeps the language clean and simplifies tool maintenance.

2.6 Installing Configurations

There are currently two drivers that can run Click router configurations, a Linux in-kernel driver and a user-level driver that communicates with the net-

work using Berkeley packet filters [McCanne and Jacobson 1993] or a similar packet socket mechanism. The user-level driver is most useful for profiling and debugging, while the in-kernel driver is good for production work. The rest of this paper concentrates on the in-kernel driver.

To install a Click router configuration, the user passes a Click-language file to the kernel driver. The driver then parses the file, checks it for errors, initializes every element, and puts the router on line. It breaks the initialization process into stages, allowing cyclic configurations without forcing any particular initialization order. In the early stages, elements set object variables, add and remove ports, and specify whether those ports are push or pull. In later stages, they query flow-based router context, place themselves on the task queue, and attach to Linux kernel structures.

Installing a new configuration normally destroys any old configuration; for instance, any packets stored in old queues are dropped. This starts the new configuration from a predictable empty state. However, Click supports two techniques for changing a configuration without losing information:

—**Handlers.** Each element can easily install any number of *handlers*, which are access points for user interaction. They appear to the user as files in Linux's `/proc` file system; for example, a `count` handler belonging to an element named `c` would be accessible in the file `/proc/click/c/count`. One of `c`'s methods is called when the user reads or writes this file. This lightweight mechanism is most appropriate for modifications local to an element, such as changing a maximum queue length. A Click routing table element, for example, would likely provide `add_route` and `del_route` handlers as access points for user-level routing protocol implementations. Handlers are also useful for exporting statistics and other element information.

—**Hot swapping.** Some configuration changes, such as adding new elements, are more complex than handlers can support. In these cases, the user can write a new configuration file and install it with a hot-swapping option. This will only install the new configuration if it initializes correctly—if there are any errors, the old configuration will continue routing packets without a break. Also, if the new configuration is correct, it will atomically take the old configuration's state before being placed on line; for example, any enqueued packets are moved into the new configuration. This happens only with element classes that explicitly support it, and can be prevented by giving the new elements different names than the old ones.

Finally, element class definitions can be dynamically added to and removed from the Click kernel driver; combined with hot swapping, this makes Click an interesting platform for active networking.

2.7 Element Implementation

Each Click element class corresponds to a subclass of the C++ class `Element`, which has about 20 virtual functions. `Element` provides reasonable default implementations for many of these, so most subclasses must override just six of them or less. Only three virtual functions are used during router operation, namely `push`, `pull`, and `run_scheduled` (used by the task scheduler); the others

```
class NullElement: public Element { public:
  NullElement()                        { add_input(); add_output(); }
  const char *class_name() const       { return "Null"; }
  NullElement *clone() const           { return new NullElement; }
  const char *processing() const       { return AGNOSTIC; }
  void push(int port, Packet *p)       { output(0).push(p); }
  Packet *pull(int port)               { return input(0).pull(); }
};
```

Fig. 7. The complete implementation of a do-nothing element: *Null* passes packets from its single input to its single output unchanged.

are used for identification, push and pull specification, configuration, initialization, and statistics.

Subclasses of `Element` are easy to write, so we expect that users will easily write new element classes as needed. In fact, the complete implementation of a simple working element class takes less than 10 lines of code; see Figure 7. Most elements define functions for configuration string parsing and initialization in addition to those in Figure 7, and take about 120 lines of code. When linked with 100 common elements, the Linux kernel driver contains roughly 14,000 lines of core and library source code and 19,000 lines of element source code (not counting comments); this compiles to about 341,000 bytes of i386 instructions, most of which are used only at router initialization time. A simple element's `push` or `pull` function compiles into a few dozen i386 instructions.

2.8 Element Design and Architectural Limitations

A Click user will generally prefer fine-grained elements, which have simple specifications, to coarse-grained elements with more complex specifications. For IP routing, for example, a collection of small elements is preferable to a single element, since the collection of small elements supports arbitrary extensions and modifications via configuration graph manipulation. However, small elements are not appropriate for all problems, since Click's main organizational principle is packet flow. Coarse-grained elements are required when control or data flow doesn't match the flow of packets. For example, complex protocol processing often requires a coarse-grained element; a routing protocol like BGP does not naturally break into parts among which packets flow.

A conventional router contains shared structures that don't participate in packet forwarding, such as routing tables, network statistics, and so forth. In Click, these structures are more naturally incorporated into the packet forwarding path. Routing tables, such as the IP routing table and the ARP cache, are encapsulated by elements that make routing decisions, and statistics are localized inside the elements responsible for collecting them. Of course, these elements can export a method interface so other elements can access the structures.

Several other modular networking systems are built around an abstraction that represents individual network flows [Hutchinson and Peterson 1991; Mosberger and Peterson 1996]. These systems automatically create and destroy modules as network flows are encountered. This is a fast, limited form of con-

figuration installation, as each new or deleted flow changes a localized section of the configuration. Hot-swap installation is fast in Click—on a 700 MHz Pentium III, installing a 50-element configuration takes less than a tenth of a second—but not fast enough to support flow creation and deletion. Most of the benefits of a flow-based design can be realized in Click as is; many configurations only require per-flow-class state and CPU scheduling, and elements can cooperate to maintain per-flow private state. Unlike flow-based systems, however, Click cannot schedule the CPU per individual flow.

Flow-based router context is appropriate for many situations where a method interface must be located, but it is not always sufficiently specific. Consider a configuration where bad packets, which occur rarely, are sent to a set of elements for error handling. Upstream elements using flow-based router context will always include the error-handling elements in their search, which might or might not be what the user wanted. Most elements adopt a simple solution: the user can override flow-based router context with a configuration argument when more specificity is required.

The Click language contains no notion of configuration variables. For example, it would be useful to refer to "the Ethernet address of the device handled by that *FromDevice* element" in a configuration string. Currently, the user must copy such an Ethernet address everywhere it is required, which leads to duplication of information. A configuration-variable extension would be simple to add.

3. AN IP ROUTER

This section presents a real Click router configuration: an IP router that forwards unicast packets in nearly full compliance with the standards [Baker 1995; Postel 1981a; Postel 1981b]. Figure 8 shows this configuration for a bridge router with two network interfaces. (The reader may want to refer to Appendix 8, a glossary of Click elements used in the IP router and elsewhere in this paper.) The rest of this section describes the IP router in more detail; Section 4 shows how to extend it by changing its scheduling and queueing behavior, and Section 6 evaluates its performance.

IP forwarding tasks that involve only local information fit naturally into the Click framework. For example, *DecIPTTL* decides if a packet's time-to-live field (TTL) has expired. If the TTL is still valid, *DecIPTTL* decrements it, incrementally updates the packet's checksum, and emits the packet on its first output; if the TTL has expired, *DecIPTTL* emits the packet on its second output (usually connected to an element that generates ICMP errors). These actions depend only on the packet's contents, and do not interact with decisions made elsewhere except as expressed in the packet's path through the element graph. Self-contained elements like *DecIPTTL* can be easily composed. For example, one could connect *DecIPTTL*'s "expired" output to a *Discard* to avoid generating ICMP errors, or insert an element that limited the rate at which errors are generated.

Some forwarding tasks require that information about a packet be calculated in one place and used in another. Click uses *annotations* to carry such information along. An annotation is a piece of information attached to the packet

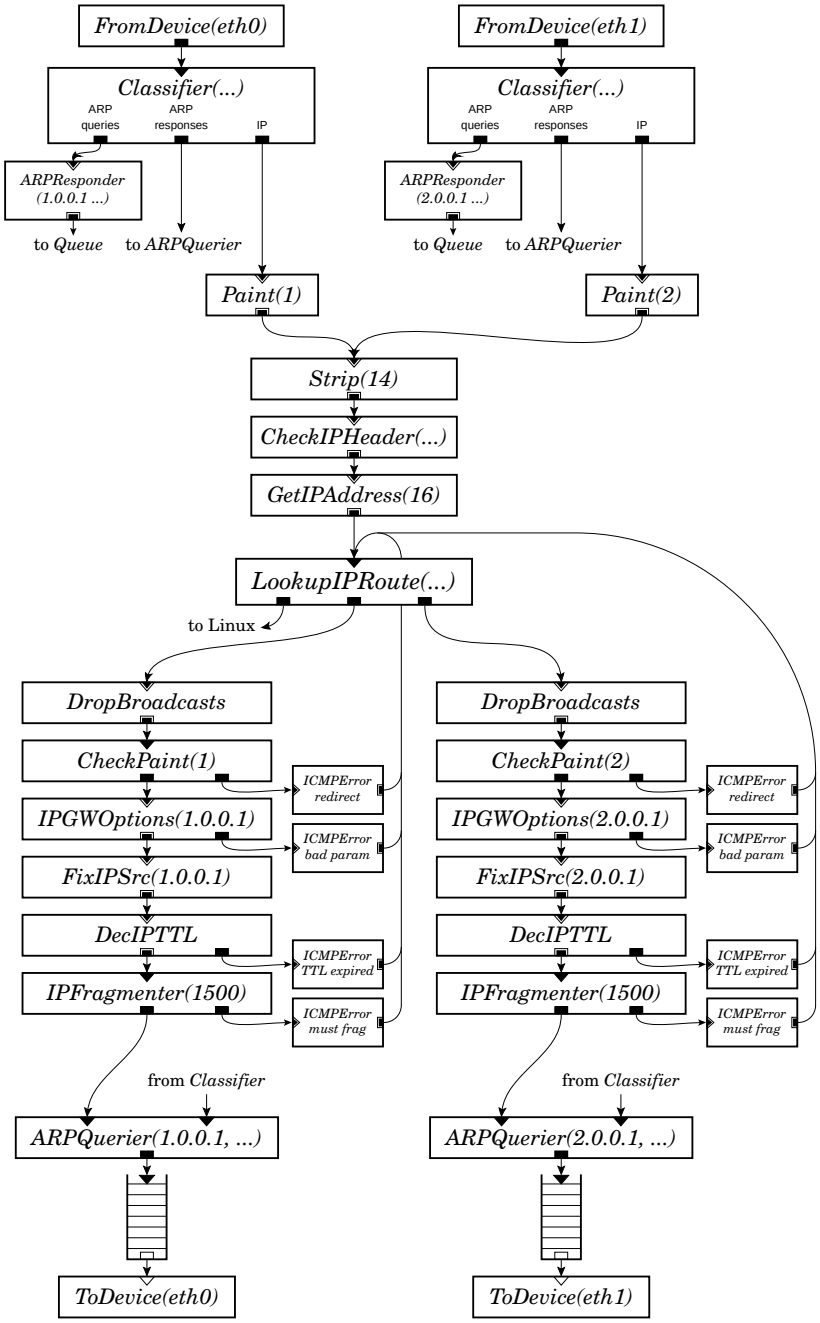


Fig. 8. An IP router configuration.

header, but not part of the packet data; the Linux packet structure, `sk_buff`, provides 48 bytes of annotation space. The annotations used in the IP router include:

—**Destination IP address.** Elements that deal with a packet's destination IP address use this annotation rather than the IP header field. This allows elements to modify the destination address—for example, to set it to the next-hop gateway address—without modifying the packet. *GetIPAddress* copies an address from the IP header into the annotation, *LookupIPRoute* replaces the annotation with the next-hop gateway's address, and *ARPQuerier* maps the annotation to the next-hop Ethernet address.

—**Paint.** The *Paint* element marks a packet with an integer "color". *CheckPaint* emits every packet on its first output, and a copy of each packet with a specified color on its second output. The IP router uses color to decide whether a packet is leaving the same interface on which it arrived, and thus should prompt an ICMP redirect.

—**Link-level broadcast flag.** *FromDevice* sets this flag on packets that arrived as link-level broadcasts. The IP router uses *DropBroadcast* to drop such packets if they are about to be forwarded to another interface.

—**ICMP Parameter Problem pointer.** This is set by *IPGWOptions* on erroneous packets to specify the bad IP header byte, and used by *ICMPError* when constructing an error message.

—**Fix IP Source flag.** The IP source address of an ICMP error packet must be the address of the interface on which the error is sent. *ICMPError* can't predict this interface, so it uses a default address and sets the Fix IP Source annotation. After the ICMP packet has been routed towards a particular interface, a *FixIPSrc* on that path will see the flag, insert the correct source address, and recompute the IP checksum.

In a few cases elements require information of an inconveniently global nature. A router usually has a separate IP address on each attached network, and each network usually has a separate IP broadcast address. All of these addresses must be known at multiple points in the Click configuration: *LookupIPRoute* must decide if a packet is destined to the router itself, *CheckIPHeader* must discard a packet with any of the IP broadcast addresses as its source address, *ICMPError* must suppress responses to IP broadcasts, and *IPGWOptions* must recognize any of the router's addresses in an IP Timestamp option. Each of these elements takes the complete list of addresses as part of its configuration string, but ideally they would derive the list automatically (perhaps using flow-based router context).

The IP router conforms to the routing standards both because of the behavior of individual elements and because of their arrangement. In terms of individual elements, the *CheckIPHeader* element checks exactly the properties required by the standards: the IP length fields, the IP source address, and the IP checksum. Also, *IPGWOptions* processes just the Record Route and Timestamp options, since the source route options should be processed only on packets addressed to the router. *IPFragmenter* fragments packets larger than the configured MTU,

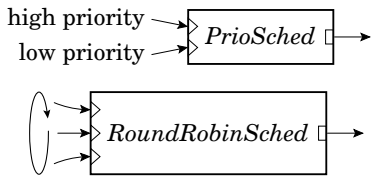


Fig. 9. Some composable packet scheduler elements.

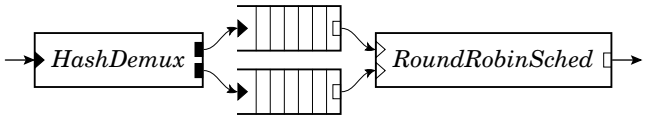


Fig. 10. A virtual queue implementing Stochastic Fairness Queueing.

but sends large, unfragmentable packets to an error output instead. Finally, *ICMPError*, which encapsulates most input packets in an ICMP error message and outputs the result, does not respond to broadcasts, ICMP errors, fragments, and source-routed packets; ICMP errors should not be generated in response to these packets. In terms of element arrangement properties, the standards mandate the placement of *DecIPTTL* after the *LookupIPRoute* routing element; a packet's time-to-live can be decremented only after it is determined that the packet is not destined for the router itself.

4. EXTENSIONS

The Click IP router is modular—it has sixteen elements on its forwarding path—and, therefore, easy to understand and easy to extend. This section demonstrates these benefits of modularity by presenting IP router extensions and other configurations built with the same elements. We show simple extensions supporting scheduling and dropping policies, queueing requirements, and Differentiated Services; an IP tunneling configuration; and one non-IP router, an Ethernet switch. The Click software supports many other extensions not described here, including RFC 2507-compatible IP header compression and decompression, IP security, communication with wireless radios, network address translation, firewalling, and other specialized routing tasks.

4.1 Scheduling

Packet scheduling is a kind of multiplexing: a scheduler decides how a number of packet sources—usually queues—will share a single output channel. A Click packet scheduler is naturally implemented as a pull element with multiple inputs and one output. This element reacts to requests for packets by choosing one of its inputs, pulling a packet from it, and returning that packet. (If the chosen input has no packets ready, the scheduler will usually try other inputs.)

We have implemented three packet scheduler elements: *RoundRobinSched*, *PrioSched*, and *StrideSched*. *RoundRobinSched* pulls from its inputs in round-robin order, returning the first packet it finds (or no packet, if no input has a packet ready). It always starts with the input cyclically following the last successful pull. *PrioSched* is a strict priority scheduler; it always tries its

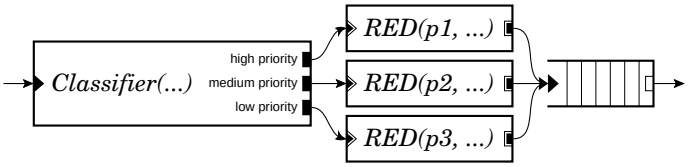


Fig. 11. Weighted RED. The three *RED* elements have different RED parameters, so packets with different priorities are dropped with different probabilities.

first input, then its second, and so forth, returning the first packet it gets. *StrideSched* implements a stride scheduling algorithm [Waldspurger and Weihl 1995] with ticket values specified by the user. Other arbitrary packet scheduling algorithms could be implemented in the same way. *RoundRobinSched* and *PrioSched* are illustrated in Figure 9.

Both *Queue*s and scheduling elements have a single pull output, so to an element downstream, *Queue*s and schedulers are indistinguishable. We can exploit this property to build *virtual queues*, compound elements that act like queues but implement more complex behavior than FIFO queueing. Figure 10 shows a virtual queue that implements a version of stochastic fairness queueing [McKenney 1990]: packets are hashed into one of several queues that are scheduled round-robin, providing some isolation between competing flows. The IP router can be extended to use stochastic fairness queueing by replacing its *Queue*s with copies of Figure 10.

4.2 Dropping Policies

The *Queue* element implements a simple dropping policy: a configurable maximum length beyond which all incoming packets are dropped. Other policies build on *Queue* rather than replacing it. For example, an independent *RED* element implements random early detection dropping [Floyd and Jacobson 1993]. *RED* contains only drop decision code; this separates the dropping policy from the packet storage policy, allowing either to be independently replaced. It also allows important policy variants through configuration rearrangement.

Random early detection is more likely to drop packets when there is network congestion; a link is considered congested when there are many packets in the queue servicing that link. The *RED* element therefore queries router queue lengths when deciding whether to drop a passing packet. Specifically, it queries the number of packets in the nearest downstream *Storage* elements, which it finds using flow-based router context. (*Storage* is a simple method interface implemented by *Queue* and other elements that store packets.) *RED* can handle one or more downstream *Storage* elements; if there are more than one, it adds their packet counts together to form a single virtual count. This generalization lets it implement variants like RED over multiple queues: a *RED* element placed before the *HashDemux* in Figure 10 would count both of the figure's *Queue*s. A different arrangement implements weighted RED [Cisco Corporation 1999], where packets are dropped with different probabilities depending on their priority; see Figure 11. Finally, *RED* can be positioned after the *Queue*s instead of before them. In this case, it is a pull element and looks upstream

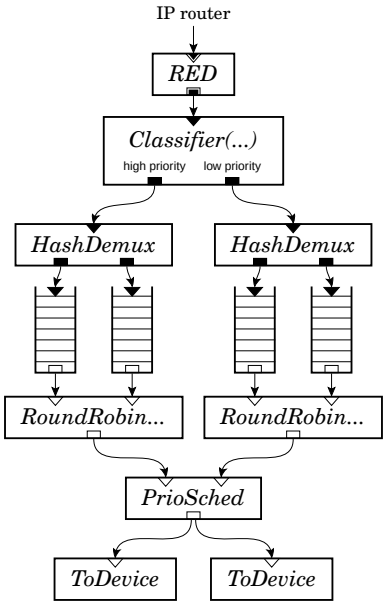


Fig. 12. A complex combination of dropping, queueing, and scheduling. The *Classifier* prioritizes input packets into two virtual queues, each of which implements stochastic fair queueing (Figure 10). *PrioSched* implements priority scheduling on the virtual queues, preferring packets from the left. The router is driving two equivalent T1 lines that pull packets from the same sources, providing a form of load balancing. Finally, *RED*, at the top, implements RED dropping over all four *Queue*s.

rather than downstream for *Storage* elements, creating a strategy like drop-from-front RED [Lakshman et al. 1996].

Simple RED dropping can be added to the IP router by adding a *RED* element before each *Queue*.

4.3 Complex Extensions and Simpler Subsets

Click is equally well suited for building simple and complex packet processors. A network administrator can easily extend the Click IP router to handle specialized routing tasks, or use a subset of its elements for some other application.

For example, imagine a complex IP router with the following requirements:

—two parallel T1 links to a backbone, between which traffic should be load-balanced;

—division of traffic into two priority levels;

—fairness among the connections within each priority level;

—RED dropping driven by the total number of packets queued.

Click's modular scheduling, queueing and dropping policy elements make this easy; Figure 12 shows the extension.

Now consider Figure 13, where a remote mobile host (43.0.0.9) would like to appear as part of its home network (1.0.0.*x*) with the help of a home node on that network (1.0.0.10). A simple Click configuration for this task can reuse several

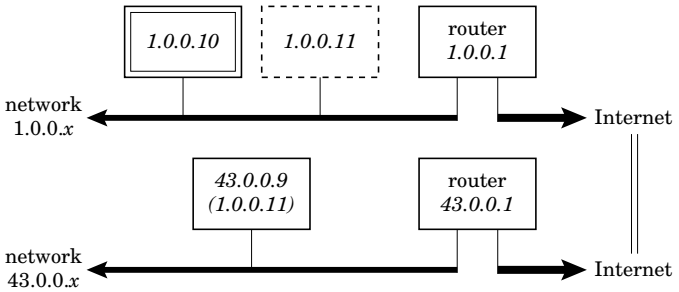


Fig. 13. Network arrangement for a configuration supporting mobility. The mobile host, 43.0.0.9, normally has the IP address 1.0.0.11. Its home node, 1.0.0.10, will encapsulate packets destined for 1.0.0.11 in packets destined for 43.0.0.9.

of the IP router's elements. Figure 14 shows the configuration active at the home node; it proxy-ARPs for the mobile host's home IP address and performs IP-in-IP encapsulation and unencapsulation to communicate with the mobile host.

4.4 Differentiated Services

The Differentiated Services architecture [Blake et al. 1998] provides mechanisms for border and core routers to jointly manage aggregate traffic streams. Diffserv border routers classify and tag packets according to traffic type and ensure that traffic enters the network no faster than allowed; core routers queue and schedule packets based on their tags. The diffserv architecture envisions flexible combinations of classification, tagging, shaping, dropping, queueing, and scheduling functions. We have implemented all of these components as Click elements, which gives router administrators full control over how they are arranged.

For example, Figure 15 implements a realistic diffserv traffic conditioning block. This configuration separates incoming traffic into 4 streams based on the IP header's Differentiated Services Code Point field (DSCP) [Nichols et al. 1998]. The first three streams are rate-limited, while the fourth represents normal best-effort delivery. The rate-limited streams are given priority over the normal stream. From top to bottom in Figure 15, these streams are (A) limited by dropping—the stream is dropped when more than 7,500 packets per second are received on average; (B) shaped—at most 10,000 packets per second are allowed through the *Shaper*, with any excess packets left enqueued; and (C) limited by reclassification—when more than 12,500 packets per second are received, the stream is reclassified as best-effort delivery and sent into the lower priority queue.

4.5 Ethernet Switch

Figure 16 shows a Click configuration unrelated to IP, namely a functional, IEEE 802.1d-compliant Ethernet switch. It acts as a learning bridge and participates with other 802.1d-compliant bridges to determine a spanning tree for the network.

The *EtherSwitch* element can be used alone as a simple, functional learn-

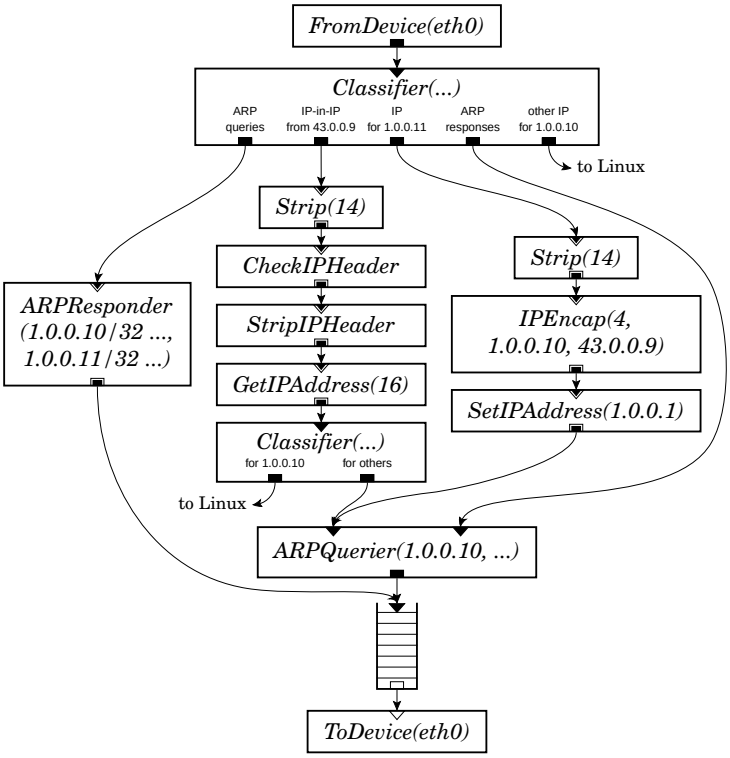


Fig. 14. A configuration for the home node, 1.0.0.10, in the arrangement of Figure 13. The home node proxy-ARPs for the mobile node (*ARPResponder*), unencapsulates packets from the remote node, sending them onto the local network (*CheckIPHeader* path), and performs IP encapsulation for packets destined for the mobile node (*IPEncap* path). Elements not shown ensure that packets generated by 1.0.0.10 itself are properly encapsulated.

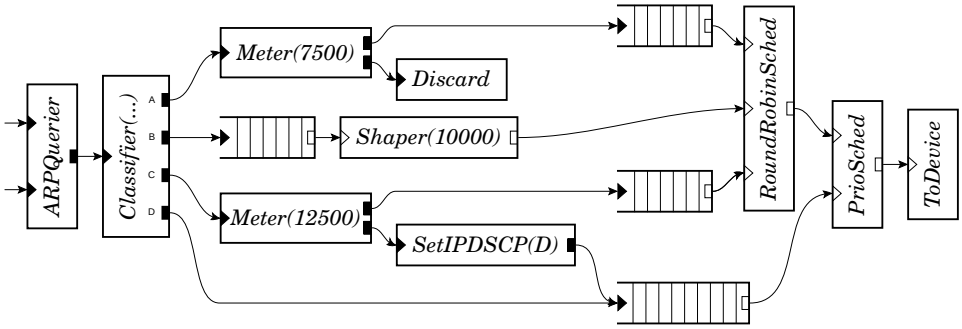


Fig. 15. A sample traffic conditioning block. *Meter*s and *Shaper*s measure traffic rates in packets per second. A, B, C, and D represent DSCP values.

ing bridge. The *EtherSpanTree* and *Suppressor* elements are necessary only to avoid cycles when multiple bridges are used in a LAN. *EtherSpanTree* implements the IEEE 802.1d protocol for constructing a network spanning tree; it works by controlling the *Suppressor* elements. *Suppressor* normally forwards

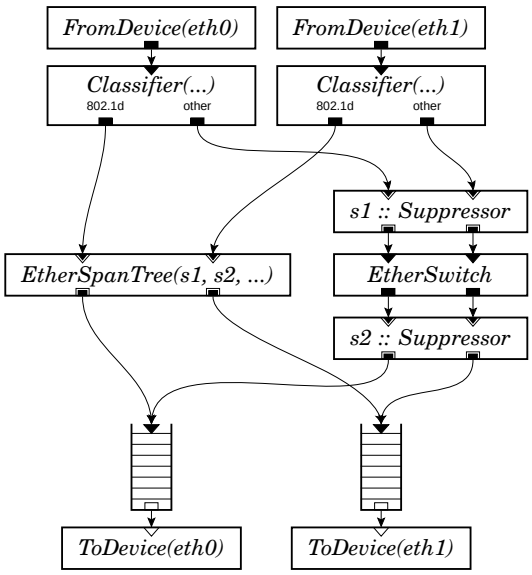


Fig. 16. An Ethernet switch configuration.

packets from each input to the corresponding output, but also exports a method interface for suppressing and unsuppressing individual ports. Packets arriving on a suppressed port are dropped. *EtherSpanTree* uses this interface to prevent *EtherSwitch* from forwarding packets along edges not on the spanning tree. The *Suppressor*s cannot be found using flow-based router context, so the user must specify them by name in *EtherSpanTree*'s configuration string.

5. KERNEL ENVIRONMENT

Click runs as a kernel thread inside a Linux 2.2 kernel. The kernel thread runs the router driver, which loops over the task queue and runs each task. Only interrupts can preempt this thread, but to keep the system responsive, the router driver voluntarily gives up the CPU to Linux from time to time. (The user can configure how often with a *ScheduleLinux* element.) There is generally only one driver thread active at a time. However, during configuration installation, there may be two or more simultaneously active router configurations and driver threads—the new router plus several old ones in the process of dying.

The Click kernel module uses Linux's `/proc` filesystem to communicate with user processes. To bring a router on line, the user creates a configuration description in the Click language and writes it to `/proc/click/config` (or `/proc/click/hotconfig` for a hot-swap installation). Other files in `/proc/click` export information about the currently installed configuration and memory statistics. When installing a router configuration, Click creates a subdirectory under `/proc/click` for each element. This subdirectory contains that element's handlers (Section 2.6). The code that handles accesses to `/proc` runs outside the Click driver thread, in the context of the reading or writing process.

A running Click router contains four important object categories: elements,

a router, packets, and timers.

—**Elements.** The system contains an element object for each element in the current configuration, as well as prototype objects for every kind of primitive element that could be used.

—**Router.** The single router object collects information relevant to a given router configuration. It configures the elements, checks that connections are valid, and puts the router on line; it also manages the task queue.

—**Packets.** Click packet data is stored in a single block of memory, as opposed to a BSD `mbuf`-like structure. Packet data is copy-on-write—when copying a packet, the system copies the packet header but not the data. Annotations are stored in the packet header in a fixed static order; there is currently no way to dynamically add a new kind of annotation. In the Linux kernel, Click packet objects are equivalent to `sk_buffs` (Linux's packet abstraction).

—**Timers.** In the Linux kernel, Click timers are implemented with Linux timer queues, which on Intel PCs have .01-second resolution.

5.1 Polling and Device Handling

The original Click system [Morris et al. 1999] shared Linux's interrupt structure and device handling; our goal was to change Linux as little as possible. However, interrupt overhead and device handling dominated that system's performance, consuming about 4.8 and 6.7 μs respectively of the 13 μs required to forward a packet on a 700 MHz Pentium III PC. In addition, Click processed packets at a lower priority than interrupts, leading to receive livelock [Mogul and Ramakrishnan 1997]: with increasing numbers of input packets, interrupt processing eventually starved all other system tasks, leading to reduced throughput.

Click now eliminates interrupts in favor of polling. Device-handling elements —namely, *FromDevice* and *ToDevice*—place themselves on Click's task queue. When activated, *FromDevice* polls its device's receive DMA queue for newly arrived packets; if any are found, it pushes them through the configuration. *ToDevice* examines its device's transmit DMA queue for empty slots, which it fills by pulling packets from its input. These elements also refill the receive DMA list with empty buffers and remove transmitted buffers from the transmit DMA list. The device never interrupts the processor; furthermore, Linux networking code never executes, except as requested by the Click router configuration. Mogul and Ramakrishnan [1997] also used polling to eliminate receive livelock. However, their system left interrupts enabled under light load, while Click is a pure polling system—even infrequent PC interrupts are simply too expensive.

Polling required changes to Linux's structure representing network devices and the drivers for the network devices we used. The new device structure includes, for example, methods to turn interrupts on and off, to poll for received packets, and to clean the transmit DMA ring.

The introduction of polling let us eliminate all programmed I/O (PIO) interaction with the Ethernet controllers. One PIO reenabled receive interrupts after an interrupt was processed, and was not necessary in a polling system.

The second PIO was used in the send routine to tell the controller to look for a new packet to transmit. To eliminate it, we configured the network card to periodically check the transmit DMA queue for new packets. As a result, all communication between Click and the controller takes place indirectly through the shared DMA queues in main memory, in the style of the Osiris network controller [Druschel et al. 1994]. This saved roughly two microseconds per packet, as a PIO takes about 1 μs to execute.

These improvements eliminated receive livelock, reduced the total time required to process a packet from 13 μs to 2.8 μs, and increased Click's maximum loss-free forwarding rate of minimum-sized packets more than fourfold. Mogul and Ramakrishnan [1997] found that polling eliminated receive livelock but did not increase peak forwarding rate. We hypothesize that interrupts are relatively much more expensive on PCs than on the Alpha hardware they used.

6. EVALUATION

This section evaluates Click's performance for IP routing and for several extended configurations. We also describe and analyze sources of overhead in Click's design and implementation.

6.1 Experimental Setup

The experimental setup consists of a total of nine Intel PCs running Linux—specifically, a version of Linux 2.2.14 modified to add support for Click's polling device drivers. Of the nine PCs, one is the router host, four are source hosts, and four are destination hosts. The router host has eight 100 Mbit/s Ethernet controllers connected, by point-to-point links, to the source and destination hosts. During a test, each source generates an even flow of UDP packets addressed to a corresponding destination; the router is expected to get them there. The tests use version 1.1 of the Click software.

The router host has a 700 MHz Intel Pentium III CPU and an Intel L440GX+ motherboard. Its Ethernet controllers are DEC 21140 Tulip 100 Mbit/s PCI controllers [Digital Equipment Corporation 1998] on multi-port cards, split across the motherboard's two independent PCI buses. The Pentium III has a 16 KB L1 instruction cache, a 16 KB L1 data cache, and a 256 KB L2 unified cache. The source and destination hosts have 733 MHz Pentium III CPUs and 200 MHz Pentium Pro CPUs, respectively. Each has one DEC 21140 Ethernet controller. The source-to-router and router-to-destination links are point-to-point full-duplex 100 Mbit/s Ethernet.

The source hosts generate UDP packets at specified rates, and can generate up to 147,900 64-byte packets per second. The destination hosts count and discard the forwarded UDP packets. Each 64-byte UDP packet includes Ethernet, IP, and UDP headers as well as 14 bytes of data and the 4-byte Ethernet CRC. When the 64-bit preamble and 96-bit inter-frame gap are added, a 100 Mbit/s Ethernet link can carry up to 148,800 such packets per second.

6.2 Forwarding Rates

We characterize performance by measuring the rate at which a router can forward 64-byte packets over a range of input rates. Minimum-size packets stress

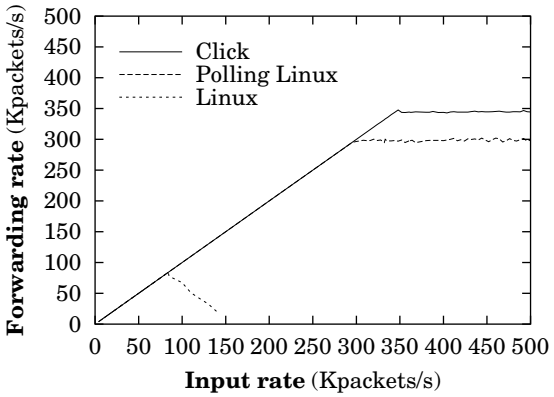


Fig. 17. Forwarding rate as a function of input rate for Click and Linux IP routers (64-byte packets).

the router harder than larger packets; the CPU and several other bottleneck resources are consumed in proportion to the number of packets forwarded, not in proportion to bandwidth. Plotting forwarding rate versus input rate indicates both the maximum loss-free forwarding rate (MLFFR) and the router's behavior under overload. An ideal router would emit every input packet regardless of input rate, corresponding to the line $y = x$.

Figure 17 compares Click's and Linux's performance for IP routing. The line marked "Click" shows the performance of an eight-interface version of the Click IP router configuration in Figure 8. This configuration has a total of 161 elements (19 elements per interface plus 9 elements shared by all interfaces). The "Linux" and "Polling Linux" lines show the performance of Linux IP routing; "Linux" uses the standard interrupting device drivers, while "Polling Linux" was modified to use Click's polling drivers.

Click's maximum loss-free forwarding rate is 333,000 packets per second; its peak forwarding rate, attainable with bursty senders, is 360,000 packets per second. When these rates are exceeded, the Ethernet controllers report that they are discarding received packets due to insufficient receive DMA descriptors, which suggests that the bottleneck is the router's CPU, not its PCI bus bandwidth.

Standard Linux's MLFFR is 84,000 packets per second; its output rate declines as the input rate increases due to receive livelock. Polling Linux's MLFFR of 284,000 packets per second is over three times standard Linux's MLFFR, which demonstrates the advantages of avoiding interrupts and handling device I/O efficiently. Polling Linux is slightly slower than Click, perhaps because Linux's routing table lookup algorithms are slower, but more scalable.

An otherwise idle Click IP router forwards 64-byte packets with a one-way latency of 29 μs. This number was calculated by taking the round-trip ping time through the router (measured with `tcpdump`), subtracting the round-trip ping time with the router replaced by a wire, and dividing by two. 5.8 μs of the 29 are due to the time required to transmit a 64-byte Ethernet packet at 100 megabits per second. 10 μs are due to the costs of polling eight Ethernet controllers, all

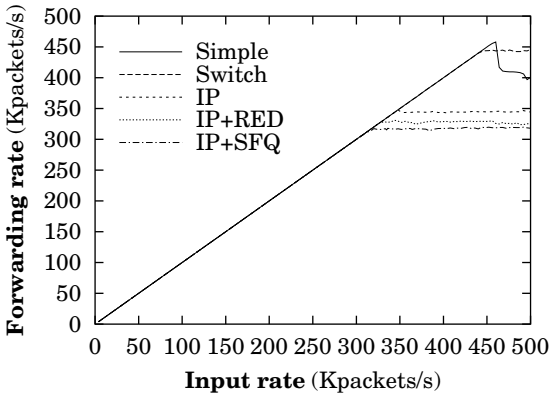


Fig. 18. Forwarding rate as a function of input rate for extended IP router configurations and some non-IP routers (64-byte packets).

but two of which are idle in this test. 2.5 μs of delay is the expected amount of time the Tulip controller will take to poll for a new packet to transmit. The latency of a router running standard Linux IP code is 33 μs.

Figure 18 shows the performance of some extended IP router configurations and some non-IP-router configurations. The "Simple" line shows the performance of the simplest possible Click router, which forwards input packets directly from input to output with no intervening processing. The MLFFR for this configuration is 452,000 packets per second. It is limited by the PCI bus, not the CPU, which explains its different behavior under overload. The "Switch" line corresponds to the Ethernet switch configuration of Figure 16; its MLFFR is close to that of "Simple". The "IP" line, repeated from Figure 17, shows the performance of the base IP router configuration. The "IP+RED" line corresponds to an IP router in which a *RED* element is inserted before each *Queue*. No packets were dropped by RED in the test, since the router's output links are as fast as its inputs. The "IP+SFQ" line shows the performance of an IP router with each *Queue* replaced by a stochastic fair queue—namely, a version of Figure 10 with four component *Queue*s. As router complexity increases from simple forwarding through IP routing to IP routing with stochastic fair queueing, Click's performance drops only gradually.

This test configuration shows both Click and Linux in a better light than might be seen in a real network. The tests did not involve fragmentation, IP options, or ICMP errors, though Figure 8 has all the code needed to handle these. Increasing the number of hosts involved might slow Click down by increasing the number of ARP table entries. Increasing the routing table size would also decrease performance, a problem addressed by previous work on fast lookup in large tables [Degermark et al. 1997; Waldvogel et al. 1997]. The configuration is nevertheless sufficient to analyze the architectural aspects of Click's performance.

Table I. Microbenchmarks of Tasks Involved in the Click IP Forwarding Path

Task	**Time** (ns/packet)
Polling packet	528
Refill receive DMA ring	90
Push through Click forwarding path	1565
Pull from Click queue	103
Enqueue packet for transmit	161
Clean transmit DMA ring	351
Total	**2798**

6.3 Analysis

Table I breaks down the cost of forwarding a packet through a Click router. Costs were measured in nanoseconds by Pentium III cycle counters [Intel Corporation 1996]; each cost is the accumulated cost for all packets in a 10-second run divided by the number of packets forwarded. "Polling packet" is the time *FromDevice* spends taking a packet from the Tulip's receive DMA ring. "Refill receive DMA ring" is the time *FromDevice* spends replacing the DMA descriptor of the packet it just received with a new descriptor, so that the Tulip may receive a new packet. The Click IP forwarding path (see Figure 8) is divided into a push path and a pull path. The push path involves 15 elements; it begins when an input packet is emitted by some *FromDevice* element and ends when the packet reaches the *Queue* element before the appropriate transmitting device. The pull path involves just two elements—when a *ToDevice* element is ready to send, it pulls a packet from a *Queue*. The time consumed by Linux's equivalent of the push path is 1.65 μs, slightly more than Click's 1.57 μs. "Enqueue packet for transmit" is the time *ToDevice* spends enqueueing a packet onto the Tulip's transmit DMA ring. "Clean transmit DMA ring" is the time *ToDevice* spends removing DMA descriptors of transmitted packets. Overall, Click code takes 60% of the time required to process a packet; device code takes the other 40%.

Table II breaks down the 1.57 μs push cost by element. Each element's cost is the difference between the Pentium III cycle counter value before and after executing the element, decreased by the time required to read the cycle counter; it includes the virtual function call(s) that move a packet from one element to the next. (*IPFragmenter* is so inexpensive because it does nothing—no packets need to be fragmented in this test—and its packet handoff requires one virtual function call. Other elements that do nothing in this test, such as *IPGWOptions*, have agnostic ports that incur two virtual function calls per packet handoff.) The *FromDevice* and *Queue* elements are not included.

The total cost of 2798 ns measured with performance counters implies a maximum forwarding rate of about 357,000 packets per second, consistent with the observed peak forwarding rate of 360,000 packets per second.

Forwarding a packet through Click incurs five L2 data cache misses (measured using Pentium III performance counters): one to read the receive DMA descriptor, two to read the packet's Ethernet and IP headers, and two to remove the packet from the transmit DMA queue and put it into a pool of reusable packets. Each of these loads takes about 112 nanoseconds. Click runs without

Table II. Execution Times of Elements Involved in IP Forwarding, in Nanoseconds per Packet

Element	**Time** (ns)
Classifier	70
Paint	77
Strip	67
CheckIPHeader	457
GetIPAddress	120
LookupIPRoute	140
DropBroadcasts	77
CheckPaint	67
IPGWOptions	63
FixIPSrc	63
DecIPTTL	119
IPFragmenter	29
ARPQuerier	106
Subtotal	**1455**

incurring any other L2 data or instruction cache misses. 1295 instructions are retired during the forwarding of a packet, implying that significantly more complex Click configurations could be supported without exhausting the Pentium III's 16K L1 instruction cache.

6.4 Overhead of Modularity

Click's modularity imposes performance costs in two ways: the overhead of passing packets between elements, and the overhead of unnecessarily general element code.

Passing a packet from one element to the next causes either one or two virtual function calls, where a virtual function call includes loading the relevant function pointer from a virtual function table as well as an indirect jump through that function pointer. Indirect jumps take about 10 nanoseconds if the Pentium III's branch target buffer successfully predicts the target address, but can take dozens of nanoseconds if it fails. Most elements in the IP router use the simplified agnostic-port interface that causes two virtual function calls per packet handoff; the CPU usually predicts one call but doesn't predict the other. As a result, the total cost of packet handoff is about 70 nanoseconds. Thus, adding an empty element to a configuration graph usually adds 70 nanoseconds of overhead, and the IP router, with 16 elements on its forwarding path, has about 1 μs of overhead. This overhead is avoidable—we have built tools that can automatically eliminate all virtual function calls from a Click configuration [Kohler 2000].

Several elements in the IP router configuration were implemented with more generality than the IP router requires. The best example is *Classifier*, which the IP router uses to classify Ethernet packets into ARP queries, ARP responses, and IP packets. However, *Classifier* can handle many classification tasks; at run time, it interprets a small data structure (derived from its configuration string) that tells it which packet data to examine. An element specialized for the IP router's classification task could use straight-line code with compiled-in

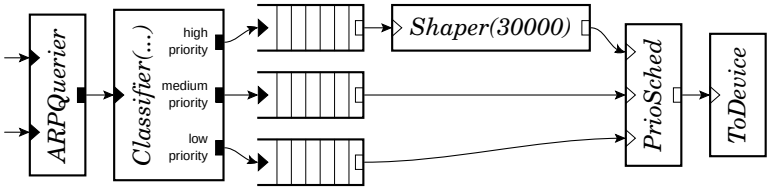


Fig. 19. Another Click diffserv traffic conditioning block. This configuration fragment implements strict priority with three priority levels; in addition, the high-priority stream is rate-limited to 30,000 packets per second.

constants, avoiding interpretation overhead. We measured this overhead by writing such a specialized classifier. This element cost 24% less CPU time per packet than the generic *Classifier*, but even the generic *Classifier* takes only 4% of the per-packet CPU time expended inside the Click configuration. Since the rest of the IP router's elements offer less opportunity for specialization, we conclude that element generality has a relatively small effect on Click's performance.

To analyze the combined effects of these overheads, we wrote two single elements that combine much of the processing in Figure 8. These elements reduce the number of virtual function calls on the forwarding path, specialize some of the more general elements, and offer the compiler better opportunities for optimization. One element implements the functionality of *Paint*, *Strip*, *CheckIPHeader*, and *GetIPAddress*; the other, the functionality of *DropBroadcasts*, *CheckPaint*, *IPGWOptions*, *FixIPSrc*, *DecIPTTL*, and *IPFragmenter*. The resulting configuration is equivalent to Figure 8, but has only eight elements on the forwarding path instead of sixteen. The new configuration's push path processes an IP packet in 1.03 μs instead of 1.57 μs. When we add eight distinct do-nothing elements to the forwarding path of this new configuration, restoring the number of elements to sixteen, the packet processing time rises to 1.58 μs. This suggests that the entire reduction from 1.57 μs to 1.03 μs is due to reducing the number of virtual function calls.

6.5 Differentiated Services Evaluation

Section 4.4 showed that Click can conveniently model Differentiated Services configurations; this section shows that Click can enforce diffserv policies using only packet scheduling elements like *PrioSched*. The test configuration involves three source hosts sending packets marked as high, medium, and low priority respectively. The router has a single 100 Mbit/s Ethernet output link. It uses the configuration in Figure 8 with each *Queue* replaced with Figure 19. The configuration queues each priority level separately; it also rate-limits the highest priority traffic to no more than 30,000 packets per second. The *PrioSched* packet scheduling element implicitly schedules CPU time as well as link bandwidth.

Figure 20 shows the results. The per-source sending rate varies along the x axis; the total input load on the router is three times the x-axis value. The y axis indicates how many packets per second the router forwarded from each of the sources. Once the per-source input rate rises above 30,000 packets per

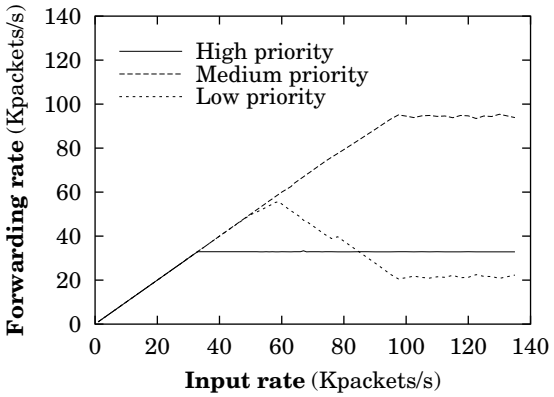


Fig. 20. Performance of the Click diffserv configuration in Figure 19. Three traffic sources send 64-byte UDP packets with three different priorities into a router with a single 100 Mbit/s output link. The sources send equal numbers of high, medium, and low-priority packets; the x axis indicates the input rate for each priority level.

second, the router starts dropping high-priority packets, since they are rate-limited. Above 57,000 packets per second, the output link is saturated, and medium priority packets start taking bandwidth away from low. Above 100,000 packets per second, the router runs out of CPU time, forcing the Ethernet controllers to drop packets. Since the controllers don't understand priority, they drop medium-priority packets as well as low. Each medium-priority packet that a controller drops leaves room on the output link for a low-priority packet, which is why the low-priority forwarding rate isn't driven down to zero.

There is probably no way to achieve perfect packet scheduling under overload; instead, a router should be engineered with sufficient processing power to carry all packets to the point where the priority decision is made. The Click user can approximate such a design by moving the classification and per-priority queues of Figure 19 earlier in the configuration—that is, nearer to *FromDevice* than *ToDevice*. This wastes less CPU time on lower priority packets that will eventually be dropped, because they travel through less of the configuration before reaching the drop point. Even with the priority decision late in the configuration, Click enforces priority reasonably well under overload and very well with ordinary loads.

7. RELATED WORK

The x-kernel [Hutchinson and Peterson 1991] is a framework for implementing and composing network protocols. It is intended for use at end nodes, where packet motion is vertical (between the network and user level) rather than horizontal (between network interfaces). Like a Click router, an x-kernel configuration is a graph of processing nodes, and packets are passed between nodes by virtual function calls. However, an x-kernel configuration graph is always acyclic and layered, as x-kernel nodes were intended to represent protocols in a protocol stack. This prevents cyclic configurations like our IP router. The x-kernel's inter-node communication protocols are more complex than Click's.

Connections between nodes are bidirectional—packets travel up the graph to user level and down the graph to the network. Packets pass alternately through "protocol" nodes and "session" nodes, where the session nodes correspond to end-to-end network connections like TCP sessions. Bidirectional connections and session nodes are irrelevant to most routers.

Scout [Mosberger and Peterson 1996; Peterson et al. 1999] was designed for use as an operating system at end nodes, but it is better suited for routing than the *x*-kernel; for example, cyclic configurations are partially supported, and session nodes are not mandatory. Execution in Scout is centered on "paths", sequences of packet processing functions. Each path has implicit queues on its inputs and outputs; it is not clear, therefore, how many queues would appear in a complex configuration like the IP router, which is not amenable to linearization. Each path is run by a single thread and has a CPU priority, and packets are classified into the correct path as early as possible. For example, TCP/IP packets containing MPEG data can be routed immediately to the correct path, whose scheduling behavior might be specialized for MPEG. If desired, Click primitives can support some uses of paths: an early *Classifier* element could send MPEG-in-TCP-in-IP-in-Ethernet traffic into a special *Queue*, and the element that took packets off that *Queue* could be scheduled appropriately for MPEG data. However, Scout supports per-flow paths, where a path object is created for each individual network flow. This allows its CPU scheduler to treat flows independently; Click's CPU scheduler can only treat flow classes independently.

The UNIX System V STREAMS system [Ritchie 1984] is also built from composable packet processing modules. STREAMS modules include implicit queueing by default. Each module must be prepared for the next module's queue to fill up, and to respond by queueing or discarding or deferring the processing of incoming packets. Modules with multiple inputs or outputs must also make packet scheduling decisions. STREAMS' tendency to spread scheduling and queueing logic throughout the configuration conflicts with a router's need for precise control over these functions.

FreeBSD contains a modular networking system called Netgraph [Elischer and Cobbs 1998] whose nodes strongly resemble Click elements. Netgraph was designed for composing coarse-grained modules such as PPP encapsulation and other framing protocols. Configurations are built up dynamically; there are commands that add and delete nodes, and that connect and disconnect "hooks" (the equivalent of ports), but no external specification for a configuration. This supports on-line configuration modification somewhat more naturally than Click, but there is no way to analyze a Netgraph configuration off line or install a new configuration atomically.

ALTQ [Cho 1998], a system for configurable traffic management, is also shipped with FreeBSD. It contains several sophisticated queueing policies, but its configurability is limited to the specification of one queueing policy per output interface.

The router plugins system [Decasper et al. 1998; Decasper 1999] was designed to make IP routers more extensible. A router plugin is a software module executed when a classifier matches a particular flow. These classifiers can

be installed at any of several "gates". However, gates are fixed points in the IP forwarding path; there is no way to add new gates or to control the path itself.

To the best of our knowledge, commercial routers are difficult to extend, either because they use specialized hardware [Newman et al. 1998; Partridge et al. 1998] or because their software is proprietary. However, open router software does not necessarily make a router easily extensible. A network administrator could, in principle, implement new routing functions in Linux, but we expect few administrators have the time or ability to modify the monolithic Linux kernel. Click's modularity makes it easy to take advantage of its extensibility.

The active networking research program allows anyone to write code that will affect a router [Tennenhouse et al. 1997; Smith et al. 1999]. Click allows a trusted user to change any aspect of a router; active networking allows untrusted packets to decide how they should be routed. The two approaches are complementary, and Click may provide a good platform for active network research.

Click's polling implementation and device driver improvements were derived from previous work [Druschel et al. 1994; Mills 1988; Mogul and Ramakrishnan 1997; Wroclawski 1997].

8. CONCLUSION

Click is an open, extensible, and configurable router framework. The Click IP router demonstrates that real routers can be built from small, modular elements, and our performance analysis shows that modularity is compatible with good forwarding performance for PC hardware. IP router extensions for scheduling and dropping policies, complex queueing, and Differentiated Services simply require adding and rearranging a couple elements. Finally, Click is flexible enough to support other packet processing applications. The Click system is free software; it is available for download at `http://www.pdos.lcs.mit.edu/click/`.

APPENDIX

ELEMENT GLOSSARY

This section lists 38 of the approximately 130 elements included with version 1.1 of the Click software. Each entry follows this format:

ElementName(configuration arguments) Push, pull, or agnostic (specifies port types). Port descriptions (packet types and numbers of ports). Description.

More detailed descriptions are accessible on line at `http://www.pdos.lcs.mit.edu/click/doc/`.

ARPQuerier(...) Push. First input takes IP packets, second input takes ARP responses with Ethernet headers. Output emits ARP queries and IP-in-Ethernet packets. Uses ARP to find the Ethernet address corresponding to each input IP packet's destination IP address annotation; encapsulates the packet in an Ethernet header with that destination Ethernet address.

ARPResponder(ip eth, ...) Agnostic. Input takes ARP queries, output emits ARP responses. Responds to ARP queries for IP address *ip* with the static Ethernet address *eth*.

CheckIPHeader(...) Agnostic. Input takes IP packets. Discards packets with invalid IP length, source address, or checksum fields; forwards valid packets unchanged.

CheckPaint(p) Agnostic. Input takes any packet. Forwards packets with paint annotation *p* to both outputs; otherwise just to first.

Classifier(...) Push. Input takes any packet. Examines packet data according to a set of classifiers, one classifier per output port. Forwards packet to output port corresponding to the first classifier that matched. Example classifier: "*12/0800*" checks that the packet's data bytes 12 and 13 contain values 8 and 0, respectively. See also *IPClassifier*.

DecIPTTL Agnostic. Input takes IP packets. Decrements input packets' IP time-to-live field. If the packet is still live, incrementally updates IP checksum and sends modified packet to first output; if it has expired, sends unmodified packet to second output.

Discard Push. Discards all input packets.

DropBroadcasts Agnostic. Input takes any packet. Discards packets that arrived as link-level broadcasts; forwards others unchanged.

EtherSpanTree Agnostic. Inputs take IEEE 802.1d messages. Implements the IEEE 802.1d spanning tree algorithm for Ethernet switches.

EtherSwitch Push. Inputs take Ethernet packets; one input per output. Learning, forwarding Ethernet switch. Learns the interfaces corresponding to Ethernet addresses by examining input packets' source addresses; forwards packets to the correct output port, or broadcasts them if the destination Ethernet address is not yet known.

FixIPSrc(ip) Agnostic. Input takes IP packets. Sets the IP header's source address field to the static IP address *ip* if the packet's Fix IP Source annotation is set; forwards other packets unchanged.

FromDevice(devicename) Push. No inputs. Sends packets to its single output as they arrive from a Linux device driver.

FromLinux(devicename, ip/netmask) Push. No inputs. Installs into Linux a fake Ethernet device *devicename* and a routing table entry that sends packets for *ip/netmask* to that fake device. The result is that packets generated at the router host and destined for *ip/netmask* are emitted on *FromLinux*'s single output as they arrive from Linux.

GetIPAddress(16) Agnostic. Input takes IP packets. Copies the IP header's destination address field (offset 16 in the IP header) into the destination IP address annotation; forwards packets unchanged.

HashDemux(offset, length) Push. Input takes any packet; arbitrary number of outputs. Forwards packet to one of its outputs, chosen by a hash bytes $[offset, offset + length)$ of the packet's data.

ICMPError(ip, type, code) Agnostic. Input takes IP packets, output emits ICMP error packets. Encapsulates first part of input packet in ICMP error

header with source address *ip*, error type *type*, and error code *code*. Sets the Fix IP Source annotation on output packets.

IPClassifier(...) Push. Input takes IP packets. Examines packet data according to a set of classifiers, one classifier per output port. Forwards packet to output port corresponding to the first classifier that matched. Example classifier: "*ip src 1.0.0.1 and dst tcp port www*" checks that the packet's source IP address is 1.0.0.1, its IP protocol is 6 (TCP), and its destination TCP port is 80. See also *Classifier*.

IPEncap(p, ipsrc, ipdst) Agnostic. Input takes any packet, output emits IP packets. Encapsulates input packets in an IP header with protocol *p*, source address *ipsrc*, and destination address *ipdst*.

IPFragmenter(mtu) Push. Input takes IP packets. Fragments IP packets larger than *mtu*; sends fragments, and packets smaller than *mtu*, to first output. Too-large packets with the don't-fragment IP header flag set are sent unmodified to the second output.

IPGWOptions Agnostic. Input takes IP packets. Processes IP Record Route and Timestamp options; packets with invalid options are sent to the second output.

LookupIPRoute(table) Push. Input takes IP packets with valid destination IP address annotations. Arbitrary number of outputs. Looks up input packets' destination annotations in a static routing table specified in the configuration string. Forwards each packet to the output port specified in the resulting routing table entry; sets its destination annotation to the resulting gateway address, if any.

Meter(r) Push. Input takes any packet. Sends packets to first output if recent input rate averages $< r$, second output otherwise. Multiple rate arguments are allowed.

Paint(p) Agnostic. Input takes any packet. Sets each input packet's paint annotation to *p* before forwarding it.

PrioSched Pull. Inputs take any packet; one output port, arbitrary number of input ports. Responds to pull requests by trying inputs in numeric order, returning the first packet it receives. Thus, lower-numbered inputs have priority.

Queue(n) Push input, pull output. Input takes any packet. Stores packets in a FIFO queue; maximum queue capacity is *n*.

RED(min-thresh, max-thresh, max-p) Agnostic. Input takes any packet. Drops packets probabilistically according to the Random Early Detection algorithm [Floyd and Jacobson 1993] with the given parameters; forwards other packets unchanged. Examines nearby router queue lengths when making its drop decision.

RoundRobinSched Pull. Inputs take any packet; one output port, arbitrary number of input ports. Responds to pull requests by trying inputs in round-robin order; returns the first packet it receives. It first tries the input port just after the one that succeeded on the last pull request.

ScheduleInfo(elementname param, ...) No inputs or outputs. Specifies elements' initial task queue scheduling parameters by element name. Each scheduling parameter is a real number; an element with parameter $2p$ will be scheduled twice as often as an element with parameter p.

ScheduleLinux No inputs or outputs. Places itself on the task queue, and returns to Linux's scheduler every time it is run.

SetIPAddress(ip) Agnostic. Input takes IP packets. Sets each packet's destination IP address annotation to the static IP address *ip*. Forwards modified packets to first output.

SetIPDSCP(c) Agnostic. Input takes IP packets. Sets each packet's IP diffserv code point field [Nichols et al. 1998] to *c* and incrementally updates the IP header checksum. Forwards modified packets to first output.

Shaper(n) Pull. Input takes any packet. Simple pull traffic shaper: forwards at most *n* pull requests per second to its input. Pull requests over that rate get a null pointer in response.

Strip(n) Agnostic. Input takes any packet. Strips off each packet's first *n* bytes; forwards modified packets to first output.

StripIPHeader Agnostic. Input takes IP packets. Strips off each packet's IP header, including any options; forwards modified packets to first output.

Suppressor Agnostic. Inputs take any packet; arbitrary number of input ports, same number of output ports. Normally forwards packets arriving on each input port to the corresponding output port. A method interface allows other elements to ask *Suppressor* to drop packets arriving on particular input ports.

Tee(n) Push. Input takes any packet; *n* output ports. Forwards each input packet to all *n* output ports.

ToDevice(device) Pull. Input takes Ethernet packets; no outputs. Hands packets to a Linux device driver for transmission. Activates pull requests only when the device is ready.

ToLinux Push. Input takes Ethernet packets; Linux will ignore the Ethernet header except for the protocol field. No outputs. Hands input packets to Linux's default network input software.

ACKNOWLEDGMENTS

Massimiliano Poletto helped us enormously with tests and benchmarks. We also thank Alex Snoeren for his work on the IPsec elements, Chuck Blake for help with hardware, and Hari Balakrishnan, Daniel Jackson, Greg Minshall, John Wroclawski, Chandu Thekkath, and the anonymous reviewers for their helpful comments.

REFERENCES

BAKER, F. 1995. Requirements for IP Version 4 routers. RFC 1812 (June), Internet Engineering Task Force. `ftp://ftp.ietf.org/rfc/rfc1812.txt`.

BLAKE, S., BLACK, D., CARLSON, M., DAVIES, E., WANG, Z., AND WEISS, W. 1998. An architecture for differentiated services. RFC 2475 (Dec.), Internet Engineering Task Force. `ftp://ftp.ietf.org/rfc/rfc2475.txt`.

CHO, K. 1998. A framework for alternate queueing: towards traffic management by PC-UNIX based routers. In *Proc. USENIX 1998 Annual Technical Conference* (June 1998), pp. 247–258.

CISCO CORPORATION. 1999. Distributed WRED. Technical report. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/cc111/wred.htm`, as of January 2000.

CLARK, D. 1985. The structuring of systems using upcalls. In *Proc. of the 10th ACM Symposium on Operating Systems Principles (SOSP)* (Dec. 1985), pp. 171–180.

DECASPER, D., DITTIA, Z., PARULKAR, G., AND PLATTNER, B. 1998. Router plugins: A software architecture for next generation routers. In *Proc. ACM SIGCOMM Conference (SIGCOMM '98)* (Oct. 1998), pp. 229–240.

DECASPER, D. S. 1999. *A software architecture for next generation routers*. Ph. D. thesis, Swiss Federal Institute of Technology, Zurich.

DEGERMARK, M., BRODNIK, A., CARLSSON, S., AND PINK, S. 1997. Small forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM Conference (SIGCOMM '97)* (Oct. 1997), pp. 3–14.

DIGITAL EQUIPMENT CORPORATION. 1998. DIGITAL Semiconductor 21140A PCI Fast Ethernet LAN Controller Hardware Reference Manual. `http://developer.intel.com/design/network/manuals`.

DRUSCHEL, P., PETERSON, L., AND DAVIE, B. 1994. Experiences with a high-speed network adaptor: A software perspective. In *Proc. ACM SIGCOMM Conference (SIGCOMM '94)* (Aug. 1994), pp. 2–13.

ELISCHER, J. AND COBBS, A. 1998. The Netgraph networking system. Technical report (Jan.), Whistle Communications. `http://www.elischer.com/netgraph/`, as of July 2000.

FLOYD, S. AND JACOBSON, V. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Networking 1*, 4 (Aug.), 397–413.

HUTCHINSON, N. C. AND PETERSON, L. L. 1991. The *x*-kernel: an architecture for implementing network protocols. *IEEE Trans. Software Engineering 17*, 1 (Jan.), 64–76.

INTEL CORPORATION. 1996. Pentium Pro Family Developer's Manual, Volume 3. `http://developer.intel.com/design/pro/manuals`.

KOHLER, E. 2000. *The Click modular router*. Ph. D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology.

LAKSHMAN, T. V., NEIDHARDT, A., AND OTT, T. J. 1996. The drop from front strategy in TCP and in TCP over ATM. In *Proc. IEEE Infocom*, Volume 3 (March 1996), pp. 1242–1250.

MCCANNE, S. AND JACOBSON, V. 1993. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. Winter 1993 USENIX Conference* (Jan. 1993), pp. 259–269.

MCKENNEY, P. E. 1990. Stochastic fairness queueing. In *Proc. IEEE Infocom*, Volume 2 (June 1990), pp. 733–740.

MILLS, D. L. 1988. The Fuzzball. In *Proc. ACM SIGCOMM Conference (SIGCOMM '88)* (Aug. 1988), pp. 115–122.

MOGUL, J. C. AND RAMAKRISHNAN, K. K. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Computer Systems 15*, 3 (Aug.), 217–252.

MORRIS, R., KOHLER, E., JANNOTTI, J., AND KAASHOEK, M. F. 1999. The Click modular router. In *Proc. of the 17th ACM Symposium on Operating Systems Principles (SOSP)* (Dec. 1999), pp. 217–231.

MOSBERGER, D. AND PETERSON, L. L. 1996. Making paths explicit in the Scout operating system. In *Proc. 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)* (Oct. 1996), pp. 153–167.

NEWMAN, P., MINSHALL, G., AND LYON, T. L. 1998. IP switching—ATM under IP. *IEEE/ACM Trans. Networking 6*, 2 (April), 117–129.

NICHOLS, K., BLAKE, S., BAKER, F., AND BLACK, D. 1998. Definition of the Differentiated Services field (DS field) in the IPv4 and IPv6 headers. RFC 2474 (Dec.), Internet Engineering Task Force. `ftp://ftp.ietf.org/rfc/rfc2474.txt`.

PARTRIDGE, C., CARVEY, P. P., BURGESS, E., CASTINEYRA, I., CLARKE, T., GRAHAM, L., HATHAWAY, M., HERMAN, P., KING, A., KOHALMI, S., MA, T., MCALLEN, J., MENDEZ, T., MILLIKEN, W. C., PETTYJOHN, R., ROKOSZ, J., SEEGER, J., SOLLINS, M., STORCH, S., TOBER, B., TROXEL, G. D., WAITZMAN, D.,

AND WINTERBLE, S. 1998. A 50-Gb/s IP router. *IEEE/ACM Trans. Networking 6*, 3 (June), 237–248.

PETERSON, L. L., KARLIN, S. C., AND LI, K. 1999. OS support for general-purpose routers. In *Proc. 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)* (March 1999), pp. 38–43. IEEE Computer Society Technical Committee on Operating Systems.

POSTEL, J. 1981a. Internet Protocol. RFC 791 (Sept.), Internet Engineering Task Force. `ftp://ftp.ietf.org/rfc/rfc0791.txt`.

POSTEL, J. 1981b. Internet Control Message Protocol. RFC 792 (Sept.), Internet Engineering Task Force. `ftp://ftp.ietf.org/rfc/rfc0792.txt`.

RITCHIE, D. M. 1984. A stream input-output system. *AT&T Bell Laboratories Technical Journal 63*, 8 (Oct.), 1897–1910.

SMITH, J. M., CALVERT, K. L., MURPHY, S. L., ORMAN, H. K., AND PETERSON, L. L. 1999. Activating networks: a progress report. *IEEE Computer 32*, 4 (April), 32–41.

TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W., WETHERALL, D. J., AND MINDEN, G. J. 1997. A survey of active network research. *IEEE Communications Magazine 35*, 1 (Jan.), 80–86.

WALDSPURGER, C. A. AND WEIHL, W. E. 1995. Stride scheduling: deterministic proportional-share resource management. Technical Memo MIT/LCS/TM-528 (June), MIT Laboratory for Computer Science.

WALDVOGEL, M., VARGHESE, G., TURNER, J., AND PLATTNER, B. 1997. Scalable high speed IP routing lookups. In *Proc. ACM SIGCOMM Conference (SIGCOMM '97)* (Oct. 1997), pp. 25–38.

WROCLAWSKI, J. 1997. Fast PC routers. Technical report (Jan.), MIT LCS Advanced Network Architecture Group. `http://mercury.lcs.mit.edu/PC-Routers/pcrouter.html`, as of July 2000.

Received March 2000; revised July 2000; accepted July 2000

Consistency Analysis in Bloom: a CALM and Collected Approach

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak

{palvaro, nrc, hellerstein, wrm}@cs.berkeley.edu
University of California, Berkeley

ABSTRACT

Distributed programming has become a topic of widespread interest, and many programmers now wrestle with tradeoffs between data consistency, availability and latency. Distributed transactions are often rejected as an undesirable tradeoff today, but in the absence of transactions there are few concrete principles or tools to help programmers design and verify the correctness of their applications.

We address this situation with the *CALM* principle, which connects the idea of distributed consistency to program tests for logical monotonicity. We then introduce *Bloom*, a distributed programming language that is amenable to high-level consistency analysis and encourages order-insensitive programming. We present a prototype implementation of Bloom as a domain-specific language in Ruby. We also propose a program analysis technique that identifies *points of order* in Bloom programs: code locations where programmers may need to inject coordination logic to ensure consistency. We illustrate these ideas with two case studies: a simple key-value store and a distributed shopping cart service.

1. INTRODUCTION

Until fairly recently, distributed programming was the domain of a small group of experts. But recent technology trends have brought distributed programming to the mainstream of open source and commercial software. The challenges of distribution—concurrency and asynchrony, performance variability, and partial failure—often translate into tricky data management challenges regarding task coordination and data consistency. Given the growing need to wrestle with these challenges, there is increasing pressure on the data management community to help find solutions to the difficulty of distributed programming.

There are two main bodies of work to guide programmers through these issues. The first is the "ACID" foundation of distributed transactions, grounded in the theory of serializable read/write schedules and consensus protocols like Paxos and Two-Phase Commit. These techniques provide strong consistency guarantees, and can help shield programmers from much of the complexity of distributed programming. However, there is a widespread belief that the costs of these mechanisms are too high in many important scenarios where availability and/or low-latency response is critical. As a result, there is a great deal of interest in building distributed software that avoids using these mechanisms.

The second point of reference is a long tradition of research and system development that uses application-specific reasoning to tolerate "loose" consistency arising from flexible ordering of reads, writes and messages (e.g., [6, 12, 13, 18, 24]). This approach enables machines to continue operating in the face of temporary delays, message reordering, and component failures. The challenge with this design style is to ensure that the resulting software tolerates the inconsistencies in a meaningful way, producing acceptable results in all cases. Although there is a body of wisdom and best practices that informs this approach, there are few concrete software development tools that codify these ideas. Hence it is typically unclear what guarantees are provided by systems built in this style, and the resulting code is hard to test and hard to trust.

Merging the best of these traditions, it would be ideal to have a robust theory and practical tools to help programmers reason about and manage high-level program properties in the face of loosely coordinated consistency. In this paper we demonstrate significant progress in this direction. Our approach is based on the use of a declarative language and program analysis techniques that enable both static analyses and runtime annotations of consistency. We begin by introducing the *CALM* principle, which connects the theory of non-monotonic logic to the need for distributed coordination to achieve consistency. We present an initial version of our *Bloom* declarative language, and translate concepts of monotonicity into a practical program analysis technique that detects potential consistency anomalies in distributed Bloom programs. We then show how such anomalies can be handled by a programmer during the development process, either by introducing coordination mechanisms to ensure consistency or by applying program rewrites that can track inconsistency "taint" as it propagates through code. To illustrate the Bloom language and the utility of our analysis, we present two case studies: a replicated key-value store and a fault-tolerant shopping cart service.

2. CONSISTENCY AND LOGICAL MONOTONICITY (CALM)

In this section we present the connection between distributed consistency and logical monotonicity. This discussion informs the language and analysis tools we develop in subsequent sections.

A key problem in distributed programming is reasoning about the consistent behavior of a program in the face of *temporal nondeterminism*: the delay and re-ordering of messages and data across nodes. Because delays can be unbounded, analysis typically focuses on "eventual consistency" after all messages have been delivered [26]. A sufficient condition for eventual consistency is *order independence*:

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2011.
5th *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9–12, 2011, Asilomar, California, USA.
.

the independence of program execution from temporal nondeterminism.

Order independence is a key attribute of declarative languages based on sets, which has led most notably to the success of parallel databases and web search infrastructure. But even set-oriented languages can require a degree of ordering in their execution if they are sufficiently expressive. The theory of relational databases and logic programming provides a framework to reason about these issues. *Monotonic* programs—e.g., programs expressible via selection, projection and join (even with recursion)—can be implemented by streaming algorithms that incrementally produce output elements as they receive input elements. The final order or contents of the input will never cause any earlier output to be "revoked" once it has been generated.[1] *Non-monotonic* programs—e.g., those that contain aggregation or negation operations—can only be implemented correctly via blocking algorithms that do not produce any output until they have received all tuples in logical partitions of an input set. For example, aggregation queries need to receive entire "groups" before producing aggregates, which in general requires receiving the entire input set.

The implications for distributed programming are clear. Monotonic programs are easy to distribute: they can be implemented via streaming set-based algorithms that produce actionable outputs to consumers while tolerating message reordering and delay from producers. By contrast, even simple non-monotonic tasks like counting are difficult in distributed systems. As a mnemonic, we say that *counting requires waiting* in a distributed system: in general, a complete count of distributed data must wait for all its inputs, including stragglers, before producing the correct output.

"Waiting" is specified in a program via *coordination logic*: code that (a) computes and transmits auxiliary information from producers to enable the recipient to determine when a set has completely arrived across the network, and (b) postpones production of results for consumers until after that determination is made. Typical coordination mechanisms include sequence numbers, counters, and consensus protocols like Paxos or Two-Phase Commit.

Interestingly, these coordination mechanisms themselves typically involve counting. For example, Paxos requires counting messages to establish that a majority of the members have agreed to a proposal; Two-Phase Commit requires counting to establish that all members have agreed. Hence we also say that *waiting requires counting*, the converse of our earlier mnemonic.

Our observations about waiting and counting illustrate the crux of what we call the *CALM* principle: the tight relationship between Consistency And Logical Monotonicity. Monotonic programs *guarantee* eventual consistency under any interleaving of delivery and computation. By contrast, non-monotonicity—the property that adding an element to an input set may revoke a previously valid element of an output set—requires coordination schemes that "wait" until inputs can be guaranteed to be complete.

We typically wish to minimize the use of coordination, because of well-known concerns about latency and availability in the face of message delays and network partitions. We can use the CALM principle to develop checks for distributed consistency in logic languages, where conservative tests for monotonicity are well understood. A simple syntactic check is often sufficient: if the program does not contain any of the symbols in the language that correspond to non-monotonic operators (e.g., `NOT IN` or aggregate symbols), then it is monotonic and can be implemented without coordination, regardless of any read-write dataflow dependencies in the code. As students of the logic programming literature will recognize [19, 20, 21], these conservative checks can be refined further to consider semantics of predicates in the language. For example, the expression "`MIN(x)` < 100" is monotonic despite containing an aggregate, by virtue of the semantics of `MIN` and <: once a subset S satisfies this test, any superset of S will also satisfy it. Many refinements along these lines exist, increasing the ability of program analyses to verify monotonicity.

In cases where an analysis cannot guarantee monotonicity of a whole program, it can instead provide a conservative assessment of the points in the program where coordination may be required to ensure consistency. For example, a shallow syntactic analysis could flag all non-monotonic predicates in a program (e.g., `NOT IN` tests or predicates with aggregate values as input). The loci produced by a non-monotonicity analysis are the program's *points of order*. A program with non-monotonicity can be made consistent by including coordination logic at its points of order.

The reader may observe that because "waiting requires counting," adding a code module with coordination logic actually increases the number of syntactic points of order in a program. To avoid this problem, the coordination module itself must be verified for order independence, either manually or via a refined monotonicity test during analysis. When the verification is done by hand, annotations can inform the analysis tool to skip the module in its analysis, and hence avoid attempts to coordinate the coordination logic.

Because analyses based on the CALM principle operate with information about program semantics, they can avoid coordination logic in cases where traditional read/write analysis would require it. Perhaps more importantly, as we will see in our discussion of shopping carts (Section 5), logic languages and the analysis of points of order can help programmers redesign code to reduce coordination requirements.

3. BUD: BLOOM UNDER DEVELOPMENT

Bloom is based on the conjecture that many of the fundamental problems with parallel programming come from a legacy of ordering assumptions implicit in classical von Neumann architectures. In the von Neumann model, state is captured in an ordered array of addresses, and computation is expressed via an ordered list of instructions. Traditional imperative programming grew out of these pervasive assumptions about order. Therefore, it is no surprise that popular imperative languages are a bad match to parallel and distributed platforms, which make few guarantees about order of execution and communication. By contrast, set-oriented approaches like SQL and batch dataflow approaches like MapReduce translate better to architectures with loose control over ordering.

Bloom is designed in the tradition of programming styles that are "disorderly" by nature. State is captured in unordered sets. Computation is expressed in logic: an unordered set of declarative rules, each consisting of an unordered conjunction of predicates. As we discuss below, mechanisms for imposing order are available when needed, but the programmer is provided with tools to evaluate the need for these mechanisms as special-case behaviors, rather than a default model. The result is code that runs naturally on distributed machines with a minimum of coordination overhead.

Unlike earlier efforts such as Prolog, active database languages, and our own Overlog language for distributed systems [16], Bloom is *purely declarative*: the syntax of a program contains the full specification of its semantics, and there is no need for the programmer to understand or reason about the behavior of the evaluation engine. Bloom is based on a formal temporal logic called Dedalus [3].

The prototype version of Bloom we describe here is embodied in an implementation we call *Bud* (Bloom Under Development). Bud

[1] Formally, in a monotonic logic program, any true statement continues to be true as new axioms—including new facts—are added to the program.

Type	Behavior
table	A collection whose contents persist across timesteps.
scratch	A collection whose contents persist for only one timestep.
channel	A scratch collection with one attribute designated as the *location specifier*. Tuples "appear" at the network address stored in their location specifier.
periodic	A scratch collection of key-value pairs (`id`, `timestamp`). The definition of a periodic collection is parameterized by a `period` in seconds; the runtime system arranges (in a best-effort manner) for tuples to "appear" in this collection approximately every `period` seconds, with a unique `id` and the current wall-clock time.
interface	A scratch collection specially designated as an interface point between modules.

Op	Valid lhs types	Meaning
=	**scratch**	rhs defines the contents of the lhs for the current timestep. lhs must not appear in lhs of any other statement.
<=	**table**, **scratch**	lhs includes the content of the rhs in the current timestep.
<+	**table**, **scratch**	lhs will include the content of the rhs in the next timestep.
<-	**table**	tuples in the rhs will be absent from the lhs at the start of the next timestep.
<~	**channel**	tuples in the rhs will appear in the (remote) lhs at some non-deterministic future time.

Figure 1: Bloom collection types and operators.

is a domain-specific subset of the popular Ruby scripting language and is evaluated by a stock Ruby interpreter via a Bud Ruby class. Compared to other logic languages, we feel it has a familiar and programmer-friendly flavor, and we believe that its learning curve will be relatively flat for programmers familiar with modern scripting languages. Bud uses a Ruby-flavored syntax, but this is not fundamental; we have experimented with analogous Bloom embeddings in other languages including Python, Erlang and Scala, and they look similar in structure.

3.1 Bloom Basics

Bloom programs are bundles of declarative statements about collections of "facts" or tuples, similar to SQL views or Datalog rules. A statement can only reference data that is local to a node. Bloom statements are defined with respect to atomic "timesteps," which can be implemented via successive rounds of evaluation. In each timestep, certain "ground facts" exist in collections due to persistence or the arrival of messages from outside agents (e.g., the network or system clock). The statements in a Bloom program specify the derivation of additional facts, which can be declared to exist either in the current timestep, at the very next timestep, or at some non-deterministic time in the future at a remote node.

A Bloom program also specifies the way that facts persist (or do not persist) across consecutive timesteps on a single node. Bloom is a side-effect free language with no "mutable state": if a fact is defined at a given timestep, its existence at that timestep cannot be refuted by any expression in the language. This technicality is key to avoiding many of the complexities involved in reasoning about earlier "stateful" rule languages. The paper on Dedalus discusses these points in more detail [3].

3.2 State in Bloom

Bloom programs manage state using five collection types described in the top of Figure 1. A collection is defined with a relational-style schema of named columns, including an optional subset of those columns that forms a primary key. Line 15 in Figure 2 defines a collection named `send_buf` with four columns `dst`, `src`, `ident`, and `payload`; the primary key is `(dst, src, ident)`. The type system for columns is taken from Ruby, so it is possible to have a column based on any Ruby class the programmer cares to define or import (including nested Bud collections). In Bud, a tuple in a collection is simply a Ruby array containing as many elements as the columns of the collection's schema. As in other object-relational ADT schemes like Postgres [23], column values can be manipulated using their own (non-destructive) methods. Bloom also provides for nesting and unnesting of collections using standard Ruby constructs like `reduce` and `flat_map`. Note that collections in Bloom provide set semantics—collections do not contain duplicates.

```
 0 module DeliveryProtocol
 1   def state
 2     interface input, :pipe_in,
 3       ['dst', 'src', 'ident'], ['payload']
 4     interface output, :pipe_sent,
 5       ['dst', 'src', 'ident'], ['payload']
 6   end
 7 end

 9 module ReliableDelivery
10   include DeliveryProtocol

12   def state
13     channel :data_chan, ['@dst', 'src', 'ident'], ['payload']
14     channel :ack_chan, ['@src', 'dst', 'ident']
15     table :send_buf, ['dst', 'src', 'ident'], ['payload']
16     periodic :timer, 10
17   end

19   declare
20   def send_packet
21     send_buf <= pipe_in
22     data_chan <~ pipe_in
23   end

25   declare
26   def timer_retry
27     data_chan <~ join([send_buf, timer]).map{|p, t| p}
28   end

30   declare
31   def send_ack
32     ack_chan <~ data_chan.map{|p| [p.src, p.dst, p.ident]}
33   end

35   declare
36   def recv_ack
37     got_ack = join [ack_chan, send_buf],
38                    [ack_chan.ident, send_buf.ident]
39     pipe_sent <= got_ack.map{|a, sb| sb}
40     send_buf <- got_ack.map{|a, sb| sb}
41   end
42 end
```

Figure 2: Reliable unicast messaging in Bloom.

The persistence of a tuple is determined by the type of the collection that contains the tuple. **scratch** collections are useful for transient data like intermediate results and "macro" definitions that enable code reuse. The contents of a **table** persist across consecutive timesteps (until that persistence is interrupted via a Bloom statement containing the `<-` operator described below). Although there are precise declarative semantics for this persistence [3], it is convenient to think operationally as follows: scratch collections are "emptied" before each timestep begins, tables are "stored" collections (similar to tables in SQL), and the `<-` operator represents batch deletion before the beginning of the next timestep.

The facts of the "real world," including network messages and the passage of wall-clock time, are captured via **channel** and **periodic** collections; these are scratch collections whose contents "appear" at non-deterministic timesteps. The paper on Dedalus delves deeper

Method	Description
bc.map	Takes a code block and returns the collection formed by applying the code block to each element of bc.
bc.flat_map	Equivalent to map, except that any nested collections in the result are flattened.
bc.reduce	Takes a memo variable and code block, and applies the block to memo and each element of bc in turn.
bc.empty?	Returns true if bc is empty.
bc.include?	Takes an object and returns true if that object is equal to any element of bc.
bc.group	Takes a list of grouping columns, a list of aggregate expressions and a code block. For each group, computes the aggregates and then applies the code block to the group/aggregation result.
join, leftjoin, outerjoin, natjoin	Methods of the Bud class to compute join variants over BudCollections. join, leftjoin and outerjoin take an array of collections to join, as well as a variable-length list of arrays of join conditions. The natural join natjoin takes only the array of BudCollection objects as an argument.

Figure 3: Commonly used methods of the BudCollection class.

into the logical semantics of this non-determinism [3]. Note that failure of nodes or communication is captured here: it can be thought of as the repeated "non-appearance" of a fact at every timestep. Again, it is convenient to think operationally as follows: the facts in a channel are sent to a remote node via an unreliable transport protocol like UDP; the address of the remote node is indicated by a distinguished column in the channel called the *location specifier* (denoted by the symbol @). The definition of a periodic collection instructs the runtime to "inject" facts at regular wall-clock intervals to "drive" further derivations. Lines 13 and 16 in Figure 2 contain examples of channel and periodic definitions, respectively.

The final type of collection is an **interface**, which specifies a connection point between Bloom modules. Interfaces are described in Section 3.4.

3.3 Bloom Statements

Bloom statements are declarative relational expressions that define the contents of derived collections. They can be viewed operationally as specifying the insertion or accumulation of expression results into collections. The syntax is:

<collection-variable> <op> <collection-expression>

The bottom of Figure 1 describes the five operators that can be used to define the contents of the left-hand side (lhs) in terms of the right-hand side (rhs). As in Datalog, the lhs of a statement may be referenced recursively in its rhs, or recursion can be defined mutually across statements.

In the Bud prototype, both the lhs and rhs are instances of (a descendant of) a Ruby class called BudCollection, which supports several useful methods for manipulating collections (Figure 3).[2] The rhs of a statement typically invokes BudCollection methods on one or more collection objects to produce a derived collection. The most commonly used method is map, which applies a scalar operation to every tuple in a collection; this can be used to implement relational selection and projection. For example, line 32 of Figure 2 projects the data_chan collection to its src, dst, and ident fields. Multiway joins are specified using the join method, which takes a list of input collections and an optional list of join conditions. Lines 37–38 of Figure 2 show a join between ack_chan and send_buf. Syntax sugar for natural joins and outer joins is also provided. BudCollection also defines a group method similar to SQL's GROUP BY, supporting the standard SQL aggregates; for example, lines 15–17 of Figure 14 compute the count of unique reqid values for every combination of values for session, item and action.

Bloom statements are specified within method definitions that are flagged with the declare keyword (e.g., line 20 of Figure 2). The semantics of a Bloom program are defined by the union of its declare methods; the order of statements is immaterial. Dividing statements into multiple methods improves the readability of the program and also allows the use of Ruby's method overriding and inheritance features: because a Bloom class is just a stylized Ruby class, any of the methods in a Bloom class can be overridden by a subclass. We expand upon this idea next.

3.4 Modules and Interfaces

Conventional wisdom in certain quarters says that rule-based languages are untenable for large programs that evolve over time, since the interactions among rules become too difficult to understand. Bloom addresses this concern in two different ways. First, unlike many prior rule-based languages, Bloom is purely declarative; this avoids forcing the programmer to reason about the interaction between declarative statements and imperative constructs. Second, Bloom borrows object-oriented features from Ruby to enable programs to be broken into small modules and to allow modules to interact with one another by exposing narrow interfaces. This aids program comprehension, because it reduces the amount of code a programmer needs to read to understand the behavior of a module.

A Bloom module is a bundle of collections and statements. Like modules in Ruby, a Bloom module can "mixin" one or more other modules via the include statement; mixing-in a module imports its collections and statements. A common pattern is to specify an abstract interface in one module and then use the mixin feature to specify several concrete realizations in separate modules. To support this idiom, Bloom provides a special type of collection called an **interface**. An input interface defines a place where a module accepts stimuli from the outside world (e.g., other Bloom modules). Typically, inserting a fact into an input interface results in a corresponding fact appearing (perhaps after a delay) in one of the module's output interfaces.

For example, the DeliveryProtocol module in Figure 2 defines an abstract interface for sending messages to a remote address. Clients use an implementation of this interface by inserting a fact into pipe_in; this represents a new message to be delivered. A corresponding fact will eventually appear in the pipe_sent output interface; this indicates that the delivery operation has been completed. The ReliableDelivery module of Figure 2 is one possible implementation of the abstract DeliveryProtocol interface—it uses a buffer and acknowledgment messages to delay emitting a pipe_sent fact until the corresponding message has been acknowledged by the remote node. Figure 18 in Appendix A contains a different implementation of the abstract DeliveryProtocol. A client program that is indifferent to the details of message delivery can simply interact with the abstract DeliveryProtocol; the particular implementation of this protocol can be chosen independently.

A common requirement is for one module to "override" some of the statements in a module that it mixes in. For example, an OrderedDelivery module might want to reuse the functionality provided by ReliableDelivery but prevent a message with sequence number x from being delivered until all messages with sequence numbers $< x$ have been acknowledged. To support this pattern, Bloom allows an interface defined in another module to be overridden simply by redeclaring it. Internally, both of these redundantly-named interfaces exist in the namespace of the module that declared them, but they

[2]Note that many of these methods are provided by the standard Ruby Enumerable module, which BudCollection imports.

only need to be referenced by a fully qualified name if their use is otherwise ambiguous. If an input interface appears in the lhs of a statement in a module that declared the interface, it is rewritten to reference the interface with the same name in a mixed-in class, because a module cannot insert into its own input interface. The same is the case for output interfaces appearing in the rhs of statements. This feature allows programmers to reuse existing modules and interpose additional logic in a style reminiscent of superclass invocation in object-oriented languages. We provide an example of interface overriding in Section 4.3.

3.5 Bud Implementation

Bud is intended to be a lightweight rapid prototype of Bloom: a first effort at embodying the Dedalus logic in a syntax familiar to programmers. Bud consists of less than 2400 lines of Ruby code, developed as a part-time effort over the course of a semester.

A Bud program is just a Ruby class definition. To make it operational, a small amount of imperative Ruby code is needed to create an instance of the class and invoke the Bud `run` method. This imperative code can then be launched on as many nodes as desired (e.g., via the popular Capistrano package for Ruby deployments). As an alternative to the `run` method, the Bud class also provides a `tick` method that can be used to force evaluation of a single timestep; this is useful for debugging Bloom code with standard Ruby debugging tools or for executing a Bud program that is intended as a "one-shot" query.

Because Bud is pure Ruby, some programmers may choose to embed it as a domain-specific language (DSL) within traditional imperative Ruby code. In fact, nothing prevents a subclass of Bud from having both Bloom code in `declare` methods and imperative code in traditional Ruby methods. This is a fairly common usage model for many DSLs. A mixture of declarative Bloom methods and imperative Ruby allows the full range of existing Ruby code—including the extensive RubyGems repositories—to be combined with checkable distributed Bloom programs. The analyses we describe in the remaining sections still apply in these cases; the imperative Ruby code interacts with the Bloom logic in the same way as any external agent sending and receiving network messages.

4. CASE STUDY: KEY-VALUE STORE

In this section, we present two variants of a key-value store (KVS) implemented using Bloom.[3] We begin with an abstract protocol that any key-value store will satisfy, and then provide both single-node and replicated implementations of this protocol. We then introduce a graphical visualization of the dataflow in a Bloom program and use this visualization to reason about the *points of order* in our programs: places where additional coordination may be required to guarantee consistent results.

4.1 Abstract Key-Value Store Protocol

Figure 4 specifies a protocol for interacting with an abstract key-value store. The protocol comprises two input interfaces (representing attempts to insert and fetch items from the store) and a single output interface (which represents the outcome of a fetch operation). To use an implementation of this protocol, a Bloom program can store key-value pairs by inserting facts into `kvput`. To retrieve the value associated with a key, the client program inserts a fact into `kvget` and looks for a corresponding response tuple in `kvget_response`. For both put and get operations, the client must supply a unique request identifier (`reqid`) to differentiate tuples in the event of multiple concurrent requests.

[3]The complete source code for both of the case studies presented in this paper can be found at `http://boom.cs.berkeley.edu/cidr11/`.

```
module KVSProtocol
  def state
    interface input, :kvput,
      ['client', 'key', 'reqid'], ['value']
    interface input, :kvget, ['reqid'], ['key']
    interface output, :kvget_response,
      ['reqid'], ['key', 'value']
  end
end
```

Figure 4: Abstract key-value store protocol.

```
0  module BasicKVS
1    include KVSProtocol

3    def state
4      table :kvstate, ['key'], ['value']
5    end

7    declare
8    def do_put
9      kvstate <+ kvput.map{|p| [p.key, p.value]}
10     prev = join [kvstate, kvput], [kvstate.key, kvput.key]
11     kvstate <- prev.map{|b, p| b}
12   end

14   declare
15   def do_get
16     getj = join [kvget, kvstate], [kvget.key, kvstate.key]
17     kvget_response <= getj.map do |g, t|
18       [g.reqid, t.key, t.value]
19     end
20   end
21 end
```

Figure 5: Single-node key-value store implementation.

A module which uses a key-value store but is indifferent to the specifics of the implementation may simply mixin the abstract protocol and postpone committing to a particular implementation until runtime. As we will see shortly, an implementation of the KVSProtocol is a collection of Bloom statements that read tuples from the protocol's input interfaces and send results to the output interface.

4.2 Single-Node Key-Value Store

Figure 5 contains a single-node implementation of the abstract key-value store protocol. Key-value pairs are stored in a persistent table called `kvstate` (line 4). When a `kvput` tuple is received, its key-value pair is stored in `kvstate` at the next timestep (line 9). If the given key already exists in `kvstate`, we want to replace the key's old value. This is done by joining `kvput` against the current version of `kvstate` (line 10). If a matching tuple is found, the old key-value pair is removed from `kvstate` at the beginning of the next timestep (line 11). Note that we also insert the new key-value pair into `kvstate` in the next timestep (line 9); hence, an overwriting update is implemented as an atomic deletion and insertion.

4.3 Replicated Key-Value Store

Next, we extend the basic key-value store implementation to support replication (Figure 6). To communicate between replicas, we use a simple multicast library implemented in Bloom; the source code for this library can be found in Appendix A. To send a multicast, a program inserts a fact into `send_mcast`; a corresponding fact appears in `mcast_done` when the multicast is complete. The multicast library also exports the membership of the multicast group in a table called `members`.

Our replicated key-value store is implemented on top of the single-node key-value store described in the previous section. When a new key is inserted by a client, we multicast the insertion to the other

```
0  module ReplicatedKVS
1    include BasicKVS
2    include MulticastProtocol

4    def state
5      interface input, :kvput,
6        ['client', 'key', 'reqid'], ['value']
7    end

9    declare
10   def replicate
11     send_mcast <= kvput.map do |k|
12       unless members.include? [k.client]
13         [k.reqid, [@local_addr, k.key, k.reqid, k.value]]
14       end
15     end
16   end

18   declare
19   def apply_put
20     kvput <= mcast_done.map{|m| m.payload}

22     kvput <= pipe_chan.map do |d|
23       if d.payload.fetch(1) != @local_addr
24         d.payload
25       end
26     end
27   end
28 end
```

Figure 6: Replicated key-value store implementation.

```
0  class RealizedReplicatedKVS < Bud
1    include ReplicatedKVS
2    include SimpleMulticast
3    include BestEffortDelivery
4  end

6  kvs = RealizedReplicatedKVS.new("localhost", 12345)
7  kvs.run
```

Figure 7: A fully specified key-value store program.

replicas (lines 11–15). To avoid repeated multicasts of the same inserted key, we avoid multicasting updates we receive from another replica (line 12). We apply an update to our local `kvstate` table in two cases: (1) if a multicast succeeds at the node that originated it (line 20) (2) whenever a multicast is received by a peer replica (lines 22–26). Note that `@local_addr` is a Ruby instance variable defined by Bud that contains the network address of the current Bud instance.

In Figure 6 ReplicatedKVS wants to "intercept" `kvput` events from clients, and only apply them to the underlying BasicKVS module when certain conditions are met. To achieve this, we "override" the declaration of the `kvput` input interface as discussed in Section 3.4 (lines 5–6). In ReplicatedKVS, references to `kvput` appearing in the lhs of statements are resolved to the `kvput` provided by BasicKVS, while references in the rhs of statements resolve to the local `kvput`. As described in Section 3.4, this is unambiguous because a module cannot insert into its own input or read from its own output interfaces.

Figure 7 combines ReplicatedKVS with a concrete implementation of MulticastProtocol and DeliveryProtocol. The resulting class, a subclass of Bud, may be instantiated and run as shown in lines 6 and 7.

4.4 Predicate Dependency Graphs

Now that we have introduced two concrete implementations of the abstract key-value store protocol, we turn to analyzing the properties of these programs. We begin by describing the graphical dataflow representation used by our analysis. In the following section, we

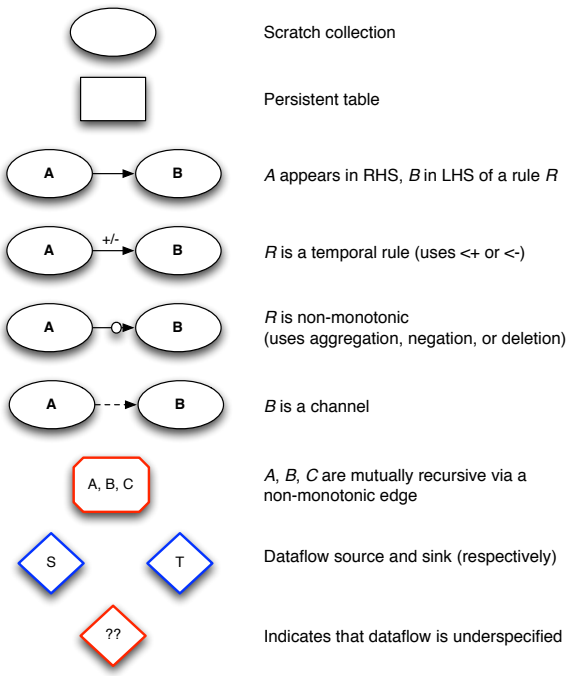


Figure 8: Visual analysis legend.

discuss the dataflow graphs generated for the two key-value store implementations.

A Bloom program may be viewed as a dataflow graph with external input interfaces as sources, external output interfaces as sinks, collections as internal nodes, and rules as edges. This graph represents the dependencies between the collections in a program and is generated automatically by the Bud interpreter. Figure 8 contains a list of the different symbols and annotations in the graphical visualization; we provide a brief summary below.

Each node in the graph is either a collection or a cluster of collections; tables are shown as rectangles, ephemeral collections (scratch, periodic and channel) are depicted as ovals, and clusters (described below) as octagons. A directed edge from node A to node B indicates that B appears in the lhs of a Bloom statement that references A in the rhs, either directly or through a join expression. An edge is annotated based on the operator symbol in the statement. If the statement uses the <+ or <- operators, the edge is marked with "+/−". This indicates that facts traversing the edge "spend" a timestep to move from the rhs to the lhs. Similarly, if the statement uses the <~ operator, the edge is a dashed line—this indicates that facts from the rhs appear at the lhs at a non-deterministic future time. If the statement involves a non-monotonic operation (aggregation, negation, or deletion via the <- operator), then the edge is marked with a white circle. To make the visualizations more readable, any strongly connected component marked with both a circle and a +/− edge is collapsed into an octagonal "temporal cluster," which can be viewed abstractly as a single, non-monotonic node in the dataflow. Any non-monotonic edge in the graph is a *point of order*, as are all edges incident to a temporal cluster, including their implicit self-edge.

4.5 Analysis

Figure 9 presents a visual representation of the abstract key-value store protocol. Naturally, the abstract protocol does not specify a connection between the input and output events; this is indicated in the diagram by the red diamond labeled with "??", denoting an underspecified dataflow. A concrete realization of the key-value

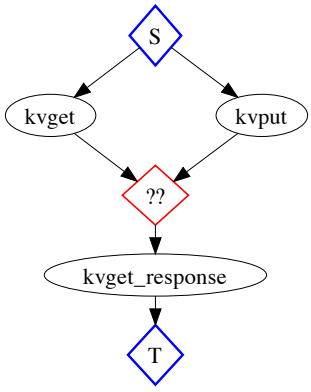


Figure 9: Visualization of the abstract key-value store protocol.

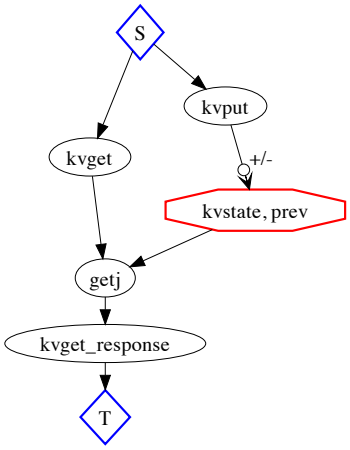


Figure 10: Visualization of the single-node key-value store.

store protocol must, at minimum, supply a dataflow that connects an input interface to an output interface.

Figure 10 shows the visual analysis of the single-node KVS implementation, which supplies a concrete dataflow for the unspecified component in the previous graph. `kvstate` and `prev` are collapsed into a red octagon because they are part of a strongly connected component in the graph with both negative and temporal edges. Any data flowing from `kvput` to the sink must cross at least one non-monotonic point of order (at ingress to the octagon) and possibly an arbitrary number of them (by traversing the dependency cycle collapsed into the octagon), and any path from `kvget` to the sink must join state potentially affected by non-monotonicity (because `kvstate` is used to derive `kvget_response`).

Reviewing the code in Figure 5, we see the source of the non-monotonicity. The contents of `kvstate` may be defined via a "destructive" update that combines the previous state and the current input from `kvput` (lines 9–11 of Figure 5). Hence the contents of `kvstate` may depend on the order of arrival of `kvput` tuples.

5. CASE STUDY: SHOPPING CART

In this section, we develop two different designs for a distributed shopping-cart service in Bloom. In a shopping cart system, clients add and remove items from their shopping cart. To provide fault tolerance and persistence, the content of the cart is stored by a

```
0  module CartProtocol
1    def state
2      channel :action_msg,
3        ['@server', 'client', 'session', 'reqid'],
4        ['item', 'action']
5      channel :checkout_msg,
6        ['@server', 'client', 'session', 'reqid']
7      channel :response_msg,
8        ['@client', 'server', 'session'], ['contents']
9    end
10 end

12 module CartClientProtocol
13   def state
14     interface input, :client_action,
15       ['server', 'session', 'reqid'], ['item', 'action']
16     interface input, :client_checkout,
17       ['server', 'session', 'reqid']
18     interface output, :client_response,
19       ['client', 'server', 'session'], ['contents']
20   end
21 end
```

Figure 11: Abstract shopping cart protocol.

```
module CartClient
  include CartProtocol
  include CartClientProtocol

  declare
  def client
    action_msg <~ client_action.map do |a|
      [a.server, @local_addr, a.session, a.reqid, a.item, a.action]
    end
    checkout_msg <~ client_checkout.map do |a|
      [a.server, @local_addr, a.session, a.reqid]
    end
    client_response <= response_msg
  end
end
```

Figure 12: Shopping cart client implementation.

collection of server replicas. Once a client has finished shopping, they perform a "checkout" request, which returns the final state of their cart.

After presenting the abstract shopping cart protocol and a simple client program, we implement a "destructive," state-modifying shopping cart service that uses the key-value store introduced in Section 4. Second, we illustrate a "disorderly" cart that accumulates updates in a set-wise fashion, summarizing updates at checkout into a final result. These two different designs illustrate our analysis tools and the way they inform design decisions for distributed programming.

5.1 Shopping Cart Client

An abstract shopping cart protocol is presented in Figure 11. Figure 12 contains a simple shopping cart client program: it takes client operations (represented as `client_action` and `client_checkout` facts) and sends them to the shopping cart service using the CartProtocol. We omit logic for clients to choose a cart server replica; this can be based on simple policies like round-robin or random selection, or via more explicit load balancing.

5.2 "Destructive" Shopping Cart Service

We begin with a shopping cart service built on a key-value store. Each cart is a `(key,value)` pair, where `key` is a unique session identifier and `value` is an object containing the session's state, including a Ruby array that holds the items currently in the cart. Adding or deleting items from the cart result in "destructive" updates: the value associated with the key is replaced by a new value

```
0  module DestructiveCart
1    include CartProtocol
2    include KVSProtocol

4    declare
5    def do_action
6      kvget <= action_msg.map{|a| [a.reqid, a.key]}

8      kvput <= action_msg.map do |a|
9        if a.action == "A"
10         unless kvget_response.map{|b| b.key}.include? a.session
11           [a.server, a.client, a.session, a.reqid, [a.item]]
12         end
13       end
14     end

16     old_state = join [kvget_response, action_msg],
17                      [kvget_response.key, action_msg.session]
18     kvput <= old_state.map do |b, a|
19       if a.action == "A"
20         [a.server, a.client, a.session,
21          a.reqid, b.value.push(a.item)]
22       elsif a.action == "D"
23         [a.server, a.client, a.session,
24          a.reqid, delete_one(b.value, a.item)]
25       end
26     end
27   end

29   declare
30   def do_checkout
31     kvget <= checkout_msg.map{|c| [c.reqid, c.session]}
32     lookup = join [kvget_response, checkout_msg],
33                   [kvget_response.key, checkout_msg.session]
34     response_msg <~ lookup.map do |r, c|
35       [c.client, c.server, c.session, r.value]
36     end
37   end
38 end
```

Figure 13: Destructive cart implementation.

that reflects the effect of the update. Deletion requests are ignored if the item they refer to does not exist in the cart.

Figure 13 shows the Bloom code for this design. The `kvput` collection is provided by the abstract KVSProtocol described in Section 4. Our shopping cart service would work with any concrete realization of the KVSProtocol; we will choose to use the replicated key-value store (Section 4.3) to provide fault-tolerance.

When client actions arrive from the CartClient, the cart service checks to see if there is a record in the key-value store associated with the client's session. If no record is found (i.e., this is the first operation for a new session), then lines 9–13 generate an entry for the new session in `kvstate`. Otherwise, the join conditions in line 17 are satisfied and lines 19–25 "replace" the value in the key-value store with an updated set of items for this session; this uses the built-in overwriting capability provided by the key-value store. When a `checkout_msg` appears at a server replica, the key-value store is queried to retrieve the cart state associated with the given session (lines 31–35), and the results are returned to the client.

5.3 "Disorderly" Shopping Cart Service

Figure 14 shows an alternative shopping cart implementation, in which updates are monotonically accumulated in a set, and summed up only at checkout. Lines 12–14 insert client updates into the persistent table `cart_action`. Lines 15–17 define `action_cnt` as an aggregate over `cart_action`, in the style of an SQL `GROUP BY` statement: for each item associated with a cart, we separately count the number of times it was added and the number of times it was deleted. Lines 22–27 ensure that when a `checkout_msg` tuple arrives, `status` contains a record for every added item for which there was no corresponding deletion in the session. Lines 29–36

```
0  module DisorderlyCart
1    include CartProtocol

3    def state
4      table :cart_action, ['session', 'item', 'action', 'reqid']
5      table :action_cnt, ['session', 'item', 'action'], ['cnt']
6      scratch :status, ['server', 'client', 'session', 'item'],
7                       ['cnt']
8    end

10   declare
11   def do_action
12     cart_action <= action_msg.map do |c|
13       [c.session, c.item, c.action, c.reqid]
14     end
15     action_cnt <= cart_action.group(
16       [cart_action.session, cart_action.item, cart_action.action],
17       count(cart_action.reqid))
18   end

20   declare
21   def do_checkout
22     del_items = action_cnt.map{|a| a.item if a.action == "Del"}
23     status <= join([action_cnt, checkout_msg]).map do |a, c|
24       if a.action == "Add" and not del_items.include? a.item
25         [c.client, c.server, a.session, a.item, a.cnt]
26       end
27     end

29     status <= join([action_cnt, action_cnt,
30                     checkout_msg]).map do |a1, a2, c|
31       if a1.session == a2.session and a1.item == a2.item and
32          a1.session == c.session and
33          a1.action == "A" and a2.action == "D"
34         [c.client, c.server, c.session, a1.item, a1.cnt - a2.cnt]
35       end
36     end

38    response_msg <~ status.group(
39       [status.client, status.server, status.session],
40       accum(status.cnt.times.map{status.item}))
41   end
42 end
```

Figure 14: Disorderly cart implementation.

additionally define `status` as the 3-way join of the `checkout_msg` message and two copies of `action_cnt`—one corresponding to additions and one to deletions. Thus, for each item, `status` contains its final quantity: the difference between the number of additions and deletions (line 34), or simply the number of additions if there are no deletions (line 25). Upon the appearance of a `checkout_msg`, the replica returns a `response_msg` to the client containing the final quantity (lines 38–40). Because the CartClient expects the cart to be returned as an array of items on checkout, we use the `accum` aggregate function to nest the set of items into an array.

5.4 Analysis

Figure 15 presents the analysis of the "destructive" shopping cart variant. Note that because all dependencies are analyzed, collections defined in mixins but not referenced in the code sample (e.g., `pipe_chan`, `member`) also appear in the graph. Although there is no syntactic non-monotonicity in Figure 13, the underlying key-value store uses the non-monotonic `<-` operator to model updateable state. Thus, while the details of the implementation are encapsulated by the key-value store's abstract interface, its points of order resurface in the full-program analysis. Figure 15 indicates that there are points of order between `action_msg`, `member`, and the temporal cluster. This figure also tells the (sad!) story of how we could ensure consistency of the destructive cart implementation: introduce coordination between client and server—and between the chosen server and all its replicas—for *every client action or kvput update*. The programmer can achieve this coordination by supplying a "reliable"

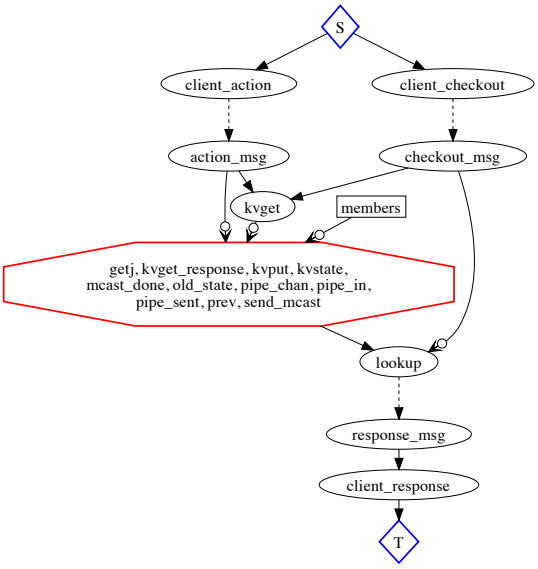


Figure 15: Visualization of the destructive cart program.

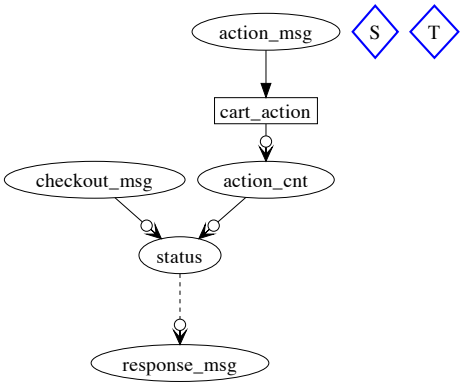


Figure 16: Visualization of the core logic for the disorderly cart.

implementation of multicast that awaits acknowledgements from all replicas before reporting completion: this fine-grained coordination is akin to "eager replication" [9]. Unfortunately, it would incur the latency of a round of messages per server per client update, decrease throughput, and reduce availability in the face of replica failures.

Because we only care about the *set* of elements contained in the value array and not its order, we might be tempted to argue that the shopping cart application is eventually consistent when asynchronously updated and forego the coordination logic. Unfortunately, such informal reasoning can hide serious bugs. For example, consider what would happen if a delete action for an item arrived at some replica before any addition of that item: the delete would be ignored, leading to inconsistencies between replicas.

A happier story emerges from our analysis of the disorderly cart service. Figure 16 shows a visualization of the core logic of the disorderly cart module presented in Figure 14. This program is not complete: its inputs and outputs are channels rather than interfaces, so the dataflow from source to sink is not completed. To complete this program, we must mixin code that connects input and output interfaces to `action_msg`, `checkout_msg`, and `response_msg`, as

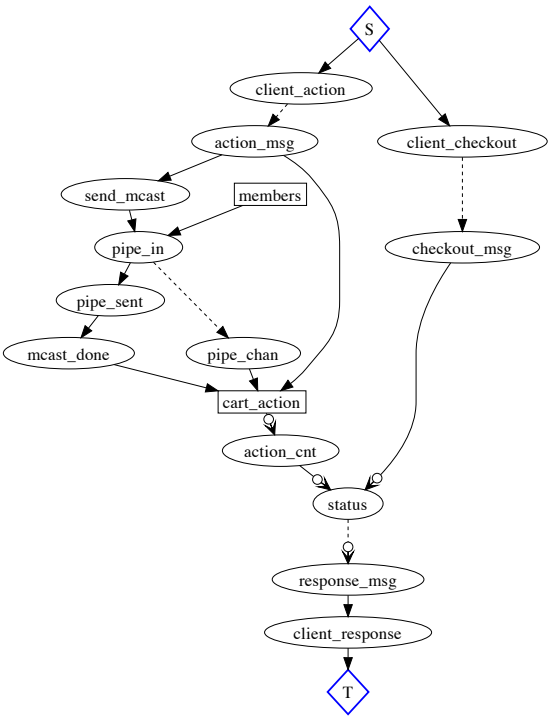


Figure 17: Visualization of the complete disorderly cart program.

the CartClient does (Figure 12). Note that the disorderly cart has points of order on all paths but there are no cycles.

Figure 17 shows the analysis for a complete implementation that mixes in both the client code and logic to replicate the `cart_action` table via best-effort multicast (see Figure 20 in Appendix A for the corresponding source code). Note that communication (via `action_msg`) between client and server—and among server replicas—crosses no points of order, so all the communication related to shopping actions converges to the same final state without coordination. However, there are points of order upon the appearance of `checkout_msg` messages, which must be joined with an `action_cnt` aggregate over the set of updates. Additionally, using the `accum` aggregate adds a point of order to the end of the dataflow, between `status` and `response_msg`. Although the accumulation of shopping actions is monotonic, summarization of the cart state requires us to ensure that there will be no further cart actions.

Comparing Figure 15 and Figure 17, we can see that the disorderly cart requires less coordination than the destructive cart: to ensure that the response to the client is deterministic and consistently replicated, we need to coordinate once per *session* (at checkout), rather than once per shopping action. This is analogous to the desired behavior in practice [13].

5.5 Discussion

Strictly monotonic programs are rare in practice, so adding some amount of coordination is often required to ensure consistency. In this running example we studied two candidate implementations of a simple distributed application with the aid of our program analysis. Both programs have points of order, but the analysis tool helped us reason about their relative coordination costs. Deciding that the disorderly approach is "better" required us to apply domain

knowledge: checkout is a coarser-grained coordination point than cart actions and their replication.

By providing the programmer with a set of abstractions that are predominantly order-independent, Bloom encourages a style of programming that minimizes coordination requirements. But as we see in the destructive cart program, it is nonetheless possible to use Bloom to write code in an imperative, order-sensitive style. Our analysis tools provide assistance in this regard. Given a particular implementation with points of order, Bloom's dataflow analysis can help a developer iteratively refine their program—either to "push back" the points to as late as possible in the dataflow, as we did in this example, or to "localize" points of order by moving them to locations in the program's dataflow where the coordination can be implemented on individual nodes without communication.

6. TOLERATING INCONSISTENCY

In the previous section we showed how to identify points of order: code locations that are sensitive to non-deterministic input ordering. We then demonstrated how to resolve the non-determinism by introducing coordination. However, in many cases adding additional coordination is undesirable due to concerns like latency and availability. In these cases, Bloom's point-of-order analysis can assist programmers with the task of *tolerating inconsistency*, rather than resolving it via coordination. A notable example of how to manage inconsistency is presented by Helland and Campbell, who reflect on their experience programming with patterns of "memories, guesses and apologies" [13]. We provide a sketch here of ideas for converting these patterns into developer tools in Bloom.

"Guesses"—facts that may not be true—may arise at the inputs to a program, e.g., from noisy sensors or untrusted software or users. But Helland and Campbell's use of the term corresponds in our analysis to unresolved points of order: non-monotonic logic that makes decisions without full knowledge of its input sets. We can rewrite the schemas of Bloom collections to include an additional attribute marking each fact as a "guarantee" or "guess," and automatically augment user code to propagate those labels through program logic in the manner of "taint checking" in program security [22, 25]. Moreover, by identifying unresolved points of order, we can identify when program logic derives "guesses" from "guarantees," and rewrite user code to label data appropriately. By rewriting programs to log guesses that cross interface boundaries, we can also implement Helland and Campbell's idea of "memories": a log of guesses that were sent outside the system.

Most of these patterns can be implemented as automatic program rewrites. We envision building a system that facilitates running low-latency, "guess"-driven decision making in the foreground, and expensive but consistent logic as a background process. When the background process detects an inconsistency in the results produced by the foreground system (e.g., because a "guess" turns out to be mistaken), it can then take corrective action by generating an "apology." Importantly, both of these subsystems are implementations of the same high-level design, except with different consistency and coordination requirements; hence, it should be possible to synthesize both variants of the program from the same source code. Throughout this process—making calculated "guesses," storing appropriate "memories," and generating the necessary "apologies"—we see significant opportunities to build scaffolding and tool support to lighten the burden on the programmer.

Finally, we hope to provide analysis techniques that can prove the consistency of the high-level workflow: i.e., prove that any combination of user behavior, background guess resolution, and apology logic will eventually lead to a consistent resolution of the business rules at both the user and system sides.

7. RELATED WORK

Systems with loose consistency requirements have been explored in depth by both the systems and database management communities (e.g., [6, 8, 9, 24]); we do not attempt to provide an exhaustive survey here. The shopping cart case study in Section 5 was motivated by the Amazon Dynamo paper [11], as well as the related discussion by Helland and Campbell [13].

The Bloom language is inspired by earlier work that attempts to integrate databases and programming languages. This includes early research such as Gem [27] and more recent object-relational mapping layers such as Ruby on Rails. Unlike these efforts, Bloom is targeted at the development of both distributed infrastructure and distributed applications, so it does not make any assumptions about the presence of a database system "underneath". Given our prototype implementation in Ruby, it is tempting to integrate Bud with Rails; we have left this for future work.

There is a long history of attempts to design programming languages more suitable to parallel and distributed systems; for example, Argus [15] and Linda [7]. Again, we do not hope to survey that literature here. More pragmatically, Erlang is an oft-cited choice for distributed programming in recent years. Erlang's features and design style encourage the use of asynchronous lightweight "actors." As mentioned previously, we did a simple Bloom prototype DSL in Erlang (which we cannot help but call "Bloomerlang"), and there is a natural correspondence between Bloom-style distributed rules and Erlang actors. However there is no requirement for Erlang programs to be written in the disorderly style of Bloom. It is not obvious that typical Erlang programs are significantly more amenable to a useful points-of-order analysis than programs written in any other functional language. For example, ordered lists are basic constructs in functional languages, and without program annotation or deeper analysis than we need to do in Bloom, any code that modifies lists would need be marked as a point of order, much like our destructive shopping cart. We believe that Bloom's "disorderly by default" style encourages order-independent programming, and we know that its roots in database theory helped produce a simple but useful program analysis technique. While we would be happy to see the analysis "ported" to other distributed programming environments, it may be that design patterns using Bloom-esque disorderly programming are the natural way to achieve this.

Our work on Bloom bears a resemblance to the Reactor language [5]. Both languages target distributed programming and are grounded in Datalog. Like many other rule languages including our earlier work on Overlog, Reactor updates mutable state in an operational step "outside Datalog" after each fixpoint computation. By contrast, Bloom is purely declarative: following Dedalus, it models updates as the logical derivation of immutable "versions" of collections over time. While Bloom uses a syntax inspired by object-oriented languages, Reactor takes a more explicitly agent-oriented approach. Reactor also includes synchronous coupling between agents as a primitive; we have opted to only include asynchronous communication as a language primitive and to provide synchronous coordination between nodes as a library.

Another recent language related to our work is Coherence [4], which also embraces "disorderly" programming. Unlike Bloom, Coherence is not targeted at distributed computing and is not based on logic programming.

8. CONCLUSION AND FUTURE WORK

In this paper we make three main contributions. First, we present the CALM principle, which connects the notion of eventual consistency in distributed programming to theoretical foundations in

database theory. Second, we show that we can bring that theory to bear on the practice of software development via "disorderly" programming patterns, complemented with automatic analysis techniques for identifying and managing a program's points of order in a principled way. Finally, we present our Bloom prototype as an example of a practically-minded disorderly and declarative programming language, with an initial implementation as a domain-specific language within Ruby.

We plan to extend the work described in this paper in several directions. First, we are building a more mature Bloom language environment, including a library of modules for distributed computing. We intend to compose those modules to implement a number of variants of distributed systems. The design of Bloom itself was motivated by our experience implementing scalable services and protocols in Overlog [1, 2], and this practice of system/language co-design continues to be part of our approach. Second, we hope to expand our suite of analysis techniques to address additional important properties in distributed systems, including idempotency and invertability of interfaces. Third, we are hopeful that the logic foundation of Bloom will enable us to develop better tools and techniques for the debugging and systematic testing of distributed systems under failure and security attacks, perhaps drawing on recent work on this topic [10, 17]. Finally, we are working to formally tighten our ideas connecting non-monotonic logic, distributed coordination, and consistency of distributed programs [14].

Acknowledgments

We would like to thank Ras Bodík, Kuang Chen, Haryadi Gunawi, Dmitriy Ryaboy, Russell Sears, and the anonymous reviewers for their helpful comments. This work was supported by NSF grants 0917349, 0803690, 0722077, 0713661 and 0435496, Air Force Office of Scientific Research award 22178970-41070-F, the Natural Sciences and Engineering Research Council of Canada, and gifts from Yahoo Research, IBM Research and Microsoft Research.

9. REFERENCES

[1] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. BOOM Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In *EuroSys*, 2010.

[2] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I Do Declare: Consensus in a Logic Language. *SIGOPS Oper. Syst. Rev.*, 43:25–30, January 2010.

[3] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in Time and Space. In *Proc. Datalog 2.0 Workshop (to appear)*, 2011.

[4] J. Edwards. Coherent Reaction. In *OOPSLA*, 2009.

[5] J. Field, M.-C. Marinescu, and C. Stefansen. Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications. *Theoretical Computer Science*, 410(2-3), February 2009.

[6] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, 1987.

[7] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, January 1985.

[8] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, 1989.

[9] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD*, 1996.

[10] H. S. Gunawi et al. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI (to appear)*, 2011.

[11] D. Hastorun et al. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, 2007.

[12] P. Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *CIDR*, 2007.

[13] P. Helland and D. Campbell. Building on Quicksand. In *CIDR*, 2009.

[14] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[15] B. Liskov. Distributed programming in Argus. *CACM*, 31:300–312, 1988.

[16] B. T. Loo et al. Implementing Declarative Overlays. In *SOSP*, 2005.

[17] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. Secureblox: customizable secure distributed data processing. In *SIGMOD*, 2010.

[18] D. Pritchett. BASE: An Acid Alternative. *ACM Queue*, 6(3):48–55, 2008.

[19] T. C. Przymusinski. *On the Declarative Semantics of Deductive Databases and Logic Programs*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[20] K. A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *PODS*, 1990.

[21] K. A. Ross. A syntactic stratification condition using constraints. In *International Symposium on Logic Programming*, pages 76–90, 1994.

[22] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *Selected Areas in Communications*, 21(1):5–19, 2003.

[23] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *ICDE*, 1986.

[24] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[25] S. Vandebogart et al. Labels and Event Processes in the Asbestos Operating System. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[26] W. Vogels. Eventually Consistent. *CACM*, 52(1):40–44, 2009.

[27] C. Zaniolo. The database language GEM. In *SIGMOD*, 1983.

APPENDIX

A. ADDITIONAL SOURCE CODE

Figures 18, 19 and 20 contain the remainder of the Bloom code used in this paper: best-effort protocols for unicast and multicast messaging, and the complete program for the replicated disorderly cart described in Section 5.

```
module BestEffortDelivery
  include DeliveryProtocol

  def state
    channel :pipe_chan,
      ['@dst', 'src', 'ident'], ['payload']
  end

  declare
  def snd
    pipe_chan <~ pipe_in
  end

  declare
  def done
    pipe_sent <= pipe_in
  end
end
```

Figure 18: Best-effort unicast messaging in Bloom.

```
module MulticastProtocol
  def state
    table :members, ['peer']
    interface input, :send_mcast, ['ident'], ['payload']
    interface output, :mcast_done, ['ident'], ['payload']
  end
end

module SimpleMulticast
  include MulticastProtocol
  include DeliveryProtocol

  declare
  def snd_mcast
    pipe_in <= join([send_mcast, members]).map do |s, m|
      [m.peer, @local_addr, s.ident, s.payload]
    end
  end

  declare
  def done_mcast
    mcast_done <= pipe_sent.map{|p| [p.ident, p.payload]}
  end
end
```

Figure 19: A simple unreliable multicast library in Bloom.

```
class ReplicatedDisorderlyCart < Bud
  include DisorderlyCart
  include SimpleMulticast
  include BestEffortDelivery

  declare
  def replicate
    send_mcast <= action_msg.map do |a|
      [a.reqid, [a.session, a.item, a.action, a.reqid]]
    end
    cart_action <= pipe_chan.map{|c| c.payload }
  end
end
```

Figure 20: The complete disorderly cart program.

Xen and the Art of Virtualization

Paul Barham*, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris,
Alex Ho, Rolf Neugebauer†, Ian Pratt, Andrew Warfield

University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, UK, CB3 0FD
{firstname.lastname}@cl.cam.ac.uk

ABSTRACT

Numerous systems have been designed which use virtualization to subdivide the ample resources of a modern computer. Some require specialized hardware, or cannot support commodity operating systems. Some target 100% binary compatibility at the expense of performance. Others sacrifice security or functionality for speed. Few offer resource isolation or performance guarantees; most provide only best-effort provisioning, risking denial of service.

This paper presents Xen, an x86 virtual machine monitor which allows multiple commodity operating systems to share conventional hardware in a safe and resource managed fashion, but without sacrificing either performance or functionality. This is achieved by providing an idealized virtual machine abstraction to which operating systems such as Linux, BSD and Windows XP, can be *ported* with minimal effort.

Our design is targeted at hosting up to 100 virtual machine instances simultaneously on a modern server. The virtualization approach taken by Xen is extremely efficient: we allow operating systems such as Linux and Windows XP to be hosted simultaneously for a negligible performance overhead — at most a few percent compared with the unvirtualized case. We considerably outperform competing commercial and freely available solutions in a range of microbenchmarks and system-wide tests.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management; D.4.2 [**Operating Systems**]: Storage Management; D.4.8 [**Operating Systems**]: Performance

General Terms

Design, Measurement, Performance

Keywords

Virtual Machine Monitors, Hypervisors, Paravirtualization

*Microsoft Research Cambridge, UK
†Intel Research Cambridge, UK

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.
Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

1. INTRODUCTION

Modern computers are sufficiently powerful to use virtualization to present the illusion of many smaller *virtual machines* (VMs), each running a separate operating system instance. This has led to a resurgence of interest in VM technology. In this paper we present Xen, a high performance resource-managed virtual machine monitor (VMM) which enables applications such as server consolidation [42, 8], co-located hosting facilities [14], distributed web services [43], secure computing platforms [12, 16] and application mobility [26, 37].

Successful partitioning of a machine to support the concurrent execution of multiple operating systems poses several challenges. Firstly, virtual machines must be isolated from one another: it is not acceptable for the execution of one to adversely affect the performance of another. This is particularly true when virtual machines are owned by mutually untrusting users. Secondly, it is necessary to support a variety of different operating systems to accommodate the heterogeneity of popular applications. Thirdly, the performance overhead introduced by virtualization should be small.

Xen hosts commodity operating systems, albeit with some source modifications. The prototype described and evaluated in this paper can support multiple concurrent instances of our XenoLinux guest operating system; each instance exports an application binary interface identical to a non-virtualized Linux 2.4. Our port of Windows XP to Xen is not yet complete but is capable of running simple user-space processes. Work is also progressing in porting NetBSD.

Xen enables users to dynamically instantiate an operating system to execute whatever they desire. In the XenoServer project [15, 35] we are deploying Xen on standard server hardware at economically strategic locations within ISPs or at Internet exchanges. We perform admission control when starting new virtual machines and expect each VM to *pay* in some fashion for the resources it requires. We discuss our ideas and approach in this direction elsewhere [21]; this paper focuses on the VMM.

There are a number of ways to build a system to host multiple applications and servers on a shared machine. Perhaps the simplest is to deploy one or more hosts running a standard operating system such as Linux or Windows, and then to allow users to install files and start processes — protection between applications being provided by conventional OS techniques. Experience shows that system administration can quickly become a time-consuming task due to complex configuration interactions between supposedly disjoint applications.

More importantly, such systems do not adequately support performance isolation; the scheduling priority, memory demand, network traffic and disk accesses of one process impact the performance of others. This may be acceptable when there is adequate provisioning and a closed user group (such as in the case of com-

putational grids, or the experimental PlanetLab platform [33]), but not when resources are oversubscribed, or users uncooperative.

One way to address this problem is to retrofit support for performance isolation to the operating system. This has been demonstrated to a greater or lesser degree with resource containers [3], Linux/RK [32], QLinux [40] and SILK [4]. One difficulty with such approaches is ensuring that *all* resource usage is accounted to the correct process — consider, for example, the complex interactions between applications due to buffer cache or page replacement algorithms. This is effectively the problem of "QoS crosstalk" [41] within the operating system. Performing multiplexing at a low level can mitigate this problem, as demonstrated by the Exokernel [23] and Nemesis [27] operating systems. Unintentional or undesired interactions between tasks are minimized.

We use this same basic approach to build Xen, which multiplexes physical resources at the granularity of an entire operating system and is able to provide performance isolation between them. In contrast to process-level multiplexing this also allows a range of guest operating systems to gracefully coexist rather than mandating a specific application binary interface. There is a price to pay for this flexibility — running a full OS is more heavyweight than running a process, both in terms of initialization (e.g. booting or resuming versus `fork` and `exec`), and in terms of resource consumption.

For our target of up to 100 hosted OS instances, we believe this price is worth paying; it allows individual users to run unmodified binaries, or collections of binaries, in a resource controlled fashion (for instance an Apache server along with a PostgreSQL backend). Furthermore it provides an extremely high level of flexibility since the user can dynamically create the precise execution environment their software requires. Unfortunate configuration interactions between various services and applications are avoided (for example, each Windows instance maintains its own registry).

The remainder of this paper is structured as follows: in Section 2 we explain our approach towards virtualization and outline how Xen works. Section 3 describes key aspects of our design and implementation. Section 4 uses industry standard benchmarks to evaluate the performance of XenoLinux running above Xen in comparison with stand-alone Linux, VMware Workstation and User-mode Linux (UML). Section 5 reviews related work, and finally Section 6 discusses future work and concludes.

2. XEN: APPROACH & OVERVIEW

In a traditional VMM the virtual hardware exposed is functionally identical to the underlying machine [38]. Although *full virtualization* has the obvious benefit of allowing unmodified operating systems to be hosted, it also has a number of drawbacks. This is particularly true for the prevalent IA-32, or *x86*, architecture.

Support for full virtualization was never part of the x86 architectural design. Certain supervisor instructions must be handled by the VMM for correct virtualization, but executing these with insufficient privilege fails silently rather than causing a convenient trap [36]. Efficiently virtualizing the x86 MMU is also difficult. These problems can be solved, but only at the cost of increased complexity and reduced performance. VMware's ESX Server [10] dynamically rewrites portions of the hosted machine code to insert traps wherever VMM intervention might be required. This translation is applied to the entire guest OS kernel (with associated translation, execution, and caching costs) since all non-trapping privileged instructions must be caught and handled. ESX Server implements shadow versions of system structures such as page tables and maintains consistency with the virtual tables by trapping every update attempt — this approach has a high cost for update-intensive operations such as creating a new application process.

Notwithstanding the intricacies of the x86, there are other arguments against full virtualization. In particular, there are situations in which it is desirable for the hosted operating systems to see real as well as virtual resources: providing both real and virtual time allows a guest OS to better support time-sensitive tasks, and to correctly handle TCP timeouts and RTT estimates, while exposing real machine addresses allows a guest OS to improve performance by using superpages [30] or page coloring [24].

We avoid the drawbacks of full virtualization by presenting a virtual machine abstraction that is similar but not identical to the underlying hardware — an approach which has been dubbed *paravirtualization* [43]. This promises improved performance, although it does require modifications to the guest operating system. It is important to note, however, that we do not require changes to the application binary interface (ABI), and hence no modifications are required to guest *applications*.

We distill the discussion so far into a set of design principles:

1. Support for unmodified application binaries is essential, or users will not transition to Xen. Hence we must virtualize all architectural features required by existing standard ABIs.

2. Supporting full multi-application operating systems is important, as this allows complex server configurations to be virtualized within a single guest OS instance.

3. Paravirtualization is necessary to obtain high performance and strong resource isolation on uncooperative machine architectures such as x86.

4. Even on cooperative machine architectures, completely hiding the effects of resource virtualization from guest OSes risks both correctness and performance.

Note that our paravirtualized x86 abstraction is quite different from that proposed by the recent Denali project [44]. Denali is designed to support thousands of virtual machines running network services, the vast majority of which are small-scale and unpopular. In contrast, Xen is intended to scale to approximately 100 virtual machines running industry standard applications and services. Given these very different goals, it is instructive to contrast Denali's design choices with our own principles.

Firstly, Denali does not target existing ABIs, and so can elide certain architectural features from their VM interface. For example, Denali does not fully support x86 segmentation although it is exported (and widely used[1]) in the ABIs of NetBSD, Linux, and Windows XP.

Secondly, the Denali implementation does not address the problem of supporting application multiplexing, nor multiple address spaces, within a single guest OS. Rather, applications are linked explicitly against an instance of the Ilwaco guest OS in a manner rather reminiscent of a libOS in the Exokernel [23]. Hence each virtual machine essentially hosts a single-user single-application unprotected "operating system". In Xen, by contrast, a single virtual machine hosts a real operating system which may itself securely multiplex thousands of unmodified user-level processes. Although a prototype *virtual MMU* has been developed which may help Denali in this area [44], we are unaware of any published technical details or evaluation.

Thirdly, in the Denali architecture the VMM performs all paging to and from disk. This is perhaps related to the lack of memory-management support at the virtualization layer. Paging within the

[1]For example, segments are frequently used by thread libraries to address thread-local data.

Memory Management	
Segmentation	Cannot install fully-privileged segment descriptors and cannot overlap with the top end of the linear address space.
Paging	Guest OS has direct read access to hardware page tables, but updates are batched and validated by the hypervisor. A domain may be allocated discontiguous machine pages.
CPU	
Protection	Guest OS must run at a lower privilege level than Xen.
Exceptions	Guest OS must register a descriptor table for exception handlers with Xen. Aside from page faults, the handlers remain the same.
System Calls	Guest OS may install a 'fast' handler for system calls, allowing direct calls from an application into its guest OS and avoiding indirecting through Xen on every call.
Interrupts	Hardware interrupts are replaced with a lightweight event system.
Time	Each guest OS has a timer interface and is aware of both 'real' and 'virtual' time.
Device I/O	
Network, Disk, etc.	Virtual devices are elegant and simple to access. Data is transferred using asynchronous I/O rings. An event mechanism replaces hardware interrupts for notifications.

Table 1: The paravirtualized x86 interface.

VMM is contrary to our goal of performance isolation: malicious virtual machines can encourage thrashing behaviour, unfairly depriving others of CPU time and disk bandwidth. In Xen we expect each guest OS to perform its own paging using its own guaranteed memory reservation and disk allocation (an idea previously exploited by self-paging [20]).

Finally, Denali virtualizes the 'namespaces' of all machine resources, taking the view that no VM can access the resource allocations of another VM if it cannot name them (for example, VMs have no knowledge of hardware addresses, only the virtual addresses created for them by Denali). In contrast, we believe that secure access control within the hypervisor is sufficient to ensure protection; furthermore, as discussed previously, there are strong correctness and performance arguments for making physical resources directly visible to guest OSes.

In the following section we describe the virtual machine abstraction exported by Xen and discuss how a guest OS must be modified to conform to this. Note that in this paper we reserve the term *guest operating system* to refer to one of the OSes that Xen can host and we use the term *domain* to refer to a running virtual machine within which a guest OS executes; the distinction is analogous to that between a *program* and a *process* in a conventional system. We call Xen itself the *hypervisor* since it operates at a higher privilege level than the supervisor code of the guest operating systems that it hosts.

2.1 The Virtual Machine Interface

Table 1 presents an overview of the paravirtualized x86 interface, factored into three broad aspects of the system: memory management, the CPU, and device I/O. In the following we address each machine subsystem in turn, and discuss how each is presented in our paravirtualized architecture. Note that although certain parts of our implementation, such as memory management, are specific to the x86, many aspects (such as our virtual CPU and I/O devices) can be readily applied to other machine architectures. Furthermore, x86 represents a *worst case* in the areas where it differs significantly from RISC-style processors — for example, efficiently virtualizing hardware page tables is more difficult than virtualizing a software-managed TLB.

2.1.1 Memory management

Virtualizing memory is undoubtedly the most difficult part of paravirtualizing an architecture, both in terms of the mechanisms required in the hypervisor and modifications required to port each guest OS. The task is easier if the architecture provides a software-managed TLB as these can be efficiently virtualized in a simple manner [13]. A tagged TLB is another useful feature supported by most server-class RISC architectures, including Alpha, MIPS and SPARC. Associating an address-space identifier tag with each TLB entry allows the hypervisor and each guest OS to efficiently coexist in separate address spaces because there is no need to flush the entire TLB when transferring execution.

Unfortunately, x86 does not have a software-managed TLB; instead TLB misses are serviced automatically by the processor by walking the page table structure in hardware. Thus to achieve the best possible performance, all valid page translations for the current address space should be present in the hardware-accessible page table. Moreover, because the TLB is not tagged, address space switches typically require a complete TLB flush. Given these limitations, we made two decisions: (i) guest OSes are responsible for allocating and managing the hardware page tables, with minimal involvement from Xen to ensure safety and isolation; and (ii) Xen exists in a 64MB section at the top of every address space, thus avoiding a TLB flush when entering and leaving the hypervisor.

Each time a guest OS requires a new page table, perhaps because a new process is being created, it allocates and initializes a page from its own memory reservation and registers it with Xen. At this point the OS must relinquish direct write privileges to the page-table memory: all subsequent updates must be validated by Xen. This restricts updates in a number of ways, including only allowing an OS to map pages that it owns, and disallowing writable mappings of page tables. Guest OSes may *batch* update requests to amortize the overhead of entering the hypervisor. The top 64MB region of each address space, which is reserved for Xen, is not accessible or remappable by guest OSes. This address region is not used by any of the common x86 ABIs however, so this restriction does not break application compatibility.

Segmentation is virtualized in a similar way, by validating updates to hardware segment descriptor tables. The only restrictions on x86 segment descriptors are: (i) they must have lower privilege than Xen, and (ii) they may not allow any access to the Xen-reserved portion of the address space.

2.1.2 CPU

Virtualizing the CPU has several implications for guest OSes. Principally, the insertion of a hypervisor below the operating system violates the usual assumption that the OS is the most privileged

entity in the system. In order to protect the hypervisor from OS misbehavior (and domains from one another) guest OSes must be modified to run at a lower privilege level.

Many processor architectures only provide two privilege levels. In these cases the guest OS would share the lower privilege level with applications. The guest OS would then protect itself by running in a separate address space from its applications, and indirectly pass control to and from applications via the hypervisor to set the virtual privilege level and change the current address space. Again, if the processor's TLB supports address-space tags then expensive TLB flushes can be avoided.

Efficient virtualizion of privilege levels is possible on x86 because it supports four distinct privilege levels in hardware. The x86 privilege levels are generally described as *rings*, and are numbered from zero (most privileged) to three (least privileged). OS code typically executes in ring 0 because no other ring can execute privileged instructions, while ring 3 is generally used for application code. To our knowledge, rings 1 and 2 have not been used by any well-known x86 OS since OS/2. Any OS which follows this common arrangement can be ported to Xen by modifying it to execute in ring 1. This prevents the guest OS from directly executing privileged instructions, yet it remains safely isolated from applications running in ring 3.

Privileged instructions are paravirtualized by requiring them to be validated and executed within Xen— this applies to operations such as installing a new page table, or yielding the processor when idle (rather than attempting to `hlt` it). Any guest OS attempt to directly execute a privileged instruction is failed by the processor, either silently or by taking a fault, since only Xen executes at a sufficiently privileged level.

Exceptions, including memory faults and software traps, are virtualized on x86 very straightforwardly. A table describing the handler for each type of exception is registered with Xen for validation. The handlers specified in this table are generally identical to those for real x86 hardware; this is possible because the exception stack frames are unmodified in our paravirtualized architecture. The sole modification is to the page fault handler, which would normally read the faulting address from a privileged processor register (`CR2`); since this is not possible, we write it into an extended stack frame[2]. When an exception occurs while executing outside ring 0, Xen's handler creates a copy of the exception stack frame on the guest OS stack and returns control to the appropriate registered handler.

Typically only two types of exception occur frequently enough to affect system performance: system calls (which are usually implemented via a software exception), and page faults. We improve the performance of system calls by allowing each guest OS to register a 'fast' exception handler which is accessed directly by the processor without indirecting via ring 0; this handler is validated before installing it in the hardware exception table. Unfortunately it is not possible to apply the same technique to the page fault handler because only code executing in ring 0 can read the faulting address from register `CR2`; page faults must therefore always be delivered via Xen so that this register value can be saved for access in ring 1.

Safety is ensured by validating exception handlers when they are presented to Xen. The only required check is that the handler's code segment does not specify execution in ring 0. Since no guest OS can create such a segment, it suffices to compare the specified segment selector to a small number of static values which are reserved by Xen. Apart from this, any other handler problems are fixed up during exception propagation — for example, if the handler's code segment is not present or if the handler is not paged into memory then an appropriate fault will be taken when Xen executes the `iret` instruction which returns to the handler. Xen detects these "double faults" by checking the faulting program counter value: if the address resides within the exception-virtualizing code then the offending guest OS is terminated.

Note that this "lazy" checking is safe even for the direct system-call handler: access faults will occur when the CPU attempts to directly jump to the guest OS handler. In this case the faulting address will be outside Xen (since Xen will never execute a guest OS system call) and so the fault is virtualized in the normal way. If propagation of the fault causes a further "double fault" then the guest OS is terminated as described above.

[2]In hindsight, writing the value into a pre-agreed shared memory location rather than modifying the stack frame would have simplified the XP port.

OS subsection	# lines	
	Linux	**XP**
Architecture-independent	78	1299
Virtual network driver	484	–
Virtual block-device driver	1070	–
Xen-specific (non-driver)	1363	3321
Total	**2995**	**4620**
(**Portion of total x86 code base**	**1.36%**	**0.04%**)

Table 2: The simplicity of porting commodity OSes to Xen. The cost metric is the number of lines of reasonably commented and formatted code which are modified or added compared with the original x86 code base (excluding device drivers).

2.1.3 Device I/O

Rather than emulating existing hardware devices, as is typically done in fully-virtualized environments, Xen exposes a set of clean and simple device abstractions. This allows us to design an interface that is both efficient and satisfies our requirements for protection and isolation. To this end, I/O data is transferred to and from each domain via Xen, using shared-memory, asynchronous buffer-descriptor rings. These provide a high-performance communication mechanism for passing buffer information vertically through the system, while allowing Xen to efficiently perform validation checks (for example, checking that buffers are contained within a domain's memory reservation).

Similar to hardware interrupts, Xen supports a lightweight event-delivery mechanism which is used for sending asynchronous notifications to a domain. These notifications are made by updating a bitmap of pending event types and, optionally, by calling an event handler specified by the guest OS. These callbacks can be 'held off' at the discretion of the guest OS — to avoid extra costs incurred by frequent wake-up notifications, for example.

2.2 The Cost of Porting an OS to Xen

Table 2 demonstrates the cost, in lines of code, of porting commodity operating systems to Xen's paravirtualized x86 environment. Note that our NetBSD port is at a very early stage, and hence we report no figures here. The XP port is more advanced, but still in progress; it can execute a number of user-space applications from a RAM disk, but it currently lacks any virtual I/O drivers. For this reason, figures for XP's virtual device drivers are not presented. However, as with Linux, we expect these drivers to be small and simple due to the idealized hardware abstraction presented by Xen.

Windows XP required a surprising number of modifications to its architecture independent OS code because it uses a variety of structures and unions for accessing page-table entries (PTEs). Each page-table access had to be separately modified, although some of

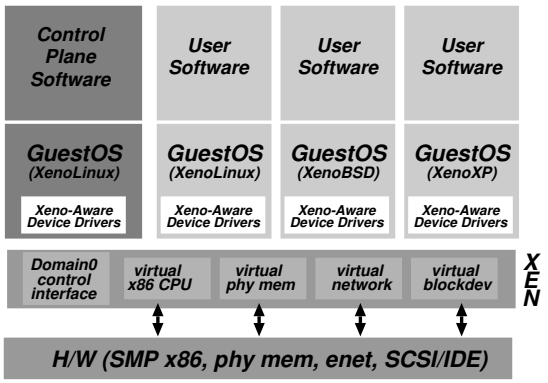


Figure 1: The structure of a machine running the Xen hypervisor, hosting a number of different guest operating systems, including *Domain0* running control software in a XenoLinux environment.

this process was automated with scripts. In contrast, Linux needed far fewer modifications to its generic memory system as it uses preprocessor macros to access PTEs — the macro definitions provide a convenient place to add the translation and hypervisor calls required by paravirtualization.

In both OSes, the architecture-specific sections are effectively a port of the x86 code to our paravirtualized architecture. This involved rewriting routines which used privileged instructions, and removing a large amount of low-level system initialization code. Again, more changes were required in Windows XP, mainly due to the presence of legacy 16-bit emulation code and the need for a somewhat different boot-loading mechanism. Note that the x86-specific code base in XP is substantially larger than in Linux and hence a larger porting effort should be expected.

2.3 Control and Management

Throughout the design and implementation of Xen, a goal has been to separate policy from mechanism wherever possible. Although the hypervisor must be involved in data-path aspects (for example, scheduling the CPU between domains, filtering network packets before transmission, or enforcing access control when reading data blocks), there is no need for it to be involved in, or even aware of, higher level issues such as how the CPU is to be shared, or which kinds of packet each domain may transmit.

The resulting architecture is one in which the hypervisor itself provides only basic control operations. These are exported through an interface accessible from authorized domains; potentially complex policy decisions, such as admission control, are best performed by management software running over a guest OS rather than in privileged hypervisor code.

The overall system structure is illustrated in Figure 1. Note that a domain is created at boot time which is permitted to use the *control interface*. This initial domain, termed *Domain0*, is responsible for hosting the application-level management software. The control interface provides the ability to create and terminate other domains and to control their associated scheduling parameters, physical memory allocations and the access they are given to the machine's physical disks and network devices.

In addition to processor and memory resources, the control interface supports the creation and deletion of virtual network interfaces (VIFs) and block devices (VBDs). These virtual I/O devices have associated access-control information which determines which domains can access them, and with what restrictions (for example, a read-only VBD may be created, or a VIF may filter IP packets to prevent source-address spoofing).

This control interface, together with profiling statistics on the current state of the system, is exported to a suite of application-level management software running in *Domain0*. This complement of administrative tools allows convenient management of the entire server: current tools can create and destroy domains, set network filters and routing rules, monitor per-domain network activity at packet and flow granularity, and create and delete virtual network interfaces and virtual block devices. We anticipate the development of higher-level tools to further automate the application of administrative policy.

3. DETAILED DESIGN

In this section we introduce the design of the major subsystems that make up a Xen-based server. In each case we present both Xen and guest OS functionality for clarity of exposition. The current discussion of guest OSes focuses on XenoLinux as this is the most mature; nonetheless our ongoing porting of Windows XP and NetBSD gives us confidence that Xen is guest OS agnostic.

3.1 Control Transfer: Hypercalls and Events

Two mechanisms exist for control interactions between Xen and an overlying domain: synchronous calls from a domain to Xen may be made using a *hypercall*, while notifications are delivered to domains from Xen using an asynchronous event mechanism.

The hypercall interface allows domains to perform a synchronous software trap into the hypervisor to perform a privileged operation, analogous to the use of system calls in conventional operating systems. An example use of a hypercall is to request a set of page-table updates, in which Xen validates and applies a list of updates, returning control to the calling domain when this is completed.

Communication from Xen to a domain is provided through an asynchronous event mechanism, which replaces the usual delivery mechanisms for device interrupts and allows lightweight notification of important events such as domain-termination requests. Akin to traditional Unix signals, there are only a small number of events, each acting to flag a particular type of occurrence. For instance, events are used to indicate that new data has been received over the network, or that a virtual disk request has completed.

Pending events are stored in a per-domain bitmask which is updated by Xen before invoking an event-callback handler specified by the guest OS. The callback handler is responsible for resetting the set of pending events, and responding to the notifications in an appropriate manner. A domain may explicitly defer event handling by setting a Xen-readable software flag: this is analogous to disabling interrupts on a real processor.

3.2 Data Transfer: I/O Rings

The presence of a hypervisor means there is an additional protection domain between guest OSes and I/O devices, so it is crucial that a data transfer mechanism be provided that allows data to move vertically through the system with as little overhead as possible.

Two main factors have shaped the design of our I/O-transfer mechanism: resource management and event notification. For resource accountability, we attempt to minimize the work required to demultiplex data to a specific domain when an interrupt is received from a device — the overhead of managing buffers is carried out later where computation may be accounted to the appropriate domain. Similarly, memory committed to device I/O is provided by the relevant domains wherever possible to prevent the crosstalk inherent in shared buffer pools; I/O buffers are protected during data transfer by pinning the underlying page frames within Xen.

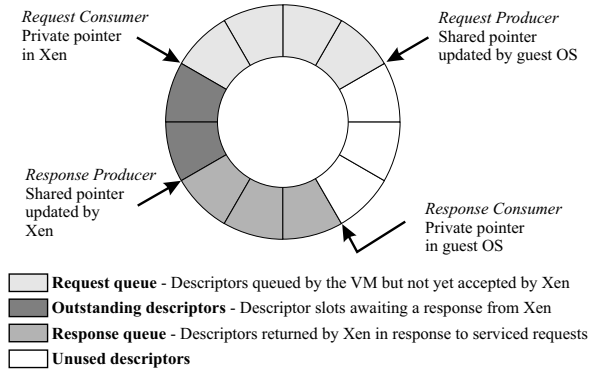


Figure 2: The structure of asynchronous I/O rings, which are used for data transfer between Xen and guest OSes.

Figure 2 shows the structure of our I/O descriptor rings. A ring is a circular queue of descriptors allocated by a domain but accessible from within Xen. Descriptors do not directly contain I/O data; instead, I/O data buffers are allocated out-of-band by the guest OS and indirectly referenced by I/O descriptors. Access to each ring is based around two pairs of producer-consumer pointers: domains place requests on a ring, advancing a request producer pointer, and Xen removes these requests for handling, advancing an associated request consumer pointer. Responses are placed back on the ring similarly, save with Xen as the producer and the guest OS as the consumer. There is no requirement that requests be processed in order: the guest OS associates a unique identifier with each request which is reproduced in the associated response. This allows Xen to unambiguously reorder I/O operations due to scheduling or priority considerations.

This structure is sufficiently generic to support a number of different device paradigms. For example, a set of 'requests' can provide buffers for network packet reception; subsequent 'responses' then signal the arrival of packets into these buffers. Reordering is useful when dealing with disk requests as it allows them to be scheduled within Xen for efficiency, and the use of descriptors with out-of-band buffers makes implementing zero-copy transfer easy.

We decouple the production of requests or responses from the notification of the other party: in the case of requests, a domain may enqueue multiple entries before invoking a hypercall to alert Xen; in the case of responses, a domain can defer delivery of a notification event by specifying a threshold number of responses. This allows each domain to trade-off latency and throughput requirements, similarly to the flow-aware interrupt dispatch in the ArseNIC Gigabit Ethernet interface [34].

3.3 Subsystem Virtualization

The control and data transfer mechanisms described are used in our virtualization of the various subsystems. In the following, we discuss how this virtualization is achieved for CPU, timers, memory, network and disk.

3.3.1 CPU scheduling

Xen currently schedules domains according to the Borrowed Virtual Time (BVT) scheduling algorithm [11]. We chose this particular algorithms since it is both work-conserving and has a special mechanism for low-latency wake-up (or *dispatch*) of a domain when it receives an event. Fast dispatch is particularly important to minimize the effect of virtualization on OS subsystems that are designed to run in a timely fashion; for example, TCP relies on the timely delivery of acknowledgments to correctly estimate network round-trip times. BVT provides low-latency dispatch by using virtual-time warping, a mechanism which temporarily violates 'ideal' fair sharing to favor recently-woken domains. However, other scheduling algorithms could be trivially implemented over our generic scheduler abstraction. Per-domain scheduling parameters can be adjusted by management software running in *Domain0*.

3.3.2 Time and timers

Xen provides guest OSes with notions of real time, virtual time and wall-clock time. Real time is expressed in nanoseconds passed since machine boot and is maintained to the accuracy of the processor's cycle counter and can be frequency-locked to an external time source (for example, via NTP). A domain's virtual time only advances while it is executing: this is typically used by the guest OS scheduler to ensure correct sharing of its timeslice between application processes. Finally, wall-clock time is specified as an offset to be added to the current real time. This allows the wall-clock time to be adjusted without affecting the forward progress of real time.

Each guest OS can program a pair of alarm timers, one for real time and the other for virtual time. Guest OSes are expected to maintain internal timer queues and use the Xen-provided alarm timers to trigger the earliest timeout. Timeouts are delivered using Xen's event mechanism.

3.3.3 Virtual address translation

As with other subsystems, Xen attempts to virtualize memory access with as little overhead as possible. As discussed in Section 2.1.1, this goal is made somewhat more difficult by the x86 architecture's use of hardware page tables. The approach taken by VMware is to provide each guest OS with a virtual page table, not visible to the memory-management unit (MMU) [10]. The hypervisor is then responsible for trapping accesses to the virtual page table, validating updates, and propagating changes back and forth between it and the MMU-visible 'shadow' page table. This greatly increases the cost of certain guest OS operations, such as creating new virtual address spaces, and requires explicit propagation of hardware updates to 'accessed' and 'dirty' bits.

Although full virtualization forces the use of shadow page tables, to give the illusion of contiguous physical memory, Xen is not so constrained. Indeed, Xen need only be involved in page table *updates*, to prevent guest OSes from making unacceptable changes. Thus we avoid the overhead and additional complexity associated with the use of shadow page tables — the approach in Xen is to register guest OS page tables directly with the MMU, and restrict guest OSes to read-only access. Page table updates are passed to Xen via a hypercall; to ensure safety, requests are *validated* before being applied.

To aid validation, we associate a type and reference count with each machine page frame. A frame may have any one of the following mutually-exclusive types at any point in time: page directory (PD), page table (PT), local descriptor table (LDT), global descriptor table (GDT), or writable (RW). Note that a guest OS may always create readable mappings to its own page frames, regardless of their current types. A frame may only safely be retasked when its reference count is zero. This mechanism is used to maintain the invariants required for safety; for example, a domain cannot have a writable mapping to any part of a page table as this would require the frame concerned to simultaneously be of types PT and RW.

The type system is also used to track which frames have already been validated for use in page tables. To this end, guest OSes indicate when a frame is allocated for page-table use — this requires a one-off validation of every entry in the frame by Xen, after which

its type is pinned to PD or PT as appropriate, until a subsequent unpin request from the guest OS. This is particularly useful when changing the page table base pointer, as it obviates the need to validate the new page table on every context switch. Note that a frame cannot be retasked until it is both unpinned and its reference count has reduced to zero – this prevents guest OSes from using unpin requests to circumvent the reference-counting mechanism.

To minimize the number of hypercalls required, guest OSes can locally queue updates before applying an entire batch with a single hypercall — this is particularly beneficial when creating new address spaces. However we must ensure that updates are committed early enough to guarantee correctness. Fortunately, a guest OS will typically execute a TLB flush before the first use of a new mapping: this ensures that any cached translation is invalidated. Hence, committing pending updates immediately before a TLB flush usually suffices for correctness. However, some guest OSes elide the flush when it is certain that no stale entry exists in the TLB. In this case it is possible that the first attempted use of the new mapping will cause a page-not-present fault. Hence the guest OS fault handler must check for outstanding updates; if any are found then they are flushed and the faulting instruction is retried.

3.3.4 Physical memory

The initial memory allocation, or *reservation*, for each domain is specified at the time of its creation; memory is thus statically partitioned between domains, providing strong isolation. A maximum-allowable reservation may also be specified: if memory pressure within a domain increases, it may then attempt to claim additional memory pages from Xen, up to this reservation limit. Conversely, if a domain wishes to save resources, perhaps to avoid incurring unnecessary costs, it can reduce its memory reservation by releasing memory pages back to Xen.

XenoLinux implements a *balloon driver* [42], which adjusts a domain's memory usage by passing memory pages back and forth between Xen and XenoLinux's page allocator. Although we could modify Linux's memory-management routines directly, the balloon driver makes adjustments by using existing OS functions, thus simplifying the Linux porting effort. However, paravirtualization can be used to extend the capabilities of the balloon driver; for example, the out-of-memory handling mechanism in the guest OS can be modified to automatically alleviate memory pressure by requesting more memory from Xen.

Most operating systems assume that memory comprises at most a few large contiguous extents. Because Xen does not guarantee to allocate contiguous regions of memory, guest OSes will typically create for themselves the illusion of contiguous *physical memory*, even though their underlying allocation of *hardware memory* is sparse. Mapping from physical to hardware addresses is entirely the responsibility of the guest OS, which can simply maintain an array indexed by physical page frame number. Xen supports efficient hardware-to-physical mapping by providing a shared translation array that is directly readable by all domains – updates to this array are validated by Xen to ensure that the OS concerned owns the relevant hardware page frames.

Note that even if a guest OS chooses to ignore hardware addresses in most cases, it must use the translation tables when accessing its page tables (which necessarily use hardware addresses). Hardware addresses may also be exposed to limited parts of the OS's memory-management system to optimize memory access. For example, a guest OS might allocate particular hardware pages so as to optimize placement within a physically indexed cache [24], or map naturally aligned contiguous portions of hardware memory using superpages [30].

3.3.5 Network

Xen provides the abstraction of a virtual firewall-router (VFR), where each domain has one or more network interfaces (VIFs) logically attached to the VFR. A VIF looks somewhat like a modern network interface card: there are two I/O rings of buffer descriptors, one for transmit and one for receive. Each direction also has a list of associated rules of the form (<*pattern*>, <*action*>) — if the *pattern* matches then the associated *action* is applied.

Domain0 is responsible for inserting and removing rules. In typical cases, rules will be installed to prevent IP source address spoofing, and to ensure correct demultiplexing based on destination IP address and port. Rules may also be associated with hardware interfaces on the VFR. In particular, we may install rules to perform traditional firewalling functions such as preventing incoming connection attempts on insecure ports.

To transmit a packet, the guest OS simply enqueues a buffer descriptor onto the transmit ring. Xen copies the descriptor and, to ensure safety, then copies the packet header and executes any matching filter rules. The packet payload is not copied since we use scatter-gather DMA; however note that the relevant page frames must be pinned until transmission is complete. To ensure fairness, Xen implements a simple round-robin packet scheduler.

To efficiently implement packet reception, we require the guest OS to exchange an unused page frame for each packet it receives; this avoids the need to copy the packet between Xen and the guest OS, although it requires that page-aligned receive buffers be queued at the network interface. When a packet is received, Xen immediately checks the set of receive rules to determine the destination VIF, and exchanges the packet buffer for a page frame on the relevant receive ring. If no frame is available, the packet is dropped.

3.3.6 Disk

Only *Domain0* has direct unchecked access to physical (IDE and SCSI) disks. All other domains access persistent storage through the abstraction of virtual block devices (VBDs), which are created and configured by management software running within *Domain0*. Allowing *Domain0* to manage the VBDs keeps the mechanisms within Xen very simple and avoids more intricate solutions such as the UDFs used by the Exokernel [23].

A VBD comprises a list of extents with associated ownership and access control information, and is accessed via the I/O ring mechanism. A typical guest OS disk scheduling algorithm will reorder requests prior to enqueuing them on the ring in an attempt to reduce response time, and to apply differentiated service (for example, it may choose to aggressively schedule synchronous metadata requests at the expense of speculative readahead requests). However, because Xen has more complete knowledge of the actual disk layout, we also support reordering within Xen, and so responses may be returned out of order. A VBD thus appears to the guest OS somewhat like a SCSI disk.

A translation table is maintained within the hypervisor for each VBD; the entries within this table are installed and managed by *Domain0* via a privileged control interface. On receiving a disk request, Xen inspects the VBD identifier and offset and produces the corresponding sector address and physical device. Permission checks also take place at this time. Zero-copy data transfer takes place using DMA between the disk and pinned memory pages in the requesting domain.

Xen services *batches* of requests from competing domains in a simple round-robin fashion; these are then passed to a standard elevator scheduler before reaching the disk hardware. Domains may explicitly pass down *reorder barriers* to prevent reordering when this is necessary to maintain higher level semantics (e.g. when us-

ing a write-ahead log). The low-level scheduling gives us good throughput, while the batching of requests provides reasonably fair access. Future work will investigate providing more predictable isolation and differentiated service, perhaps using existing techniques and schedulers [39].

3.4 Building a New Domain

The task of building the initial guest OS structures for a new domain is mostly delegated to *Domain0* which uses its privileged control interfaces (Section 2.3) to access the new domain's memory and inform Xen of initial register state. This approach has a number of advantages compared with building a domain entirely within Xen, including reduced hypervisor complexity and improved robustness (accesses to the privileged interface are sanity checked which allowed us to catch many bugs during initial development).

Most important, however, is the ease with which the building process can be extended and specialized to cope with new guest OSes. For example, the boot-time address space assumed by the Linux kernel is considerably simpler than that expected by Windows XP. It would be possible to specify a fixed initial memory layout for all guest OSes, but this would require additional bootstrap code within every guest OS to lay things out as required by the rest of the OS. Unfortunately this type of code is tricky to implement correctly; for simplicity and robustness it is therefore better to implement it within *Domain0* which can provide much richer diagnostics and debugging support than a bootstrap environment.

4. EVALUATION

In this section we present a thorough performance evaluation of Xen. We begin by benchmarking Xen against a number of alternative virtualization techniques, then compare the total system throughput executing multiple applications concurrently on a single native operating system against running each application in its own virtual machine. We then evaluate the performance isolation Xen provides between guest OSes, and assess the total overhead of running large numbers of operating systems on the same hardware. For these measurements, we have used our XenoLinux port (based on Linux 2.4.21) as this is our most mature guest OS. We expect the relative overheads for our Windows XP and NetBSD ports to be similar but have yet to conduct a full evaluation.

There are a number of preexisting solutions for running multiple copies of Linux on the same machine. VMware offers several commercial products that provide virtual x86 machines on which unmodified copies of Linux may be booted. The most commonly used version is VMware Workstation, which consists of a set of privileged kernel extensions to a 'host' operating system. Both Windows and Linux hosts are supported. VMware also offer an enhanced product called ESX Server which replaces the host OS with a dedicated kernel. By doing so, it gains some performance benefit over the workstation product. ESX Server also supports a paravirtualized interface to the network that can be accessed by installing a special device driver (vmxnet) into the guest OS, where deployment circumstances permit.

We have subjected ESX Server to the benchmark suites described below, but sadly are prevented from reporting quantitative results due to the terms of the product's End User License Agreement. Instead we present results from VMware Workstation 3.2, running on top of a Linux host OS, as it is the most recent VMware product without that benchmark publication restriction. ESX Server takes advantage of its native architecture to equal or outperform VMware Workstation and its hosted architecture. While Xen of course requires guest OSes to be ported, it takes advantage of paravirtualization to noticeably outperform ESX Server.

We also present results for User-mode Linux (UML), an increasingly popular platform for virtual hosting. UML is a port of Linux to run as a user-space process on a Linux host. Like XenoLinux, the changes required are restricted to the architecture dependent code base. However, the UML code bears little similarity to the native x86 port due to the very different nature of the execution environments. Although UML can run on an unmodified Linux host, we present results for the 'Single Kernel Address Space' (skas3) variant that exploits patches to the host OS to improve performance.

We also investigated three other virtualization techniques for running ported versions of Linux on the same x86 machine. Connectix's Virtual PC and forthcoming Virtual Server products (now acquired by Microsoft) are similar in design to VMware's, providing full x86 virtualization. Since all versions of Virtual PC have benchmarking restrictions in their license agreements we did not subject them to closer analysis. UMLinux is similar in concept to UML but is a different code base and has yet to achieve the same level of performance, so we omit the results. Work to improve the performance of UMLinux through host OS modifications is ongoing [25]. Although Plex86 was originally a general purpose x86 VMM, it has now been retargeted to support just Linux guest OSes. The guest OS must be specially compiled to run on Plex86, but the source changes from native x86 are trivial. The performance of Plex86 is currently well below the other techniques.

All the experiments were performed on a Dell 2650 dual processor 2.4GHz Xeon server with 2GB RAM, a Broadcom Tigon 3 Gigabit Ethernet NIC, and a single Hitachi DK32EJ 146GB 10k RPM SCSI disk. Linux version 2.4.21 was used throughout, compiled for architecture *i686* for the native and VMware guest OS experiments, for *xeno-i686* when running on Xen, and architecture *um* when running on UML. The Xeon processors in the machine support SMT ("hyperthreading"), but this was disabled because none of the kernels currently have SMT-aware schedulers. We ensured that the total amount of memory available to all guest OSes plus their VMM was equal to the total amount available to native Linux.

The RedHat 7.2 distribution was used throughout, installed on ext3 file systems. The VMs were configured to use the same disk partitions in 'persistent raw mode', which yielded the best performance. Using the same file system image also eliminated potential differences in disk seek times and transfer rates.

4.1 Relative Performance

We have performed a battery of experiments in order to evaluate the overhead of the various virtualization techniques relative to running on the 'bare metal'. Complex application-level benchmarks that exercise the whole system have been employed to characterize performance under a range of server-type workloads. Since neither Xen nor any of the VMware products currently support multiprocessor guest OSes (although they are themselves both SMP capable), the test machine was configured with one CPU for these experiments; we examine performance with concurrent guest OSes later. The results presented are the median of seven trials.

The first cluster of bars in Figure 3 represents a relatively easy scenario for the VMMs. The SPEC CPU suite contains a series of long-running computationally-intensive applications intended to measure the performance of a system's processor, memory system, and compiler quality. The suite performs little I/O and has little interaction with the OS. With almost all CPU time spent executing in user-space code, all three VMMs exhibit low overhead.

The next set of bars show the total elapsed time taken to build a default configuration of the Linux 2.4.21 kernel on a local ext3 file system with gcc 2.96. Native Linux spends about 7% of the CPU time in the OS, mainly performing file I/O, scheduling and

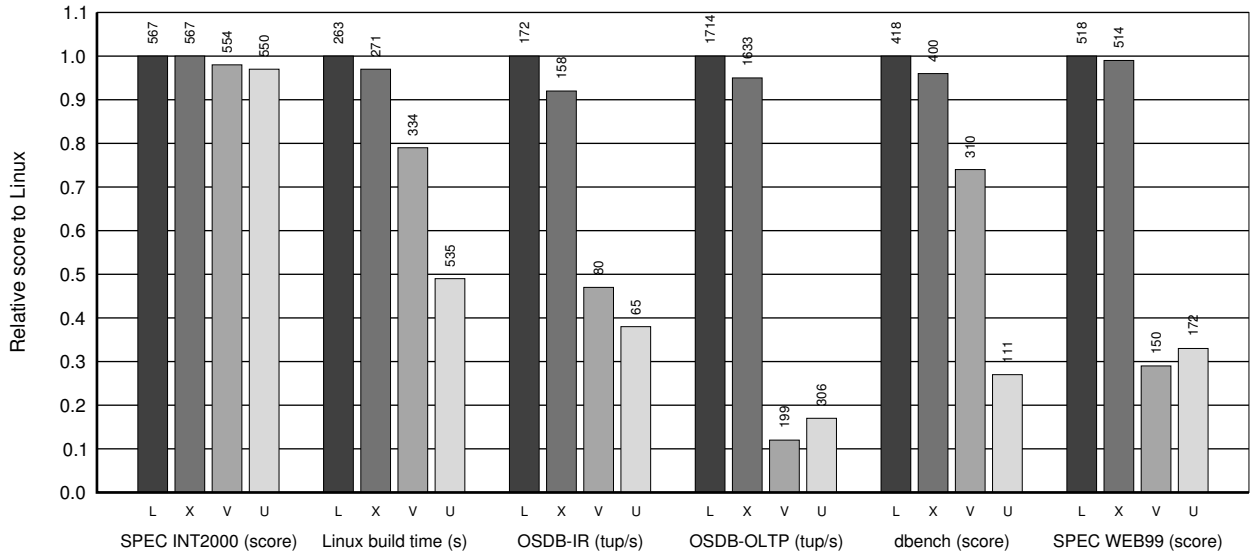


Figure 3: Relative performance of native Linux (L), XenoLinux (X), VMware workstation 3.2 (V) and User-Mode Linux (U).

memory management. In the case of the VMMs, this 'system time' is expanded to a greater or lesser degree: whereas Xen incurs a mere 3% overhead, the other VMMs experience a more significant slowdown.

Two experiments were performed using the PostgreSQL 7.1.3 database, exercised by the Open Source Database Benchmark suite (OSDB) in its default configuration. We present results for the multi-user Information Retrieval (IR) and On-Line Transaction Processing (OLTP) workloads, both measured in tuples per second. A small modification to the suite's test harness was required to produce correct results, due to a UML bug which loses virtual-timer interrupts under high load. The benchmark drives the database via PostgreSQL's native API (callable SQL) over a Unix domain socket. PostgreSQL places considerable load on the operating system, and this is reflected in the substantial virtualization overheads experienced by VMware and UML. In particular, the OLTP benchmark requires many synchronous disk operations, resulting in many protection domain transitions.

The `dbench` program is a file system benchmark derived from the industry-standard 'NetBench'. It emulates the load placed on a file server by Windows 95 clients. Here, we examine the throughput experienced by a single client performing around 90,000 file system operations.

SPEC WEB99 is a complex application-level benchmark for evaluating web servers and the systems that host them. The workload is a complex mix of page requests: 30% require dynamic content generation, 16% are HTTP POST operations and 0.5% execute a CGI script. As the server runs it generates access and POST logs, so the disk workload is not solely read-only. Measurements therefore reflect general OS performance, including file system and network, in addition to the web server itself.

A number of client machines are used to generate load for the server under test, with each machine simulating a collection of users concurrently accessing the web site. The benchmark is run repeatedly with different numbers of simulated users to determine the maximum number that can be supported. SPEC WEB99 defines a minimum Quality of Service that simulated users must receive in order to be 'conformant' and hence count toward the score: users must receive an aggregate bandwidth in excess of 320Kb/s over a series of requests. A warm-up phase is allowed in which the number of simultaneous clients is slowly increased, allowing servers to preload their buffer caches.

For our experimental setup we used the Apache HTTP server version 1.3.27, installing the `modspecweb99` plug-in to perform most but not all of the dynamic content generation — SPEC rules require 0.5% of requests to use full CGI, forking a separate process. Better absolute performance numbers can be achieved with the assistance of "TUX", the Linux in-kernel static content web server, but we chose not to use this as we felt it was less likely to be representative of our real-world target applications. Furthermore, although Xen's performance improves when using TUX, VMware suffers badly due to the increased proportion of time spent emulating ring 0 while executing the guest OS kernel.

SPEC WEB99 exercises the whole system. During the measurement period there is up to 180Mb/s of TCP network traffic and considerable disk read-write activity on a 2GB dataset. The benchmark is CPU-bound, and a significant proportion of the time is spent within the guest OS kernel, performing network stack processing, file system operations, and scheduling between the many `httpd` processes that Apache needs to handle the offered load. XenoLinux fares well, achieving within 1% of native Linux performance. VMware and UML both struggle, supporting less than a third of the number of clients of the native Linux system.

4.2 Operating System Benchmarks

To more precisely measure the areas of overhead within Xen and the other VMMs, we performed a number of smaller experiments targeting particular subsystems. We examined the overhead of virtualization as measured by McVoy's *lmbench* program [29]. We used version 3.0-a3 as this addresses many of the issues regarding the fidelity of the tool raised by Seltzer's *hbench* [6]. The OS performance subset of the lmbench suite consist of 37 microbenchmarks. In the native Linux case, we present figures for both uniprocessor (L-UP) and SMP (L-SMP) kernels as we were somewhat surprised by the performance overhead incurred by the extra locking in the SMP system in many cases.

Config	null call	null I/O	stat	open close	slct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
L-SMP	0.53	0.81	2.10	3.51	23.2	0.83	2.94	143	601	4k2
L-UP	0.45	0.50	1.28	1.92	5.70	0.68	2.49	110	530	4k0
Xen	0.46	0.50	1.22	1.88	5.69	0.69	1.75	**198**	**768**	**4k8**
VMW	0.73	0.83	1.88	2.99	11.1	1.02	4.63	874	2k3	10k
UML	24.7	25.1	36.1	62.8	39.9	26.0	46.0	21k	33k	58k

Table 3: `lmbench`: Processes - times in μs

Config	2p 0K	2p 16K	2p 64K	8p 16K	8p 64K	16p 16K	16p 64K
L-SMP	1.69	1.88	2.03	2.36	26.8	4.79	38.4
L-UP	0.77	0.91	1.06	1.03	24.3	3.61	37.6
Xen	**1.97**	**2.22**	**2.67**	**3.07**	**28.7**	**7.08**	39.4
VMW	18.1	17.6	21.3	22.4	51.6	41.7	72.2
UML	15.5	14.6	14.4	16.3	36.8	23.6	52.0

Table 4: `lmbench`: Context switching times in μs

Config	0K File create	0K File delete	10K File create	10K File delete	Mmap lat	Prot fault	Page fault
L-SMP	44.9	24.2	123	45.2	99.0	1.33	1.88
L-UP	32.1	6.08	66.0	12.5	68.0	1.06	1.42
Xen	32.5	5.86	68.2	13.6	**139**	1.40	**2.73**
VMW	35.3	9.3	85.6	21.4	620	7.53	12.4
UML	130	65.7	250	113	1k4	21.8	26.3

Table 5: `lmbench`: File & VM system latencies in μs

In 24 of the 37 microbenchmarks, XenoLinux performs similarly to native Linux, tracking the uniprocessor Linux kernel performance closely and outperforming the SMP kernel. In Tables 3 to 5 we show results which exhibit interesting performance variations among the test systems; particularly large penalties for Xen are shown in bold face.

In the process microbenchmarks (Table 3), Xen exhibits slower *fork*, *exec* and *sh* performance than native Linux. This is expected, since these operations require large numbers of page table updates which must all be verified by Xen. However, the paravirtualization approach allows XenoLinux to batch update requests. Creating new page tables presents an ideal case: because there is no reason to commit pending updates sooner, XenoLinux can amortize each hypercall across 2048 updates (the maximum size of its batch buffer). Hence each update hypercall constructs 8MB of address space.

Table 4 shows context switch times between different numbers of processes with different working set sizes. Xen incurs an extra overhead between 1μs and 3μs, as it executes a hypercall to change the page table base. However, context switch results for larger working set sizes (perhaps more representative of real applications) show that the overhead is small compared with cache effects. Unusually, VMware Workstation is inferior to UML on these microbenchmarks; however, this is one area where enhancements in ESX Server are able to reduce the overhead.

The *mmap latency* and *page fault latency* results shown in Table 5 are interesting since they require two transitions into Xen per page: one to take the hardware fault and pass the details to the guest OS, and a second to install the updated page table entry on the guest OS's behalf. Despite this, the overhead is relatively modest.

One small anomaly in Table 3 is that XenoLinux has lower signal-handling latency than native Linux. This benchmark does not require any calls into Xen at all, and the 0.75μs (30%) speedup is presumably due to a fortuitous cache alignment in XenoLinux, hence underlining the dangers of taking microbenchmarks too seriously.

	TCP MTU 1500 TX	TCP MTU 1500 RX	TCP MTU 500 TX	TCP MTU 500 RX
Linux	897	897	602	544
Xen	897 (-0%)	897 (-0%)	516 (-14%)	467 (-14%)
VMW	291 (-68%)	615 (-31%)	101 (-83%)	137 (-75%)
UML	165 (-82%)	203 (-77%)	61.1(-90%)	91.4(-83%)

Table 6: `ttcp`: Bandwidth in Mb/s

4.2.1 Network performance

In order to evaluate the overhead of virtualizing the network, we examine TCP performance over a Gigabit Ethernet LAN. In all experiments we use a similarly-configured SMP box running native Linux as one of the endpoints. This enables us to measure receive and transmit performance independently. The *ttcp* benchmark was used to perform these measurements. Both sender and receiver applications were configured with a socket buffer size of 128kB, as we found this gave best performance for all tested systems. The results presented are a median of 9 experiments transferring 400MB.

Table 6 presents two sets of results, one using the default Ethernet MTU of 1500 bytes, the other using a 500-byte MTU (chosen as it is commonly used by dial-up PPP clients). The results demonstrate that the page-flipping technique employed by the XenoLinux virtual network driver avoids the overhead of data copying and hence achieves a very low per-byte overhead. With an MTU of 500 bytes, the per-packet overheads dominate. The extra complexity of transmit firewalling and receive demultiplexing adversely impact the throughput, but only by 14%.

VMware emulate a 'pcnet32' network card for communicating with the guest OS which provides a relatively clean DMA-based interface. ESX Server also supports a special 'vmxnet' driver for compatible guest OSes, which provides significant networking performance improvements.

4.3 Concurrent Virtual Machines

In this section, we compare the performance of running multiple applications in their own guest OS against running them on the same native operating system. Our focus is on the results using Xen, but we comment on the performance of the other VMMs where applicable.

Figure 4 shows the results of running 1, 2, 4, 8 and 16 copies of the SPEC WEB99 benchmark in parallel on a two CPU machine. The native Linux was configured for SMP; on it we ran multiple copies of Apache as concurrent processes. In Xen's case, each instance of SPEC WEB99 was run in its own uniprocessor Linux guest OS (along with an `sshd` and other management processes). Different TCP port numbers were used for each web server to enable the copies to be run in parallel. Note that the size of the SPEC data set required for c simultaneous connections is $(25 + (c \times 0.66)) \times 4.88$ MBytes or approximately 3.3GB for 1000 connections. This is sufficiently large to thoroughly exercise the disk and buffer cache subsystems.

Achieving good SPEC WEB99 scores requires both high throughput and bounded latency: for example, if a client request gets stalled due to a badly delayed disk read, then the connection will be classed as non conforming and won't contribute to the score. Hence, it is important that the VMM schedules domains in a timely fashion. By default, Xen uses a 5ms time slice.

In the case of a single Apache instance, the addition of a sec-

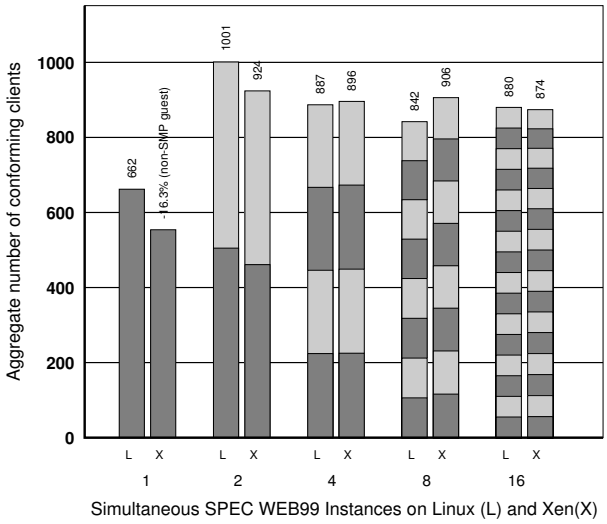


Figure 4: SPEC WEB99 for 1, 2, 4, 8 and 16 concurrent Apache servers: higher values are better.

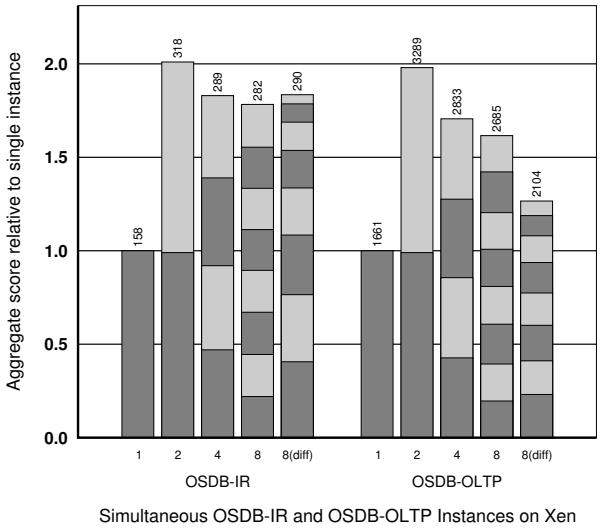


Figure 5: Performance of multiple instances of PostgreSQL running OSDB in separate Xen domains. 8(diff) bars show performance variation with different scheduler weights.

ond CPU enables native Linux to improve on the score reported in section 4.1 by 28%, to 662 conformant clients. However, the best aggregate throughput is achieved when running two Apache instances, suggesting that Apache 1.3.27 may have some SMP scalability issues.

When running a single domain, Xen is hampered by a lack of support for SMP guest OSes. However, Xen's interrupt load balancer identifies the idle CPU and diverts all interrupt processing to it, improving on the single CPU score by 9%. As the number of domains increases, Xen's performance improves to within a few percent of the native case.

Next we performed a series of experiments running multiple instances of PostgreSQL exercised by the OSDB suite. Running multiple PostgreSQL instances on a single Linux OS proved difficult, as it is typical to run a single PostgreSQL instance supporting multiple databases. However, this would prevent different users having separate database configurations. We resorted to a combination of `chroot` and software patches to avoid SysV IPC name-space clashes between different PostgreSQL instances. In contrast, Xen allows each instance to be started in its own domain allowing easy configuration.

In Figure 5 we show the aggregate throughput Xen achieves when running 1, 2, 4 and 8 instances of OSDB-IR and OSDB-OLTP. When a second domain is added, full utilization of the second CPU almost doubles the total throughput. Increasing the number of domains further causes some reduction in aggregate throughput which can be attributed to increased context switching and disk head movement. Aggregate scores running multiple PostgreSQL instances on a single Linux OS are 25-35% *lower* than the equivalent scores using Xen. The cause is not fully understood, but it appears that PostgreSQL has SMP scalability issues combined with poor utilization of Linux's block cache.

Figure 5 also demonstrates performance differentiation between 8 domains. Xen's schedulers were configured to give each domain an integer weight between 1 and 8. The resulting throughput scores for each domain are reflected in the different banding on the bar. In the IR benchmark, the weighting has a precise influence over throughput and each segment is within 4% of its expected size. However, in the OLTP case, domains given a larger share of resources to not achieve proportionally higher scores: The high level of synchronous disk activity highlights a weakness in our current disk scheduling algorithm causing them to under-perform.

4.4 Performance Isolation

In order to demonstrate the performance isolation provided by Xen, we hoped to perform a "bakeoff" between Xen and other OS-based implementations of performance isolation techniques such as resource containers. However, at the current time there appear to be no implementations based on Linux 2.4 available for download. QLinux 2.4 has yet to be released and is targeted at providing QoS for multimedia applications rather than providing full defensive isolation in a server environment. Ensim's Linux-based Private Virtual Server product appears to be the most complete implementation, reportedly encompassing control of CPU, disk, network and physical memory resources [14]. We are in discussions with Ensim and hope to be able to report results of a comparative evaluation at a later date.

In the absence of a side-by-side comparison, we present results showing that Xen's performance isolation works as expected, even in the presence of a malicious workload. We ran 4 domains configured with equal resource allocations, with two domains running previously-measured workloads (PostgreSQL/OSDB-IR and SPEC WEB99), and two domains each running a pair of extremely antisocial processes. The third domain concurrently ran a disk bandwidth hog (sustained `dd`) together with a file system intensive workload targeting huge numbers of small file creations within large directories. The fourth domain ran a 'fork bomb' at the same time as a virtual memory intensive application which attempted to allocate and touch 3GB of virtual memory and, on failure, freed every page and then restarted.

We found that both the OSDB-IR and SPEC WEB99 results were only marginally affected by the behaviour of the two domains running disruptive processes — respectively achieving 4% and 2% below the results reported earlier. We attribute this to the overhead of extra context switches and cache effects. We regard this as

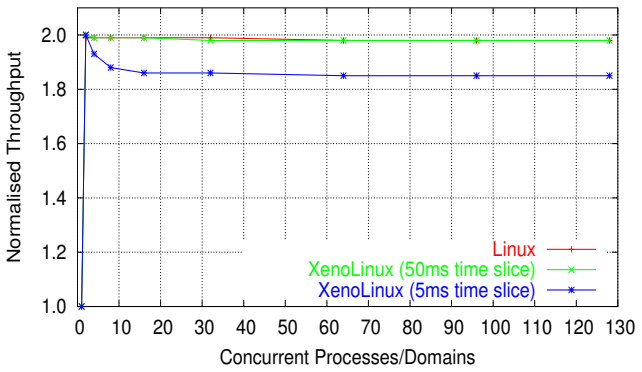


Figure 6: Normalized aggregate performance of a subset of SPEC CINT2000 running concurrently on 1-128 domains

somewhat fortuitous in light of our current relatively simple disk scheduling algorithm, but under this scenario it appeared to provide sufficient isolation from the page-swapping and disk-intensive activities of the other domains for the benchmarks to make good progress. VMware Workstation achieves similar levels of isolation, but at reduced levels of absolute performance.

We repeated the same experiment under native Linux. Unsurprisingly, the disruptive processes rendered the machine completely unusable for the two benchmark processes, causing almost all the CPU time to be spent in the OS.

4.5 Scalability

In this section, we examine Xen's ability to scale to its target of 100 domains. We discuss the memory requirements of running many instances of a guest OS and associated applications, and measure the CPU performance overhead of their execution.

We evaluated the minimum physical memory requirements of a domain booted with XenoLinux and running the default set of RH7.2 daemons, along with an `sshd` and Apache web server. The domain was given a reservation of 64MB on boot, limiting the maximum size to which it could grow. The guest OS was instructed to minimize its memory footprint by returning all pages possible to Xen. Without any swap space configured, the domain was able to reduce its memory footprint to 6.2MB; allowing the use of a swap device reduced this further to 4.2MB. A quiescent domain is able to stay in this reduced state until an incoming HTTP request or periodic service causes more memory to be required. In this event, the guest OS will request pages back from Xen, growing its footprint as required up to its configured ceiling.

This demonstrates that memory usage overhead is unlikely to be a problem for running 100 domains on a modern server class machine — far more memory will typically be committed to application data and buffer cache usage than to OS or application text pages. Xen itself maintains only a fixed 20kB of state per domain, unlike other VMMs that must maintain shadow page tables etc.

Finally, we examine the overhead of context switching between large numbers of domains rather than simply between processes. Figure 6 shows the normalized aggregate throughput obtained when running a small subset of the SPEC CINT2000 suite concurrently on between 1 and 128 domains or processes on our dual CPU server. The line representing native Linux is almost flat, indicating that for this benchmark there is no loss of aggregate performance when scheduling between so many processes; Linux identifies them all as compute bound, and schedules them with long time slices of 50ms or more. In contrast, the lower line indicates Xen's throughput when configured with its default 5ms maximum scheduling 'slice'. Although operating 128 simultaneously compute bound processes on a single server is unlikely to be commonplace in our target application area, Xen copes relatively well: running 128 domains we lose just 7.5% of total throughput relative to Linux.

Under this extreme load, we measured user-to-user UDP latency to one of the domains running the SPEC CINT2000 subset. We measured mean response times of 147ms (standard deviation 97ms). Repeating the experiment against a 129th domain that was otherwise idle, we recorded a mean response time of 5.4ms (s.d. 16ms). These figures are very encouraging — despite the substantial background load, interactive domains remain responsive.

To determine the cause of the 7.5% performance reduction, we set Xen's scheduling 'slice' to 50ms (the default value used by ESX Server). The result was a throughput curve that tracked native Linux's closely, almost eliminating the performance gap. However, as might be expected, interactive performance at high load is adversely impacted by these settings.

5. RELATED WORK

Virtualization has been applied to operating systems both commercially and in research for nearly thirty years. IBM VM/370 [19, 38] first made use of virtualization to allow binary support for legacy code. VMware [10] and Connectix [8] both virtualize commodity PC hardware, allowing multiple operating systems to run on a single host. All of these examples implement a full virtualization of (at least a subset of) the underlying hardware, rather than paravirtualizing and presenting a modified interface to the guest OS. As shown in our evaluation, the decision to present a full virtualization, although able to more easily support off-the-shelf operating systems, has detremental consequences for performance.

The virtual machine monitor approach has also been used by Disco to allow commodity operating systems to run efficiently on ccNUMA machines [7, 18]. A small number of changes had to be made to the hosted operating systems to enable virtualized execution on the MIPS architecture. In addition, certain other changes were made for performance reasons.

At present, we are aware of two other systems which take the paravirtualization approach: IBM presently supports a paravirtualized version of Linux for their zSeries mainframes, allowing large numbers of Linux instances to run simultaneously. Denali [44], discussed previously, is a contemporary isolation kernel which attempts to provide a system capable of hosting vast numbers of virtualized OS instances.

In addition to Denali, we are aware of two other efforts to use low-level virtualization to build an infrastructure for distributed systems. The vMatrix [1] project is based on VMware and aims to build a platform for moving code between different machines. As vMatrix is developed above VMware, they are more concerned with higher-level issues of distribution that those of virtualization itself. In addition, IBM provides a "Managed Hosting" service, in which virtual Linux instances may be rented on IBM mainframes.

The PlanetLab [33] project has constructed a distributed infrastructure which is intended to serve as a testbed for the research and development of geographically distributed network services. The platform is targeted at researchers and attempts to divide individual physical hosts into *slivers*, providing simultaneous low-level access to users. The current deployment uses VServers [17] and SILK [4] to manage sharing within the operating system.

We share some motivation with the operating system extensibility and active networks communities. However, when running over Xen there is no need to check for "safe" code, or for guaranteed termination — the only person hurt in either case is the client

in question. Consequently, Xen provides a more general solution: there is no need for hosted code to be digitally signed by a trusted compiler (as in SPIN [5]), to be accompanied by a safety proof (as with PCC [31]), to be written in a particular language (as in SafetyNet [22] or any Java-based system), or to rely on a particular middleware (as with mobile-agent systems). These other techniques can, of course, continue to be used within guest OSes running over Xen. This may be particularly useful for workloads with more transient tasks which would not provide an opportunity to amortize the cost of starting a new domain.

A similar argument can be made with regard to language-level virtual machine approaches: while a resource-managed JVM [9] should certainly be able to host untrusted applications, these applications must necessarily be compiled to Java bytecode and follow that particular system's security model. Meanwhile, Xen can readily support language-level virtual machines as applications running over a guest OS.

6. DISCUSSION AND CONCLUSION

We have presented the Xen hypervisor which partitions the resources of a computer between domains running guest operating systems. Our paravirtualizing design places a particular emphasis on performance and resource management. We have also described and evaluated XenoLinux, a fully-featured port of a Linux 2.4 kernel that runs over Xen.

6.1 Future Work

We believe that Xen and XenoLinux are sufficiently complete to be useful to a wider audience, and so intend to make a public release of our software in the very near future. A beta version is already under evaluation by selected parties; once this phase is complete, a general 1.0 release will be announced on our project page[3].

After the initial release we plan a number of extensions and improvements to Xen. To increase the efficiency of virtual block devices, we intend to implement a shared *universal buffer cache* indexed on block contents. This will add controlled data sharing to our design without sacrificing isolation. Adding copy-on-write semantics to virtual block devices will allow them to be safely shared among domains, while still allowing divergent file systems.

To provide better physical memory performance, we plan to implement a *last-chance page cache* (LPC) — effectively a system-wide list of free pages, of non-zero length only when machine memory is undersubscribed. The LPC is used when the guest OS virtual memory system chooses to evict a clean page; rather than discarding this completely, it may be added to the tail of the free list. A fault occurring for that page before it has been reallocated by Xen can therefore satisfied without a disk access.

An important role for Xen is as the basis of the *XenoServer* project which looks beyond individual machines and is building the control systems necessary to support an Internet-scale computing infrastructure. Key to our design is the idea that resource usage be accounted precisely and paid for by the sponsor of that job — if payments are made in real cash, we can use a congestion pricing strategy [28] to handle excess demand, and use excess revenues to pay for additional machines. This necessitates accurate and timely I/O scheduling with greater resilience to hostile workloads. We also plan to incorporate accounting into our block storage architecture by creating leases for virtual block devices.

In order to provide better support for the management and administration of XenoServers, we are incorporating more thorough support for auditing and forensic logging. We are also developing additional VFR rules which we hope will allow us to detect and prevent a wide range of antisocial network behaviour. Finally, we are continuing our work on XenoXP, focusing in the first instance on writing network and block device drivers, with the aim of fully supporting enterprise servers such as IIS.

[3] `http://www.cl.cam.ac.uk/netos/xen`

6.2 Conclusion

Xen provides an excellent platform for deploying a wide variety of network-centric services, such as local mirroring of dynamic web content, media stream transcoding and distribution, multiplayer game and virtual reality servers, and 'smart proxies' [2] to provide a less ephemeral network presence for transiently-connected devices.

Xen directly addresses the single largest barrier to the deployment of such services: the present inability to host transient servers for short periods of time and with low instantiation costs. By allowing 100 operating systems to run on a single server, we reduce the associated costs by two orders of magnitude. Furthermore, by turning the setup and configuration of each OS into a software concern, we facilitate much smaller-granularity timescales of hosting.

As our experimental results show in Section 4, the performance of XenoLinux over Xen is practically equivalent to the performance of the baseline Linux system. This fact, which comes from the careful design of the interface between the two components, means that there is no appreciable cost in having the resource management facilities available. Our ongoing work to port the BSD and Windows XP kernels to operate over Xen is confirming the generality of the interface that Xen exposes.

Acknowledgments

This work is supported by ESPRC Grant GR/S01894/01 and by Microsoft. We would like to thank Evangelos Kotsovinos, Anil Madhavapeddy, Russ Ross and James Scott for their contributions.

7. REFERENCES

[1] A. Awadallah and M. Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW 2002)*, Aug. 2002.

[2] A. Bakre and B. R. Badrinath. I-TCP: indirect TCP for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995)*, pages 136–143, June 1995.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 45–58, Feb. 1999.

[4] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg. SILK: Scout paths in the Linux kernel. Technical Report 2002-009, Uppsala University, Department of Information Technology, Feb. 2002.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles*, volume 29(5) of *ACM Operating Systems Review*, pages 267–284, Dec. 1995.

[6] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[7] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, volume 31(5) of *ACM Operating Systems Review*, pages 143–156, Oct. 1997.

[8] Connectix. Product Overview: Connectix Virtual Server, 2003. http://www.connectix.com/products/vs.html.

[9] G. Czajkowski and L. Daynés. Multitasking without compromise: a virtual machine evolution. *ACM SIGPLAN Notices*, 36(11):125–138, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001).

[10] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent*, 6397242, Oct. 1998.

[11] K. J. Duda and D. R. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles*, volume 33(5) of *ACM Operating Systems Review*, pages 261–276, Kiawah Island Resort, SC, USA, Dec. 1999.

[12] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 211–224, Boston, MA, USA, Dec. 2002.

[13] D. Engler, S. K. Gupta, and F. Kaashoek. AVM: Application-level virtual memory. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, May 1995.

[14] Ensim. Ensim Virtual Private Servers, 2003. http://www.ensim.com/products/materials/datasheet_vps_051003.pdf.

[15] K. A. Fraser, S. M. Hand, T. L. Harris, I. M. Leslie, and I. A. Pratt. The Xenoserver computing infrastructure. Technical Report UCAM-CL-TR-552, University of Cambridge, Computer Laboratory, Jan. 2003.

[16] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS Support and Applications for Trusted Computing. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems, Kauai, Hawaii*, May 2003.

[17] J. Gelinas. Virtual Private Servers and Security Contexts, 2003. http://www.solucorp.qc.ca/miscprj/s_context.hc.

[18] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles*, volume 33(5) of *ACM Operating Systems Review*, pages 154–169, Dec. 1999.

[19] P. H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, Nov. 1983.

[20] S. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 73–86, Oct. 1999.

[21] S. Hand, T. L. Harris, E. Kotsovinos, and I. Pratt. Controlling the XenoServer Open Platform, April 2003.

[22] A. Jeffrey and I. Wakeman. A Survey of Semantic Techniques for Active Networks, Nov. 1997. http://www.cogs.susx.ac.uk/projects/safetynet/.

[23] M. F. Kaashoek, D. R. Engler, G. R. Granger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, volume 31(5) of *ACM Operating Systems Review*, pages 52–65, Oct. 1997.

[24] R. Kessler and M. Hill. Page placement algorithms for large real-indexed caches. *ACM Transaction on Computer Systems*, 10(4):338–359, Nov. 1992.

[25] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 Annual USENIX Technical Conference*, Jun 2003.

[26] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications, Calicoon, NY*, Jun 2002.

[27] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, Sept. 1996.

[28] J. MacKie-Mason and H. Varian. Pricing congestible network resources. *IEEE Journal on Selected Areas In Communications*, 13(7):1141–1149, Sept. 1995.

[29] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–294, Berkeley, Jan. 1996. Usenix Association.

[30] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 89–104, Boston, MA, USA, Dec. 2002.

[31] G. C. Necula. Proof-carrying code. In *Conference Record of POPL 1997: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Jan. 1997.

[32] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the IEEE Real Time Technology and Applications Symposium*, pages 111–120, June 1999.

[33] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, NJ, USA, Oct. 2002.

[34] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*, pages 67–76, Los Alamitos, CA, USA, Apr. 22–26 2001. IEEE Computer Society.

[35] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: accounted execution of untrusted code. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, 1999.

[36] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium, Denver, CO, USA*, pages 129–144, Aug. 2000.

[37] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 377–390, Boston, MA, USA, Dec. 2002.

[38] L. Seawright and R. MacKinnon. VM/370 – a study of multiplicity and usefulness. *IBM Systems Journal*, pages 4–17, 1979.

[39] P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *Proceedings of ACM SIGMETRICS'98, the International Conference on Measurement and Modeling of Computer Systems*, pages 44–55, June 1998.

[40] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H.M.Vin. Application Performance in the QLinux Multimedia Operating System. In *Proceedings of the 8th ACM Conference on Multimedia*, Nov. 2000.

[41] D. Tennenhouse. Layered Multiplexing Considered Harmful. In Rudin and Williamson, editors, *Protocols for High-Speed Networks*, pages 143–148. North Holland, 1989.

[42] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 181–194, Boston, MA, USA, Dec. 2002.

[43] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, 2002.

[44] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 195–210, Boston, MA, USA, Dec. 2002.

Are Virtual Machine Monitors Microkernels Done Right?

Steven Hand, Andrew Warfield, Keir Fraser,
Evangelos Kotsovinos, Dan Magenheimer†
University of Cambridge Computer Laboratory
† HP Labs, Fort Collins, USA

1 Introduction

At the last HotOS, Mendel Rosenblum gave an 'outrageous' opinion that the academic obsession with microkernels during the past two decades produced many publications but little impact. He argued that virtual machine monitors (VMMs) had had considerably more practical uptake, despite—or perhaps due to—being principally developed by industry.

In this paper, we investigate this claim in light of our experiences in developing the Xen [1] virtual machine monitor. We argue that modern VMMs present a practical platform which allows the development and deployment of innovative systems research: in essence, VMMs are microkernels done right.

We first compare and contrast the architectural purity of microkernels with the pragmatic design of VMMs. In Section 3, we discuss several technical characteristics of microkernels that have proven, in our experience, to be incompatible with effective VMM design.

Rob Pike has irreverently suggested that "systems software research is irrelevant", implying that academic systems research has negligible impact outside the university. In Section 4, we claim that VMMs provide a platform on which innovative systems research ideas can be developed and deployed. We believe that providing a common framework for hosting novel systems will increase the penetration and relevance of systems research.

2 Motivation and μHistory

Microkernels and virtual machine monitors are both well explored areas of operating systems research dating back more than twenty years. Both areas have focused on a refactoring of systems into isolated components that communicate across well-defined, typically narrow interfaces. Despite considerable structural similarities, the two research areas are remarkable in their differences: Microkernels received considerable attention from academic researchers through the eighties and nineties, while VMM research has largely been the bailiwick of industrial research.

2.1 Microkernels: Noble Idealism

The most prolific academic microkernel ever developed was probably Mach [2]. A major research project at CMU, Mach's beginnings were in the Rochester Intelligent Gateway (RIG) [3] followed by the Accent kernel [4]. The key motivation to all of these systems was that the OS be "communication oriented"; that they have rigid, message-based interfaces between system components. Many of the abstractions used in Mach and later systems appeared initially in the RIG, including that of the *port*. However, the communications orientation of these systems originally intended to allow the distribution of system components across a set of dissimilar physical hosts.

The term "microkernel" was coined in response to the predominant monolithic kernels at the time. Microkernel advocates claimed that a smaller OS core would be easier to maintain, validate, and port to new architectures. A common theme throughout much of the microkernel work is that microkernels were *architecturally better* than monolithic kernels; from a research perspective they certainly are, as it is considerably easier to work on a single system component if that component is not entangled with other code.

Mach is hardly unique as an example of innovative microkernel projects. In the heyday of microkernels, many interesting systems were constructed including Chorus [5], Amoeba [6], and L3/L4 [7, 8]. Several of these evolved to show that microkernels, which were often criticized for poor performance, could match and even outperform commercial unix variants.

2.2 VMMs: Rough Pragmatism

Early work on Virtual Machines (VMs) [9, 10] was motivated by the need to improve hardware utilisation by facilitating the secure time-sharing of machines. Typically, VMs in IBM's model are identical "copies" of the underlying hardware where each instance runs its own operating system. Multiple VMs can be created and managed via interfaces exported by the VMM, a component running on the physical hardware.

As virtual machines may be owned by multiple, competing users, strong resource isolation mechanisms are required in the VMM. Another important facility provided by VMMs is that of *sharing* the hardware: securely multiplexing several virtual machines over a single set of physical resources.

The use of a VMM presents an additional layer of indirection between the hardware and the user, and it is necessary that this does not result in a noticeable performance degradation. For that reason, a significant amount of research effort in VMMs has been directed towards maintaining a low *performance overhead*, with considerable success [1].

Although VMM architectures differ in the degree of modification required to the guest operating systems they host, these modifications typically range from very small to none at all. Xen and Denali [11] host slightly modified "guest OSes" for improved system performance while VMware[1] provides full hardware virtualization so that no guestOS changes are needed.

An important characteristic of most VMMs is their ability to support the execution of *out-of-the-box applications*; users can run code that is executable on their regular desktop machines.

Because of the above properties of allowing users to securely share hardware on machines at a low performance cost, improving machine utilization, and not requiring modifications to the applications, VMMs have always presented a very appealing platform for practical deployment.

Previous research has combined microkernel and VM concepts to provide recursive VMs running on a microkernel-based OS [12]. User-mode Linux [13] achieves software-level virtualization by running a VMM as an application inside a host Linux system. Additionally, several research systems do not fall cleanly into either the VMM or microkernel camps; for example both the Exokernel [14] and Nemesis [15] systems provide low-level interfaces and resource protection above a small trusted kernel, but without the fine-grained modularization of microkernels or the OS-granularity multiplexing of VMMs.

[1] http://www.vmware.com

3 Architectural Lessons

While both microkernels and VMMs share rich histories of innovation, it is increasingly obvious that VMMs have achieved predominance in modern systems. In Section 4 we will revisit how many of the goals of microkernels remain relevant today. We first discuss some technical characteristics that consumed the research efforts of the microkernel community, but which have proven in our experience to be inconsequential in the development of modern VMMs.

We note that VMMs and microkernels bear a great deal of architectural similarity. The Denali team has re-titled their VMM μDenali in reference to its explicit restructuring as a microkernel, while there has recently been an effort to develop VMM functionality on top of the L4 microkernel. In this section, however, we focus on what we perceive to be the important differences between the two approaches.

3.1 Avoid Liability Inversion

One of the fundamental properties of microkernels is the division of a system into isolated user-space components. While the resulting kernel is smaller, this functional reduction relaxes the dependability boundaries within the system: applications must depend on other user-level components in order to run. More importantly, the microkernel itself depends on application level components, such as pagers, to make forward progress.

External pagers are an excellent example of this phenomenon: the failure conditions associated with them are one of the earliest and most recurrent problems discussed in microkernel-related literature [16]. Relegating a critical system-wide component to user-space, the kernel can be left waiting on the pager to evict a page before it can proceed. Various inelegant timeout and fallback mechanisms were required to avoid deadlock. By depending on arbitrary user-level components in order to continue execution, the kernel abdicates its liability for system liveness. We refer to this as *liability inversion*.

One of the principal design guidelines in Xen has been to avoid exactly these situations. Xen's memory management system, for instance, has no notion of paging whatsoever; rather it strictly partitions memory between VMs and allows limited facilities for sharing. VMs are themselves responsible for any paging within these allocations. The point here is perhaps a subtle one: decisions such as this are engineered to ensure that VM failure is

isolated and cannot degrade the stability of the system as a whole.

Consider, as a counterexample to external paging, the storage virtual machine used in Parallax [17]. In this case a storage VM is used to serve block storage to a collection of client VMs. A crash in the storage server could compromise the function of its clients, but not of the system as a whole: in particular, Xen itself does not depend on the correctness of the storage VM to function. Moreover, the dependency between the storage VM and its clients is *explicit*: the isolation between dependent VMs can be increased by separating the storage VM into multiple instances. This is essentially just the traditional trade-off between isolation and sharing which is observed in the design of any system.

3.2 Make IPC Performance Irrelevant

IPC performance is arguably the most revered hallmark of microkernel research. As message-based communication between system components is crucial to the operation of any microkernel, the literature is saturated with papers measuring IPC performance, improving IPC performance, and even questioning the relevance of IPC performance. However in our experience fast IPC is not a critical design concern in the construction of high-performance VMMs.

There are a number of reasons why we can avoid relying on fast, typically synchronous, IPC mechanisms. First, since VMMs hold isolation to be a key goal, IPC between virtual machines is considerably less common in general. This is a natural consequence of the fact that VMM design considers entire operating systems to be the unit of scheduling and protection: hence synchronization and protected control transfer are only necessary when two virtual machines wish to explicitly communicate.

Secondly, we have determined that a clear separation between *control* and *data* path operations allows us to optimize for the common case. In particular, we observe that by explicitly setting up communication channels, we can perform potentially expensive permission and safety checks at initialization time and then elide validation during more frequent data path operations. This decoupling furthermore allows higher-level communcation mechanisms great freedom in how they are implemented.

A particular example of this is seen in the implementation of *control-* and *device-channels* within Xen. Both of these are built upon a simple asynchronous unidirectional event mechanism which is the only communications primitive provided by Xen. However by combining pairs of events with shared memory, we can build both synchronous IPC for control operations and asynchronous producer-consumer rings for bulk, batched, data transfer. Even these latter allow considerable flexibility in use: by determining how often notifications are generated or waited upon, one can explicitly trade-off throughput and latency.

The difference between approaches to communication between isolated components is a very interesting example of the idealism versus pragmatism dichotomy described in the previous section. Microkernel designers view systems as sets of components that interact over IPC-, and potentially RPC-, based interfaces: they consider these interactions as procedure calls, in which the entire system is a collection of well-isolated components. VMM designers do not assume anywhere near the same degree of coherency within their systems: where VMs do communicate, they may not only be written in separate programming languages, but may also be running completely different operating systems. A consequence of this is that communications within VMMs typically looks like interactions with devices: a simple asynchronous control path combined with fixed-format transparent bulk data transfer.

3.3 Treat the OS as a Component

The final important difference between VMMs and microkernels is that of the *granularity of componentization*. By positioning themselves as a response to monolithic kernels, microkernels focused on dividing the functional units of an OS into discrete parts. A practical problem faced by microkernel developers is that which faces any new OS effort: by changing the API visible to applications, an OS forfeits the complete set of software available to existing systems. As such, most microkernel projects were left spending considerable effort to implement emulation interface layers for existing OSes.

VMMs differ significantly here in that their *a priori* intention is to support existing operating systems. For example, out-of-the-box code, compiled to be executable on a range of existing OSes, can be run on a guest operating system on top of Xen. This reduces the cost of entry for users and applications, makes virtualization attractive and practical for a wider community, and addresses two of the main problems of microkernel systems — the difficulty in attracting a substantial user base, and the challenge in keeping microkernel operating systems up to date with the feature sets of existing OSes.

By supporting existing OSes, VMMs need only justify the potential performance overheads they incur in order to be an attractive option. As shown in [1] and independently verified in [18], the overhead imposed by Xen is very small.

Secondly, VMMs appeal to developers because they present a *familiar development environment*. Using existing OSes as fundamental blocks of componentization allows developers to continue using the same tool set that they have on their existing system, freeing them to concentrate on more important issues.

The Parallax storage system [17], mentioned earlier, is an example of the sort of componentization that VMMs allow: The storage VM is a set of daemons running on Linux in an isolated virtual machine. The system can be used by any OS that runs on Xen because it provides the same block interface that Xen's existing block virtualization uses. Parallax provides an extension to an OS function, an ability touted by microkernels, but does it in a familiar development environment, using existing OS drivers, and providing support in turn for a range of client OSes. Moreover, the implementation is independent from both Xen and client OS code: provided that the block interface remains common, the OS extension itself does not depend on the source of the client OSes or the VMM.

Similar benefits accrue for the developers of the VMM itself: for example, Xen makes extensive use of existing tools for network routing, disk management, and configuration as part of the control software running in the privileged management VM.

The size of components — i.e., guest OSes — running on a VMM can be adjusted, depending on the functionality required from them. One example is *ttylinux*, a minimalistic Linux distribution, providing multi-tasking, multi-user, and networking capabilities within less than 4 megabytes of operating system size. It is also easy to build a simple single-threaded 'library OS' which enables the use of extremely lightweight components when desired for security or performance reasons.

4 The future for VMMs

Having illustrated what we feel are the key differences between microkernel and VMM design, we now consider how VMMs may be used to realize many of the research benefits achieved by the microkernel community. These include narrow interfaces between system components providing easy **extensibility** of device and OS functionality, a small code base that can guarantee **security** more easily than monolithic kernels, and strong isolation providing opportunities for improved **manageability**.

Narrow interfaces between system components are crucial in facilitating extensibility. The clean IPC interfaces provided by microkernels allowed researchers the ability to focus on specific system components without becoming entangled in unrelated code. Similarly, the narrow interfaces present in Xen allow devices and OSes to be easily extended. Xen's device architecture has allowed device drivers to be isolated in a separate VM for dependability [19], and permitted low-level interfaces to be extended without necessitating modification of the target OS or VMM [20]. Indeed, it seems very likely that the exploration of how services and management will be structured in a multi-OS VMM system will continue to present many exciting research opportunities.

A further advantage of narrow interfaces, coupled with a minimal privileged kernel, is the tractability of achieving a high degree of confidence in the security of a system. This has been explored in the microkernel community by projects such as Flask [21] and EROS [22]. Several groups have expressed interest in developing these ideas for Xen, using concepts from projects such as the Flask-derived SELinux.

A final avenue of innovation realized recently by VMMs has been to explore less performance-centric aspects of systems development. As with the examples above, VMMs are a promising platform because these so-called 'ilities' can be developed and applied to existing systems. For example, live OS migration [23] allows a running OS to be relocated to a new physical host, empowering administrators to better manage physical resources. The ability to 'rewind' a VM's state has been used for intrusion detection [24], debugging [25] and administration [26].

5 Conclusion

Despite having dissimilar motivations and origins, microkernels and VMMs share many architectural commonalities. In this paper we have attempted to illustrate some of the technical separations between the two classes of system that, in our opinion, have favoured the success of VMMs in recent years. More importantly though, we posit that—despite the decline in microkernel research— modern VMMs, Xen in particular, are in fact a specific point in the microkernel design space; that VMMs are microkernels done right. In light of this opinion, we observe that many of the advantages realized through the structure of microkernel systems may be similarly developed above a VMM. Moreover, because VMMs run commodity operating systems and applications we claim that they present a valuable platform for innovative systems research to have impact outside the academic laboratory.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. Summer USENIX Conference*, June 1986.

[3] E. Ball, J. Feldman, J. Low, R. Rashid, and P. Rovner. RIG, Rochester's Intelligent Gateway: System overview. In *Proc. 2nd International Conference on Software Engineering*, page 132, 1976.

[4] R. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. In *Proc. 8th ACM Symposium on Operating Systems Principles (SOSP)*, pages 64–75, 1981.

[5] V. Abrossimov, M. Rozier, and M. Gien. Virtual memory management in chorus. In *Proc. European Workshop on Process in Distributed Operating Systems and Distributed Systems Management*, pages 45–59, 1990.

[6] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.

[7] J. Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating Systems Principles (SOSP)*, December 1993.

[8] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of μ-Kernel-Based Systems. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[9] R. Adair, R. Bayles, L. Comeau, and R. Creasy. A virtual machine system for the 360/40. Technical Report 320-2007, IBM Corporation, Cambridge Scientific Center, May 1966.

[10] R. Goldberg. Architectural principles for virtual computer systems. PhD thesis, Harvard University, 1972.

[11] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th Symposium on Operating System Design and Implementation (OSDI)*, December 2002.

[12] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–151, October 1996.

[13] J. Dike. User-mode Linux. In *Proc. 5th Annual Linux Showcase and Conference*, November 2001.

[14] D. Engler, F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.

[15] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. 14(7):1280–1297, September 1996.

[16] M. Young, A. Tevanian, R. F. Rashid, D. B. Golub, J. L. Eppinger, J. Chew, W. J. Bolosky, D. L. Black, and R. V. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc. 11th ACM Symposium on Operating Systems Principles (SOSP)*, 1987.

[17] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing storage for a million machines. In *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X*.

[18] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. Matthews. Xen and the art of repeated research. In *Proc. USENIX Annual Technical Conference*, June 2004.

[19] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. ACM OASIS Workshop*, 2004.

[20] A. Warfield, K. Fraser, S. Hand, and T. Deegan. Facilitating the development of soft devices. In *Proc. USENIX Annual Technical Conference*, April 2005.

[21] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proc. Eighth USENIX Security Symposium*, August 1999.

[22] J. Shapiro, J. Smith, and D.Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, 1999.

[23] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[24] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[25] S. King, G. Dunlap, and P. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX Annual Technical Conference*, 2005.

[26] A. Whitaker, R. Cox, and S. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 77–90, December 2004.

Live Migration of Virtual Machines

Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen†,
Eric Jul†, Christian Limpach, Ian Pratt, Andrew Warfield
University of Cambridge Computer Laboratory *15 JJ Thomson Avenue, Cambridge, UK*
firstname.lastname@cl.cam.ac.uk
† *Department of Computer Science* *University of Copenhagen, Denmark*
{jacobg,eric}@diku.dk

Abstract

Migrating operating system instances across distinct physical hosts is a useful tool for administrators of data centers and clusters: It allows a clean separation between hardware and software, and facilitates fault management, load balancing, and low-level system maintenance.

By carrying out the majority of migration while OSes continue to run, we achieve impressive performance with minimal service downtimes; we demonstrate the migration of entire OS instances on a commodity cluster, recording service downtimes as low as $60ms$. We show that that our performance is sufficient to make live migration a practical tool even for servers running interactive loads.

In this paper we consider the design options for migrating OSes running services with liveness constraints, focusing on data center and cluster environments. We introduce and analyze the concept of *writable working set*, and present the design, implementation and evaluation of high-performance OS migration built on top of the Xen VMM.

1 Introduction

Operating system virtualization has attracted considerable interest in recent years, particularly from the data center and cluster computing communities. It has previously been shown [1] that paravirtualization allows many OS instances to run concurrently on a single physical machine with high performance, providing better use of physical resources and isolating individual OS instances.

In this paper we explore a further benefit allowed by virtualization: that of live OS migration. Migrating an entire OS and all of its applications as one unit allows us to avoid many of the difficulties faced by process-level migration approaches. In particular the narrow interface between a virtualized OS and the virtual machine monitor (VMM) makes it easy avoid the problem of 'residual dependencies' [2] in which the original host machine must remain available and network-accessible in order to service certain system calls or even memory accesses on behalf of migrated processes. With virtual machine migration, on the other hand, the original host may be decommissioned once migration has completed. This is particularly valuable when migration is occurring in order to allow maintenance of the original host.

Secondly, migrating at the level of an entire virtual machine means that in-memory state can be transferred in a consistent and (as will be shown) efficient fashion. This applies to kernel-internal state (e.g. the TCP control block for a currently active connection) as well as application-level state, even when this is shared between multiple cooperating processes. In practical terms, for example, this means that we can migrate an on-line game server or streaming media server without requiring clients to reconnect: something not possible with approaches which use application-level restart and layer 7 redirection.

Thirdly, live migration of virtual machines allows a separation of concerns between the users and operator of a data center or cluster. Users have 'carte blanche' regarding the software and services they run within their virtual machine, and need not provide the operator with any OS-level access at all (e.g. a root login to quiesce processes or I/O prior to migration). Similarly the operator need not be concerned with the details of what is occurring within the virtual machine; instead they can simply migrate the entire operating system and its attendant processes as a single unit.

Overall, live OS migration is a extremelely powerful tool for cluster administrators, allowing separation of hardware and software considerations, and consolidating clustered hardware into a single coherent management domain. If a physical machine needs to be removed from service an administrator may migrate OS instances including the applications that they are running to alternative machine(s), freeing the original machine for maintenance. Similarly, OS instances may be rearranged across machines in a cluster to relieve load on congested hosts. In these situations the combination of virtualization and migration significantly improves manageability.

We have implemented high-performance migration support for Xen [1], a freely available open source VMM for commodity hardware. Our design and implementation addresses the issues and tradeoffs involved in live local-area migration. Firstly, as we are targeting the migration of active OSes hosting live services, it is critically important to minimize the *downtime* during which services are entirely unavailable. Secondly, we must consider the *total migration time*, during which state on both machines is synchronized and which hence may affect reliability. Furthermore we must ensure that migration does not unnecessarily disrupt active services through resource contention (e.g., CPU, network bandwidth) with the migrating OS.

Our implementation addresses all of these concerns, allowing for example an OS running the SPECweb benchmark to migrate across two physical hosts with only $210ms$ unavailability, or an OS running a Quake 3 server to migrate with just $60ms$ downtime. Unlike application-level restart, we can maintain network connections and application state during this process, hence providing effectively seamless migration from a user's point of view.

We achieve this by using a *pre-copy* approach in which pages of memory are iteratively copied from the source machine to the destination host, all without ever stopping the execution of the virtual machine being migrated. Page-level protection hardware is used to ensure a consistent snapshot is transferred, and a rate-adaptive algorithm is used to control the impact of migration traffic on running services. The final phase pauses the virtual machine, copies any remaining pages to the destination, and resumes execution there. We eschew a 'pull' approach which faults in missing pages across the network since this adds a residual dependency of arbitrarily long duration, as well as providing in general rather poor performance.

Our current implementation does not address migration across the wide area, nor does it include support for migrating local block devices, since neither of these are required for our target problem space. However we discuss ways in which such support can be provided in Section 7.

2 Related Work

The Collective project [3] has previously explored VM migration as a tool to provide mobility to users who work on different physical hosts at different times, citing as an example the transfer of an OS instance to a home computer while a user drives home from work. Their work aims to optimize for slow (e.g., ADSL) links and longer time spans, and so stops OS execution for the duration of the transfer, with a set of enhancements to reduce the transmitted image size. In contrast, our efforts are concerned with the migration of live, in-service OS instances on fast neworks with only tens of milliseconds of downtime. Other projects that have explored migration over longer time spans by stopping and then transferring include Internet Suspend/Resume [4] and μDenali [5].

Zap [6] uses partial OS virtualization to allow the migration of process domains (pods), essentially process groups, using a modified Linux kernel. Their approach is to isolate all process-to-kernel interfaces, such as file handles and sockets, into a contained namespace that can be migrated. Their approach is considerably faster than results in the Collective work, largely due to the smaller units of migration. However, migration in their system is still on the order of seconds at best, and does not allow live migration; pods are entirely suspended, copied, and then resumed. Furthermore, they do not address the problem of maintaining open connections for existing services.

The live migration system presented here has considerable shared heritage with the previous work on NomadBIOS [7], a virtualization and migration system built on top of the L4 microkernel [8]. NomadBIOS uses pre-copy migration to achieve very short best-case migration downtimes, but makes no attempt at adapting to the writable working set behavior of the migrating OS.

VMware has recently added OS migration support, dubbed *VMotion*, to their VirtualCenter management software. As this is commercial software and strictly disallows the publication of third-party benchmarks, we are only able to infer its behavior through VMware's own publications. These limitations make a thorough technical comparison impossible. However, based on the VirtualCenter User's Manual [9], we believe their approach is generally similar to ours and would expect it to perform to a similar standard.

Process migration, a hot topic in systems research during the 1980s [10, 11, 12, 13, 14], has seen very little use for real-world applications. Milojicic *et al* [2] give a thorough survey of possible reasons for this, including the problem of the *residual dependencies* that a migrated process retains on the machine from which it migrated. Examples of residual dependencies include open file descriptors, shared memory segments, and other local resources. These are undesirable because the original machine must remain available, and because they usually negatively impact the performance of migrated processes.

For example Sprite [15] processes executing on foreign nodes require some system calls to be forwarded to the home node for execution, leading to at best reduced performance and at worst widespread failure if the home node is unavailable. Although various efforts were made to ameliorate performance issues, the underlying reliance on the availability of the home node could not be avoided. A similar fragility occurs with MOSIX [14] where a deputy process on the home node must remain available to support remote execution.

We believe the residual dependency problem cannot easily be solved in any process migration scheme – even modern mobile run-times such as Java and .NET suffer from problems when network partition or machine crash causes class loaders to fail. The migration of entire operating systems inherently involves fewer or zero such dependencies, making it more resilient and robust.

3 Design

At a high level we can consider a virtual machine to encapsulate access to a set of physical resources. Providing live migration of these VMs in a clustered server environment leads us to focus on the physical resources used in such environments: specifically on memory, network and disk.

This section summarizes the design decisions that we have made in our approach to live VM migration. We start by describing how memory and then device access is moved across a set of physical hosts and then go on to a high-level description of how a migration progresses.

3.1 Migrating Memory

Moving the contents of a VM's memory from one physical host to another can be approached in any number of ways. However, when a VM is running a live service it is important that this transfer occurs in a manner that balances the requirements of minimizing both *downtime* and *total migration time*. The former is the period during which the service is unavailable due to there being no currently executing instance of the VM; this period will be directly visible to clients of the VM as service interruption. The latter is the duration between when migration is initiated and when the original VM may be finally discarded and, hence, the source host may potentially be taken down for maintenance, upgrade or repair.

It is easiest to consider the trade-offs between these requirements by generalizing memory transfer into three phases:

Push phase The source VM continues running while certain pages are pushed across the network to the new destination. To ensure consistency, pages modified during this process must be re-sent.

Stop-and-copy phase The source VM is stopped, pages are copied across to the destination VM, then the new VM is started.

Pull phase The new VM executes and, if it accesses a page that has not yet been copied, this page is faulted in ("pulled") across the network from the source VM.

Although one can imagine a scheme incorporating all three phases, most practical solutions select one or two of the three. For example, *pure stop-and-copy* [3, 4, 5] involves halting the original VM, copying all pages to the destination, and then starting the new VM. This has advantages in terms of simplicity but means that both downtime and total migration time are proportional to the amount of physical memory allocated to the VM. This can lead to an unacceptable outage if the VM is running a live service.

Another option is *pure demand-migration* [16] in which a short stop-and-copy phase transfers essential kernel data structures to the destination. The destination VM is then started, and other pages are transferred across the network on first use. This results in a much shorter downtime, but produces a much longer total migration time; and in practice, performance after migration is likely to be unacceptably degraded until a considerable set of pages have been faulted across. Until this time the VM will fault on a high proportion of its memory accesses, each of which initiates a synchronous transfer across the network.

The approach taken in this paper, *pre-copy* [11] migration, balances these concerns by combining a bounded iterative push phase with a typically very short stop-and-copy phase. By 'iterative' we mean that pre-copying occurs in *rounds*, in which the pages to be transferred during round n are those that are modified during round $n-1$ (all pages are transferred in the first round). Every VM will have some (hopefully small) set of pages that it updates very frequently and which are therefore poor candidates for pre-copy migration. Hence we bound the number of rounds of pre-copying, based on our analysis of the *writable working set* (WWS) behavior of typical server workloads, which we present in Section 4.

Finally, a crucial additional concern for live migration is the impact on active services. For instance, iteratively scanning and sending a VM's memory image between two hosts in a cluster could easily consume the entire bandwidth available between them and hence starve the active services of resources. This *service degradation* will occur to some extent during any live migration scheme. We address this issue by carefully controlling the network and CPU resources used by the migration process, thereby ensuring that it does not interfere excessively with active traffic or processing.

3.2 Local Resources

A key challenge in managing the migration of OS instances is what to do about resources that are associated with the physical machine that they are migrating away from. While memory can be copied directly to the new host, connections to local devices such as disks and network interfaces demand additional consideration. The two key problems that we have encountered in this space concern what to do with network resources and local storage.

For network resources, we want a migrated OS to maintain all open network connections without relying on forwarding mechanisms on the original host (which may be shut down following migration), or on support from mobility or redirection mechanisms that are not already present (as in [6]). A migrating VM will include all protocol state (e.g. TCP PCBs), and will carry its IP address with it.

To address these requirements we observed that in a cluster environment, the network interfaces of the source and destination machines typically exist on a single switched LAN. Our solution for managing migration with respect to network in this environment is to generate an unsolicited ARP reply from the migrated host, advertising that the IP has moved to a new location. This will reconfigure peers to send packets to the new physical address, and while a very small number of in-flight packets may be lost, the migrated domain will be able to continue using open connections with almost no observable interference.

Some routers are configured not to accept broadcast ARP replies (in order to prevent IP spoofing), so an unsolicited ARP may not work in all scenarios. If the operating system is aware of the migration, it can opt to send directed replies only to interfaces listed in its own ARP cache, to remove the need for a broadcast. Alternatively, on a switched network, the migrating OS can keep its original Ethernet MAC address, relying on the network switch to detect its move to a new port[1].

In the cluster, the migration of storage may be similarly addressed: Most modern data centers consolidate their storage requirements using a network-attached storage (NAS) device, in preference to using local disks in individual servers. NAS has many advantages in this environment, including simple centralised administration, widespread vendor support, and reliance on fewer spindles leading to a reduced failure rate. A further advantage for migration is that it obviates the need to migrate disk storage, as the NAS is uniformly accessible from all host machines in the cluster. We do not address the problem of migrating local-disk storage in this paper, although we suggest some possible strategies as part of our discussion of future work.

3.3 Design Overview

The logical steps that we execute when migrating an OS are summarized in Figure 1. We take a conservative approach to the management of migration with regard to safety and failure handling. Although the consequences of hardware failures can be severe, our basic principle is that safe migration should at no time leave a virtual OS more exposed to system failure than when it is running on the original single host. To achieve this, we view the migration process as a transactional interaction between the two hosts involved:

[1]Note that on most Ethernet controllers, hardware MAC filtering will have to be disabled if multiple addresses are in use (though some cards support filtering of multiple addresses in hardware) and so this technique is only practical for switched networks.

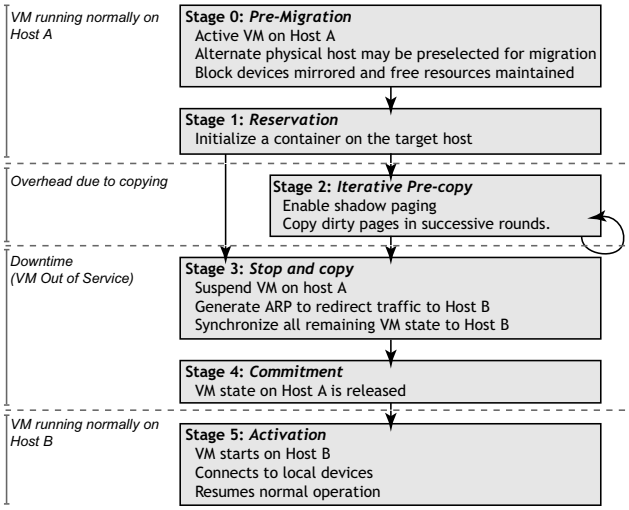


Figure 1: Migration timeline

Stage 0: Pre-Migration We begin with an active VM on physical host A. To speed any future migration, a target host may be preselected where the resources required to receive migration will be guaranteed.

Stage 1: Reservation A request is issued to migrate an OS from host A to host B. We initially confirm that the necessary resources are available on B and reserve a VM container of that size. Failure to secure resources here means that the VM simply continues to run on A unaffected.

Stage 2: Iterative Pre-Copy During the first iteration, all pages are transferred from A to B. Subsequent iterations copy only those pages dirtied during the previous transfer phase.

Stage 3: Stop-and-Copy We suspend the running OS instance at A and redirect its network traffic to B. As described earlier, CPU state and any remaining inconsistent memory pages are then transferred. At the end of this stage there is a consistent suspended copy of the VM at both A and B. The copy at A is still considered to be primary and is resumed in case of failure.

Stage 4: Commitment Host B indicates to A that it has successfully received a consistent OS image. Host A acknowledges this message as commitment of the migration transaction: host A may now discard the original VM, and host B becomes the primary host.

Stage 5: Activation The migrated VM on B is now activated. Post-migration code runs to reattach device drivers to the new machine and advertise moved IP addresses.

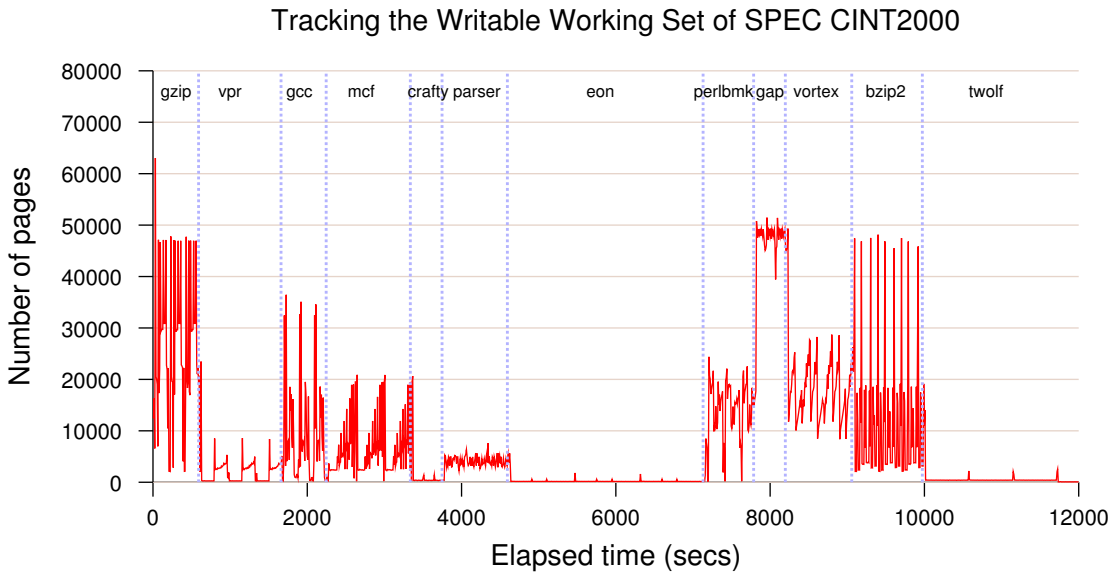


Figure 2: WWS curve for a complete run of SPEC CINT2000 (512MB VM)

This approach to failure management ensures that at least one host has a consistent VM image at all times during migration. It depends on the assumption that the original host remains stable until the migration commits, and that the VM may be suspended and resumed on that host with no risk of failure. Based on these assumptions, a migration request essentially attempts to move the VM to a new host, and on any sort of failure execution is resumed locally, aborting the migration.

4 Writable Working Sets

When migrating a live operating system, the most significant influence on service performance is the overhead of coherently transferring the virtual machine's memory image. As mentioned previously, a simple stop-and-copy approach will achieve this in time proportional to the amount of memory allocated to the VM. Unfortunately, during this time any running services are completely unavailable.

A more attractive alternative is pre-copy migration, in which the memory image is transferred while the operating system (and hence all hosted services) continue to run. The drawback however, is the wasted overhead of transferring memory pages that are subsequently modified, and hence must be transferred again. For many workloads there will be a small set of memory pages that are updated very frequently, and which it is not worth attempting to maintain coherently on the destination machine before stopping and copying the remainder of the VM.

The fundamental question for iterative pre-copy migration is: how does one determine when it is time to stop the pre-copy phase because too much time and resource is being wasted? Clearly if the VM being migrated never modifies memory, a single pre-copy of each memory page will suffice to transfer a consistent image to the destination. However, should the VM continuously dirty pages faster than the rate of copying, then all pre-copy work will be in vain and one should immediately stop and copy.

In practice, one would expect most workloads to lie somewhere between these extremes: a certain (possibly large) set of pages will seldom or never be modified and hence are good candidates for pre-copy, while the remainder will be written often and so should best be transferred via stop-and-copy – we dub this latter set of pages the *writable working set* (WWS) of the operating system by obvious extension of the original working set concept [17].

In this section we analyze the WWS of operating systems running a range of different workloads in an attempt to obtain some insight to allow us build heuristics for an efficient and controllable pre-copy implementation.

4.1 Measuring Writable Working Sets

To trace the writable working set behaviour of a number of representative workloads we used Xen's shadow page tables (see Section 5) to track dirtying statistics on all pages used by a particular executing operating system. This allows us to determine within any time period the set of pages written to by the virtual machine.

Using the above, we conducted a set of experiments to sam-

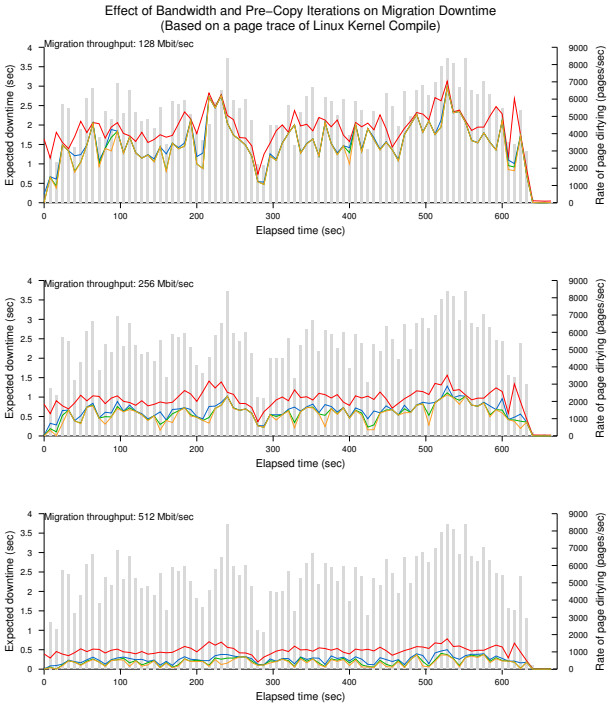


Figure 3: Expected downtime due to last-round memory copy on traced page dirtying of a Linux kernel compile.

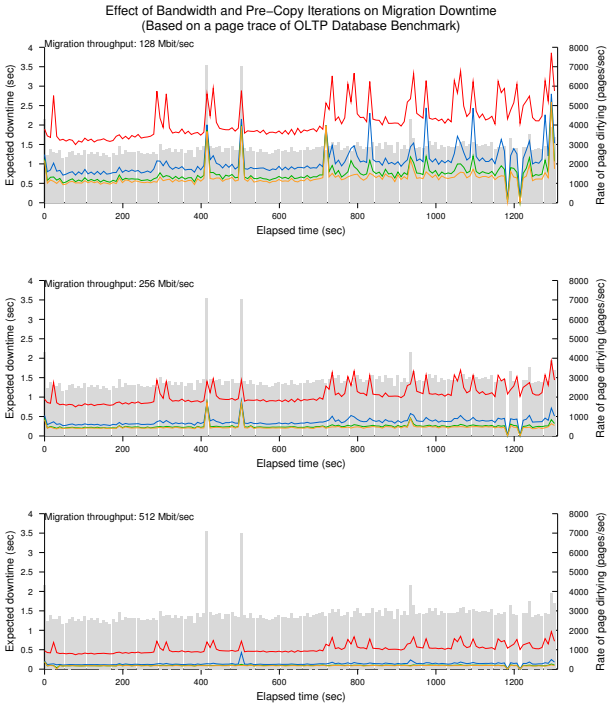


Figure 4: Expected downtime due to last-round memory copy on traced page dirtying of OLTP.

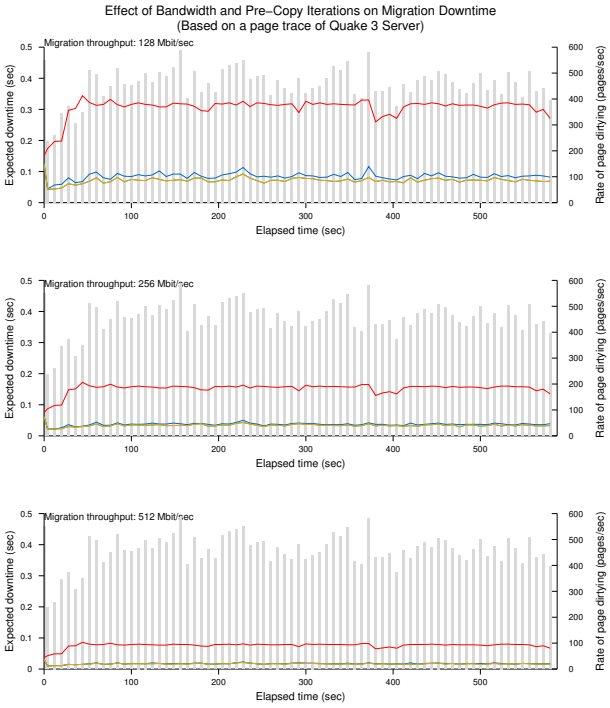


Figure 5: Expected downtime due to last-round memory copy on traced page dirtying of a Quake 3 server.

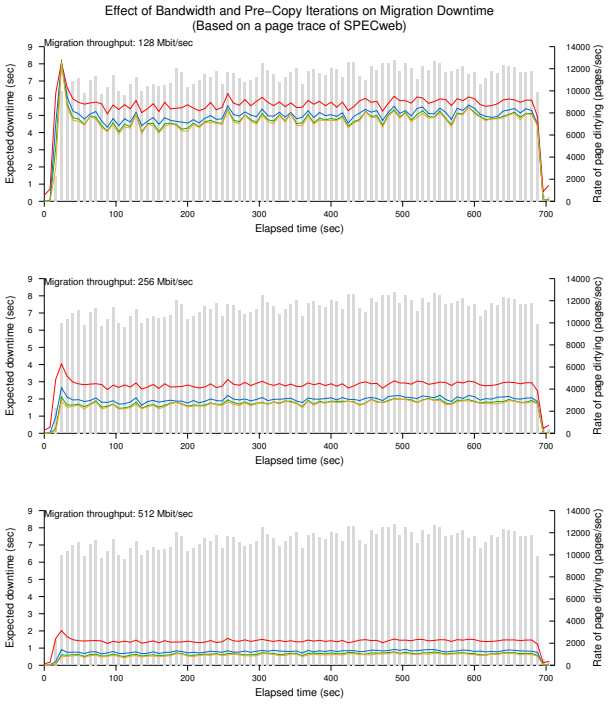


Figure 6: Expected downtime due to last-round memory copy on traced page dirtying of SPECweb.

ple the writable working set size for a variety of benchmarks. Xen was running on a dual processor Intel Xeon 2.4GHz machine, and the virtual machine being measured had a memory allocation of 512MB. In each case we started the relevant benchmark in one virtual machine and read the dirty bitmap every $50ms$ from another virtual machine, cleaning it every 8 seconds – in essence this allows us to compute the WWS with a (relatively long) 8 second window, but estimate it at a finer (50ms) granularity.

The benchmarks we ran were SPEC CINT2000, a Linux kernel compile, the OSDB OLTP benchmark using PostgreSQL and SPECweb99 using Apache. We also measured a Quake 3 server as we are particularly interested in highly interactive workloads.

Figure 2 illustrates the writable working set curve produced for the SPEC CINT2000 benchmark run. This benchmark involves running a series of smaller programs in order and measuring the overall execution time. The x-axis measures elapsed time, and the y-axis shows the number of 4KB pages of memory dirtied within the corresponding 8 second interval; the graph is annotated with the names of the sub-benchmark programs.

From this data we observe that the writable working set varies significantly between the different sub-benchmarks. For programs such as '`eon`' the WWS is a small fraction of the total working set and hence is an excellent candidate for migration. In contrast, '`gap`' has a consistently high dirtying rate and would be problematic to migrate. The other benchmarks go through various phases but are generally amenable to live migration. Thus performing a migration of an operating system will give different results depending on the workload and the precise moment at which migration begins.

4.2 Estimating Migration Effectiveness

We observed that we could use the trace data acquired to estimate the effectiveness of iterative pre-copy migration for various workloads. In particular we can simulate a particular network bandwidth for page transfer, determine how many pages would be dirtied during a particular iteration, and then repeat for successive iterations. Since we know the approximate WWS behaviour at every point in time, we can estimate the overall amount of data transferred in the final stop-and-copy round and hence estimate the downtime.

Figures 3–6 show our results for the four remaining workloads. Each figure comprises three graphs, each of which corresponds to a particular network bandwidth limit for page transfer; each individual graph shows the WWS histogram (in light gray) overlaid with four line plots estimating service downtime for up to four pre-copying rounds.

Looking at the topmost line (one pre-copy iteration), the first thing to observe is that pre-copy migration always performs considerably better than naive stop-and-copy. For a 512MB virtual machine this latter approach would require 32, 16, and 8 seconds downtime for the 128Mbit/sec, 256Mbit/sec and 512Mbit/sec bandwidths respectively. Even in the worst case (the starting phase of SPECweb), a single pre-copy iteration reduces downtime by a factor of four. In most cases we can expect to do considerably better – for example both the Linux kernel compile and the OLTP benchmark typically experience a reduction in downtime of at least a factor of sixteen.

The remaining three lines show, in order, the effect of performing a total of two, three or four pre-copy iterations prior to the final stop-and-copy round. In most cases we see an increased reduction in downtime from performing these additional iterations, although with somewhat diminishing returns, particularly in the higher bandwidth cases.

This is because all the observed workloads exhibit a small but extremely frequently updated set of 'hot' pages. In practice these pages will include the stack and local variables being accessed within the currently executing processes as well as pages being used for network and disk traffic. The hottest pages will be dirtied at least as fast as we can transfer them, and hence must be transferred in the final stop-and-copy phase. This puts a lower bound on the best possible service downtime for a particular benchmark, network bandwidth and migration start time.

This interesting tradeoff suggests that it may be worthwhile increasing the amount of bandwidth used for page transfer in later (and shorter) pre-copy iterations. We will describe our rate-adaptive algorithm based on this observation in Section 5, and demonstrate its effectiveness in Section 6.

5 Implementation Issues

We designed and implemented our pre-copying migration engine to integrate with the Xen virtual machine monitor [1]. Xen securely divides the resources of the host machine amongst a set of resource-isolated virtual machines each running a dedicated OS instance. In addition, there is one special *management virtual machine* used for the administration and control of the machine.

We considered two different methods for initiating and managing state transfer. These illustrate two extreme points in the design space: *managed migration* is performed largely outside the migratee, by a migration daemon running in the management VM; in contrast, *self migration* is implemented almost entirely within the migratee OS with only a small stub required on the destination machine.

In the following sections we describe some of the implementation details of these two approaches. We describe how we use dynamic network rate-limiting to effectively

balance network contention against OS downtime. We then proceed to describe how we ameliorate the effects of rapid page dirtying, and describe some performance enhancements that become possible when the OS is aware of its migration — either through the use of self migration, or by adding explicit paravirtualization interfaces to the VMM.

5.1 Managed Migration

Managed migration is performed by migration daemons running in the management VMs of the source and destination hosts. These are responsible for creating a new VM on the destination machine, and coordinating transfer of live system state over the network.

When transferring the memory image of the still-running OS, the control software performs *rounds* of copying in which it performs a complete scan of the VM's memory pages. Although in the first round all pages are transferred to the destination machine, in subsequent rounds this copying is restricted to pages that were dirtied during the previous round, as indicated by a *dirty bitmap* that is copied from Xen at the start of each round.

During normal operation the page tables managed by each guest OS are the ones that are walked by the processor's MMU to fill the TLB. This is possible because guest OSes are exposed to real physical addresses and so the page tables they create do not need to be mapped to physical addresses by Xen.

To log pages that are dirtied, Xen inserts *shadow page tables* underneath the running OS. The shadow tables are populated on demand by translating sections of the guest page tables. Translation is very simple for dirty logging: all page-table entries (PTEs) are initially read-only mappings in the shadow tables, regardless of what is permitted by the guest tables. If the guest tries to modify a page of memory, the resulting page fault is trapped by Xen. If write access is permitted by the relevant guest PTE then this permission is extended to the shadow PTE. At the same time, we set the appropriate bit in the VM's dirty bitmap.

When the bitmap is copied to the control software at the start of each pre-copying round, Xen's bitmap is cleared and the shadow page tables are destroyed and recreated as the migratee OS continues to run. This causes all write permissions to be lost: all pages that are subsequently updated are then added to the now-clear dirty bitmap.

When it is determined that the pre-copy phase is no longer beneficial, using heuristics derived from the analysis in Section 4, the OS is sent a control message requesting that it suspend itself in a state suitable for migration. This causes the OS to prepare for resumption on the destination machine; Xen informs the control software once the OS has done this. The dirty bitmap is scanned one last time for remaining inconsistent memory pages, and these are transferred to the destination together with the VM's checkpointed CPU-register state.

Once this final information is received at the destination, the VM state on the source machine can safely be discarded. Control software on the destination machine scans the memory map and rewrites the guest's page tables to reflect the addresses of the memory pages that it has been allocated. Execution is then resumed by starting the new VM at the point that the old VM checkpointed itself. The OS then restarts its virtual device drivers and updates its notion of wallclock time.

Since the transfer of pages is OS agnostic, we can easily support any guest operating system – all that is required is a small paravirtualized stub to handle resumption. Our implementation currently supports Linux 2.4, Linux 2.6 and NetBSD 2.0.

5.2 Self Migration

In contrast to the managed method described above, self migration [18] places the majority of the implementation within the OS being migrated. In this design no modifications are required either to Xen or to the management software running on the source machine, although a migration stub must run on the destination machine to listen for incoming migration requests, create an appropriate empty VM, and receive the migrated system state.

The pre-copying scheme that we implemented for self migration is conceptually very similar to that for managed migration. At the start of each pre-copying round every page mapping in every virtual address space is write-protected. The OS maintains a dirty bitmap tracking dirtied physical pages, setting the appropriate bits as write faults occur. To discriminate migration faults from other possible causes (for example, copy-on-write faults, or access-permission faults) we reserve a spare bit in each PTE to indicate that it is write-protected only for dirty-logging purposes.

The major implementation difficulty of this scheme is to transfer a consistent OS checkpoint. In contrast with a managed migration, where we simply suspend the migratee to obtain a consistent checkpoint, self migration is far harder because the OS must continue to run in order to transfer its final state. We solve this difficulty by logically checkpointing the OS on entry to a final *two-stage* stop-and-copy phase. The first stage disables all OS activity except for migration and then peforms a final scan of the dirty bitmap, clearing the appropriate bit as each page is transferred. Any pages that are dirtied during the final scan, and that are still marked as dirty in the bitmap, are copied to a shadow buffer. The second and final stage then transfers the contents of the shadow buffer — page updates are ignored during this transfer.

5.3 Dynamic Rate-Limiting

It is not always appropriate to select a single network bandwidth limit for migration traffic. Although a low limit avoids impacting the performance of running services, analysis in Section 4 showed that we must eventually pay in the form of an extended downtime because the hottest pages in the writable working set are not amenable to pre-copy migration. The downtime can be reduced by increasing the bandwidth limit, albeit at the cost of additional network contention.

Our solution to this impasse is to dynamically adapt the bandwidth limit during each pre-copying round. The administrator selects a minimum and a maximum bandwidth limit. The first pre-copy round transfers pages at the minimum bandwidth. Each subsequent round counts the number of pages dirtied in the previous round, and divides this by the duration of the previous round to calculate the *dirtying rate*. The bandwidth limit for the next round is then determined by adding a constant increment to the previous round's dirtying rate — we have empirically determined that 50Mbit/sec is a suitable value. We terminate pre-copying when the calculated rate is greater than the administrator's chosen maximum, or when less than 256KB remains to be transferred. During the final stop-and-copy phase we minimize service downtime by transferring memory at the maximum allowable rate.

As we will show in Section 6, using this adaptive scheme results in the bandwidth usage remaining low during the transfer of the majority of the pages, increasing only at the end of the migration to transfer the hottest pages in the WWS. This effectively balances short downtime with low average network contention and CPU usage.

5.4 Rapid Page Dirtying

Our working-set analysis in Section 4 shows that every OS workload has some set of pages that are updated extremely frequently, and which are therefore not good candidates for pre-copy migration even when using all available network bandwidth. We observed that rapidly-modified pages are very likely to be dirtied again by the time we attempt to transfer them in any particular pre-copying round. We therefore periodically 'peek' at the current round's dirty bitmap and transfer only those pages dirtied in the previous round that have not been dirtied again at the time we scan them.

We further observed that page dirtying is often physically *clustered* — if a page is dirtied then it is disproportionally likely that a close neighbour will be dirtied soon after. This increases the likelihood that, if our peeking does not detect one page in a cluster, it will detect none. To avoid this unfortunate behaviour we scan the VM's physical memory space in a pseudo-random order.

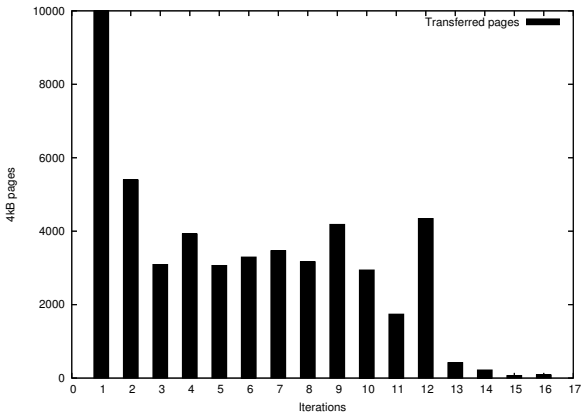


Figure 7: Rogue-process detection during migration of a Linux kernel build. After the twelfth iteration a maximum limit of forty write faults is imposed on every process, drastically reducing the total writable working set.

5.5 Paravirtualized Optimizations

One key benefit of paravirtualization is that operating systems can be made aware of certain important differences between the real and virtual environments. In terms of migration, this allows a number of optimizations by informing the operating system that it is about to be migrated – at this stage a migration stub handler within the OS could help improve performance in at least the following ways:

Stunning Rogue Processes. Pre-copy migration works best when memory pages can be copied to the destination host faster than they are dirtied by the migrating virtual machine. This may not always be the case – for example, a test program which writes one word in every page was able to dirty memory at a rate of 320 Gbit/sec, well ahead of the transfer rate of any Ethernet interface. This is a synthetic example, but there may well be cases in practice in which pre-copy migration is unable to keep up, or where migration is prolonged unnecessarily by one or more 'rogue' applications.

In both the managed and self migration cases, we can mitigate against this risk by forking a monitoring thread within the OS kernel when migration begins. As it runs within the OS, this thread can monitor the WWS of *individual processes* and take action if required. We have implemented a simple version of this which simply limits each process to 40 write faults before being moved to a wait queue – in essence we 'stun' processes that make migration difficult. This technique works well, as shown in Figure 7, although

one must be careful not to stun important interactive services.

Freeing Page Cache Pages. A typical operating system will have a number of 'free' pages at any time, ranging from truly free (page allocator) to cold buffer cache pages. When informed a migration is to begin, the OS can simply return some or all of these pages to Xen in the same way it would when using the ballooning mechanism described in [1]. This means that the time taken for the first "full pass" iteration of pre-copy migration can be reduced, sometimes drastically. However should the contents of these pages be needed again, they will need to be faulted back in from disk, incurring greater overall cost.

6 Evaluation

In this section we present a thorough evaluation of our implementation on a wide variety of workloads. We begin by describing our test setup, and then go on explore the migration of several workloads in detail. Note that none of the experiments in this section use the paravirtualized optimizations discussed above since we wished to measure the baseline performance of our system.

6.1 Test Setup

We perform test migrations between an identical pair of Dell PE-2650 server-class machines, each with dual Xeon 2GHz CPUs and 2GB memory. The machines have Broadcom TG3 network interfaces and are connected via switched Gigabit Ethernet. In these experiments only a single CPU was used, with HyperThreading enabled. Storage is accessed via the iSCSI protocol from an NetApp F840 network attached storage server except where noted otherwise. We used XenLinux 2.4.27 as the operating system in all cases.

6.2 Simple Web Server

We begin our evaluation by examining the migration of an Apache 1.3 web server serving static content at a high rate. Figure 8 illustrates the throughput achieved when continuously serving a single 512KB file to a set of one hundred concurrent clients. The web server virtual machine has a memory allocation of 800MB.

At the start of the trace, the server achieves a consistent throughput of approximately 870Mbit/sec. Migration starts twenty seven seconds into the trace but is initially rate-limited to 100Mbit/sec (12% CPU), resulting in the server throughput dropping to 765Mbit/s. This initial low-rate pass transfers 776MB and lasts for 62 seconds, at which point the migration algorithm described in Section 5 increases its rate over several iterations and finally suspends the VM after a further 9.8 seconds. The final stop-and-copy phase then transfer the remaining pages and the web server resumes at full rate after a 165ms outage.

This simple example demonstrates that a highly loaded server can be migrated with both controlled impact on live services and a short downtime. However, the working set of the server in this case is rather small, and so this should be expected to be a relatively easy case for live migration.

6.3 Complex Web Workload: SPECweb99

A more challenging Apache workload is presented by SPECweb99, a complex application-level benchmark for evaluating web servers and the systems that host them. The workload is a complex mix of page requests: 30% require dynamic content generation, 16% are HTTP POST operations, and 0.5% execute a CGI script. As the server runs, it generates access and POST logs, contributing to disk (and therefore network) throughput.

A number of client machines are used to generate the load for the server under test, with each machine simulating a collection of users concurrently accessing the web site. SPECweb99 defines a minimum quality of service that each user must receive for it to count as 'conformant'; an aggregate bandwidth in excess of 320Kbit/sec over a series of requests. The SPECweb score received is the number of conformant users that the server successfully maintains. The considerably more demanding workload of SPECweb represents a challenging candidate for migration.

We benchmarked a single VM running SPECweb and recorded a maximum score of 385 conformant clients — we used the RedHat `gnbd` network block device in place of iSCSI as the lighter-weight protocol achieves higher performance. Since at this point the server is effectively in overload, we then relaxed the offered load to 90% of maximum (350 conformant connections) to represent a more realistic scenario.

Using a virtual machine configured with 800MB of memory, we migrated a SPECweb99 run in the middle of its execution. Figure 9 shows a detailed analysis of this migration. The x-axis shows time elapsed since start of migration, while the y-axis shows the network bandwidth being used to transfer pages to the destination. Darker boxes illustrate the page transfer process while lighter boxes show the pages dirtied during each iteration. Our algorithm adjusts the transfer rate relative to the page dirty rate observed during the previous round (denoted by the height of the lighter boxes).

As in the case of the static web server, migration begins

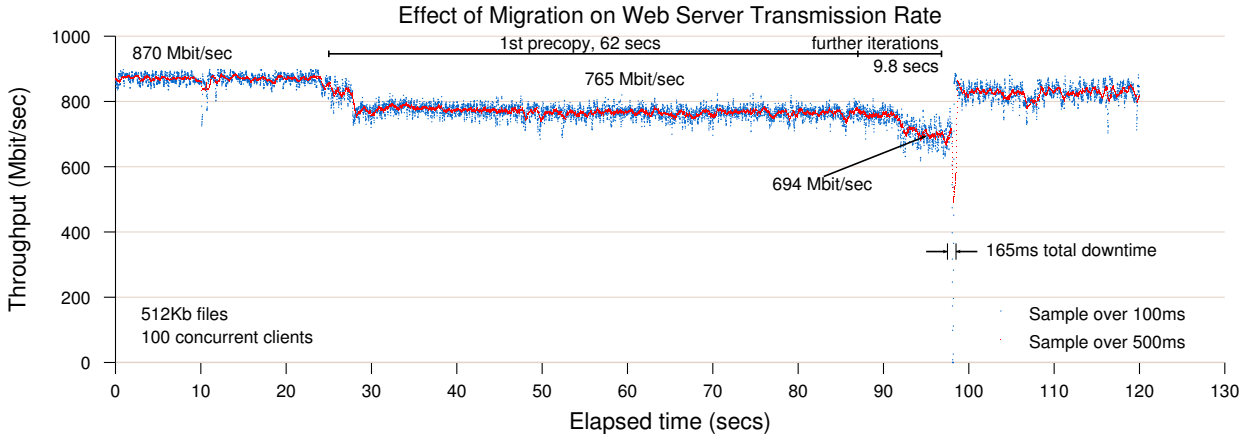


Figure 8: Results of migrating a running web server VM.

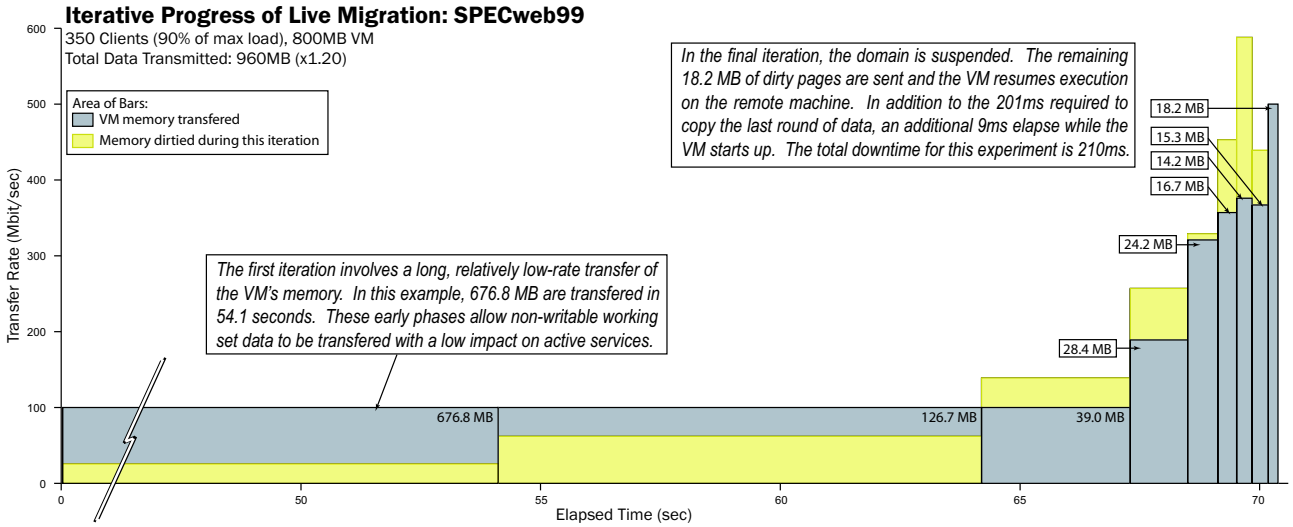


Figure 9: Results of migrating a running SPECweb VM.

with a long period of low-rate transmission as a first pass is made through the memory of the virtual machine. This first round takes 54.1 seconds and transmits 676.8MB of memory. Two more low-rate rounds follow, transmitting 126.7MB and 39.0MB respectively before the transmission rate is increased.

The remainder of the graph illustrates how the adaptive algorithm tracks the page dirty rate over successively shorter iterations before finally suspending the VM. When suspension takes place, 18.2MB of memory remains to be sent. This transmission takes $201ms$, after which an additional $9ms$ is required for the domain to resume normal execution.

The total downtime of $210ms$ experienced by the SPECweb clients is sufficiently brief to maintain the 350 conformant clients. This result is an excellent validation of our approach: a heavily (90% of maximum) loaded server is migrated to a separate physical host with a total migration time of seventy-one seconds. Furthermore the migration does not interfere with the quality of service demanded by SPECweb's workload. This illustrates the applicability of migration as a tool for administrators of demanding live services.

6.4 Low-Latency Server: Quake 3

Another representative application for hosting environments is a multiplayer on-line game server. To determine the effectiveness of our approach in this case we configured a virtual machine with 64MB of memory running a

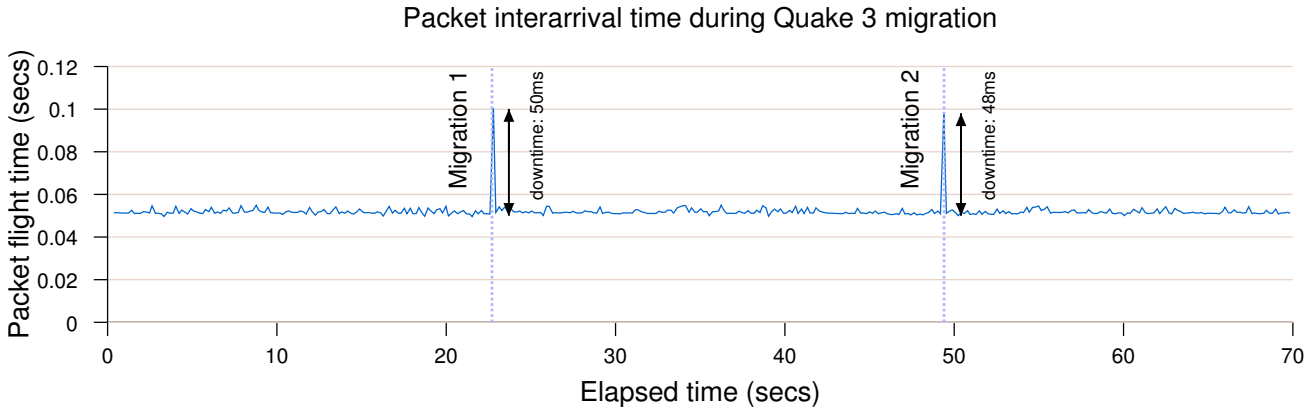


Figure 10: Effect on packet response time of migrating a running Quake 3 server VM.

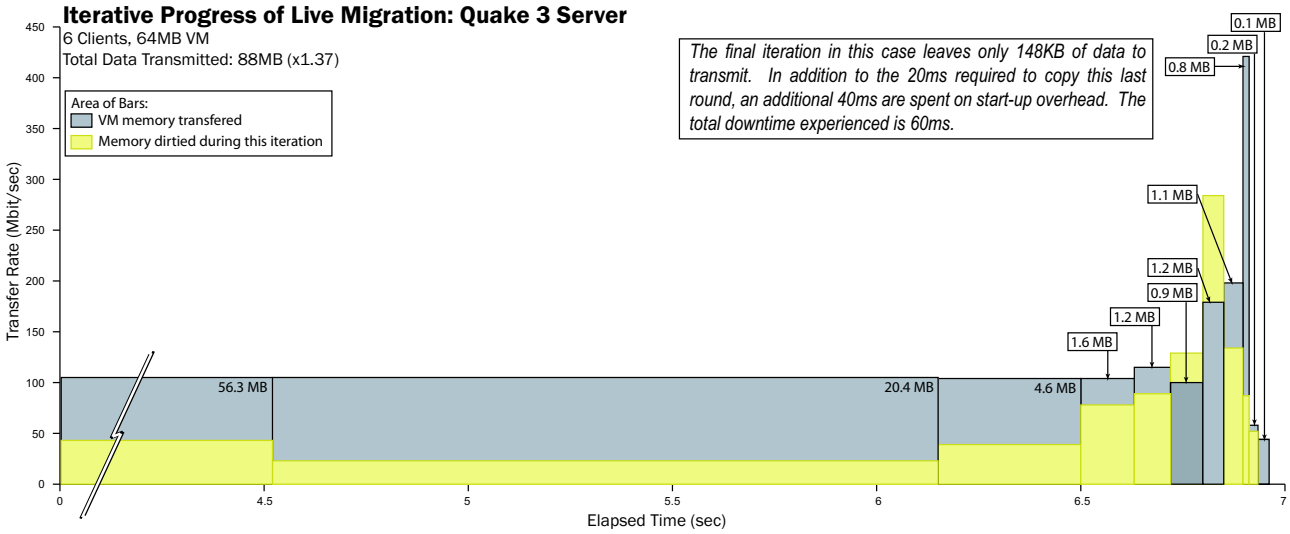


Figure 11: Results of migrating a running Quake 3 server VM.

Quake 3 server. Six players joined the game and started to play within a shared arena, at which point we initiated a migration to another machine. A detailed analysis of this migration is shown in Figure 11.

The trace illustrates a generally similar progression as for SPECweb, although in this case the amount of data to be transferred is significantly smaller. Once again the transfer rate increases as the trace progresses, although the final stop-and-copy phase transfers so little data (148KB) that the full bandwidth is not utilized.

Overall, we are able to perform the live migration with a total downtime of $60ms$. To determine the effect of migration on the live players, we performed an additional experiment in which we migrated the running Quake 3 server twice and measured the inter-arrival time of packets received by clients. The results are shown in Figure 10. As can be seen, from the client point of view migration manifests itself as a transient increase in response time of $50ms$. In neither case was this perceptible to the players.

6.5 A Diabolical Workload: MMuncher

As a final point in our evaluation, we consider the situation in which a virtual machine is writing to memory faster than can be transferred across the network. We test this diabolical case by running a 512MB host with a simple C program that writes constantly to a 256MB region of memory. The results of this migration are shown in Figure 12.

In the first iteration of this workload, we see that half of the memory has been transmitted, while the other half is immediately marked dirty by our test program. Our algorithm attempts to adapt to this by scaling itself relative to the perceived initial rate of dirtying; this scaling proves in-

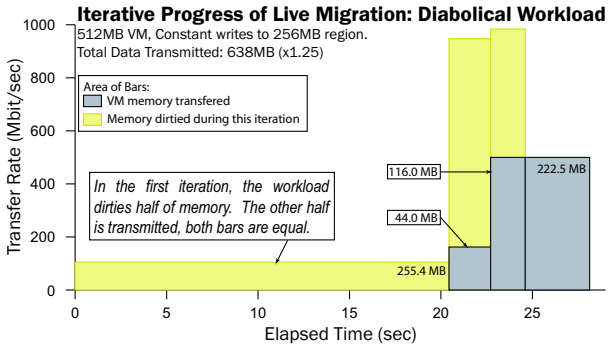


Figure 12: Results of migrating a VM running a diabolical workload.

sufficient, as the rate at which the memory is being written becomes apparent. In the third round, the transfer rate is scaled up to 500Mbit/s in a final attempt to outpace the memory writer. As this last attempt is still unsuccessful, the virtual machine is suspended, and the remaining dirty pages are copied, resulting in a downtime of 3.5 seconds. Fortunately such dirtying rates appear to be rare in real workloads.

7 Future Work

Although our solution is well-suited for the environment we have targeted – a well-connected data-center or cluster with network-accessed storage – there are a number of areas in which we hope to carry out future work. This would allow us to extend live migration to wide-area networks, and to environments that cannot rely solely on network-attached storage.

7.1 Cluster Management

In a cluster environment where a pool of virtual machines are hosted on a smaller set of physical servers, there are great opportunities for dynamic load balancing of processor, memory and networking resources. A key challenge is to develop cluster control software which can make informed decision as to the placement and movement of virtual machines.

A special case of this is 'evacuating' VMs from a node that is to be taken down for scheduled maintenance. A sensible approach to achieving this is to migrate the VMs in increasing order of their observed WWS. Since each VM migrated frees resources on the node, additional CPU and network becomes available for those VMs which need it most. We are in the process of building a cluster controller for Xen systems.

7.2 Wide Area Network Redirection

Our layer 2 redirection scheme works efficiently and with remarkably low outage on modern gigabit networks. However, when migrating outside the local subnet this mechanism will not suffice. Instead, either the OS will have to obtain a new IP address which is within the destination subnet, or some kind of indirection layer, on top of IP, must exist. Since this problem is already familiar to laptop users, a number of different solutions have been suggested. One of the more prominent approaches is that of Mobile IP [19] where a node on the home network (the *home agent*) forwards packets destined for the client (*mobile node*) to a *care-of address* on the foreign network. As with all residual dependencies this can lead to both performance problems and additional failure modes.

Snoeren and Balakrishnan [20] suggest addressing the problem of connection migration at the TCP level, augmenting TCP with a secure token negotiated at connection time, to which a relocated host can refer in a special SYN packet requesting reconnection from a new IP address. Dynamic DNS updates are suggested as a means of locating hosts after a move.

7.3 Migrating Block Devices

Although NAS prevails in the modern data center, some environments may still make extensive use of local disks. These present a significant problem for migration as they are usually considerably larger than volatile memory. If the entire contents of a disk must be transferred to a new host before migration can complete, then total migration times may be intolerably extended.

This latency can be avoided at migration time by arranging to *mirror* the disk contents at one or more remote hosts. For example, we are investigating using the built-in software RAID and iSCSI functionality of Linux to implement disk mirroring before and during OS migration. We imagine a similar use of software RAID-5, in cases where data on disks requires a higher level of availability. Multiple hosts can act as storage targets for one another, increasing availability at the cost of some network traffic.

The effective management of local storage for clusters of virtual machines is an interesting problem that we hope to further explore in future work. As virtual machines will typically work from a small set of common system images (for instance a generic Fedora Linux installation) and make individual changes above this, there seems to be opportunity to manage copy-on-write system images across a cluster in a way that facilitates migration, allows replication, and makes efficient use of local disks.

8 Conclusion

By integrating live OS migration into the Xen virtual machine monitor we enable rapid movement of interactive workloads within clusters and data centers. Our dynamic network-bandwidth adaptation allows migration to proceed with minimal impact on running services, while reducing total downtime to below discernable thresholds.

Our comprehensive evaluation shows that realistic server workloads such as SPECweb99 can be migrated with just $210ms$ downtime, while a Quake3 game server is migrated with an imperceptible $60ms$ outage.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP19)*, pages 164–177. ACM Press, 2003.

[2] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[3] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M.Rosenblum. Optimizing the migration of virtual computers. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, December 2002.

[4] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, 2002.

[5] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[6] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI-02)*, pages 361–376, December 2002.

[7] Jacob G. Hansen and Asger K. Henriksen. Nomadic operating systems. Master's thesis, Dept. of Computer Science, University of Copenhagen, Denmark, 2002.

[8] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of micro-kernel-based systems. In *Proceedings of the sixteenth ACM Symposium on Operating System Principles*, pages 66–77. ACM Press, 1997.

[9] VMWare, Inc. *VMWare VirtualCenter Version 1.2 User's Manual*. 2004.

[10] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the ninth ACM Symposium on Operating System Principles*, pages 110–119. ACM Press, 1983.

[11] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the tenth ACM Symposium on Operating System Principles*, pages 2–12. ACM Press, 1985.

[12] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988.

[13] Fred Douglis and John K. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[14] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[15] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ; ACM CR 8905-0314*, 21(2), 1988.

[16] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 13–24. ACM Press, 1987.

[17] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[18] Jacob G. Hansen and Eric Jul. Self-migration of operating systems. In *Proceedings of the 11th ACM SIGOPS European Workshop (EW 2004)*, pages 126–130, 2004.

[19] C. E. Perkins and A. Myles. Mobile IP. *Proceedings of International Telecommunications Symposium*, pages 415–419, 1997.

[20] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 155–166. ACM Press, 2000.

Cluster-Based Scalable Network Services

Armando Fox Steven D. Gribble Yatin Chawathe Eric A. Brewer Paul Gauthier
University of California at Berkeley
{fox, gribble, yatin, brewer}@cs.berkeley.edu
Inktomi Corporation
gauthier@inktomi.com

We identify three fundamental requirements for scalable network services: incremental scalability and overflow growth provisioning, 24x7 availability through fault masking, and cost-effectiveness. We argue that clusters of commodity workstations interconnected by a high-speed SAN are exceptionally well-suited to meeting these challenges for Internet-server workloads, provided the software infrastructure for managing partial failures and administering a large cluster does not have to be reinvented for each new service. To this end, we propose a general, layered architecture for building cluster-based scalable network services that encapsulates the above requirements for reuse, and a service-programming model based on composable workers that perform transformation, aggregation, caching, and customization (TACC) of Internet content. For both performance and implementation simplicity, the architecture and TACC programming model exploit BASE, a weaker-than-ACID data semantics that results from trading consistency for availability and relying on soft state for robustness in failure management. Our architecture can be used as an "off the shelf" infrastructural platform for creating new network services, allowing authors to focus on the "content" of the service (by composing TACC building blocks) rather than its implementation. We discuss two real implementations of services based on this architecture: TranSend, a Web distillation proxy deployed to the UC Berkeley dialup IP population, and HotBot, the commercial implementation of the Inktomi search engine. We present detailed measurements of TranSend's performance based on substantial client traces, as well as anecdotal evidence from the TranSend and HotBot experience, to support the claims made for the architecture.

1 Introduction

> *"One of the overall design goals is to create a computing system which is capable of meeting almost all of the requirements of a large computer utility. Such systems must run continuously and reliably 7 days a week, 24 hours a day... and must be capable of meeting wide service demands.*
>
> *"Because the system must ultimately be comprehensive and able to adapt to unknown future requirements, its framework must be general, and capable of evolving over time."*
>
> — Corbató and Vyssotsky on *Multics*, 1965 [17]

Although it is normally viewed as an operating system, Multics (Multiplexed Information and Computer Service) was originally conceived as an infrastructural computing service, so it is not surprising that its goals as stated above are similar to our own. The primary obstacle to deploying Multics was the absence of the network infrastructure, which is now in place. Network applications have exploded in popularity in part because they are easier to manage and evolve than their desktop application counterparts: they eliminate the need for software distribution, and simplify customer service and bug tracking by avoiding the difficulty of dealing with multiple platforms and versions. Also, basic queueing theory shows that a large central (virtual) server is more efficient in both cost and utilization than a collection of smaller servers; standalone desktop systems represent the degenerate case of one "server" per user. All of these support the argument for Network Computers [28].

However, network services remain difficult to deploy because of three fundamental challenges: scalability, availability and cost effectiveness.

- By *scalability,* we mean that when the load offered to the service increases, an incremental and linear increase in hardware can maintain the same per-user level of service.
- By *availability,* we mean that the service as a whole must be available 24x7, despite transient partial hardware or software failures.
- By *cost effectiveness*, we mean that the service must be economical to administer and expand, even though it potentially comprises many workstation nodes.

We observe that clusters of workstations have some fundamental properties that can be exploited to meet these requirements: using commodity PCs as the unit of scaling allows the service to ride the leading edge of the cost/performance curve, the inherent redundancy of clusters can be used to mask transient failures, and "embarrassingly parallel" network service workloads map well onto networks of workstations. However, developing cluster software and administering a running cluster remain complex. The primary contributions of this work are the design, analysis, and implementation of a layered framework for building network services that addresses this complexity. New services can use this framework as an off-the-shelf solution to scalability, availability, and several other problems, and focus instead on the *content* of the service being developed. The lower layer handles scalability, availability, load balancing, support for bursty offered load, and system monitoring and visualization, while the middle layer provides extensible support for caching, transformation among MIME types, aggregation of information from multiple sources, and personalization of the service for each of a large number of users (*mass customization*). The top layer allows composition of transformation and aggregation into a specific service, such as accelerated Web browsing or a search engine.

Pervasive throughout our design and implementation strategies is the observation that much of the data manipulated by a network service can tolerate semantics weaker than ACID [26]. We combine ideas from prior work on tradeoffs between availability and consistency, and the use of soft state for robust fault-tolerance to characterize the data semantics of many network services, which we refer to as BASE semantics (basically available, soft state, eventual consistency). In addition to demonstrating how BASE simplifies the implementation of our architecture, we present a programming model for service authoring that is a good fit for BASE semantics and that maps well onto our cluster-based service framework.

1.1 Validation: Two Real Services

Our framework reflects the implementation of two real network services in use today: TranSend, a scalable transformation and caching proxy for the 25,000 Berkeley dialup IP users (connecting through a bank of 600 modems), and the Inktomi search engine (commercialized as HotBot), which performs millions of queries per day against a database of over 50 million web pages.

The Inktomi search engine is an aggregation server that was initially developed to explore the use of cluster technology to han-

Appears in Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP-16), Saint-Malo, France, October 1997.

dle the scalability and availability requirements of network services. The commercial version, HotBot, handles several million queries per day against a full-text database of 54 million web pages. It has been incrementally scaled up to 60 nodes, provides high availability, and is extremely cost effective. Inktomi predates the framework we describe, and thus differs from it in some respects. However, it strongly influenced the framework's design, and we will use it to validate particular design decisions.

We focus our detailed discussion on TranSend, which provides Web caching and data transformation. In particular, real-time, datatype-specific distillation and refinement [22] of inline Web images results in an end-to-end latency reduction by a factor of 3-5, giving the user a much more responsive Web surfing experience with only modest image quality degradation. TranSend was developed at UC Berkeley and has been deployed for the 25,000 dialup IP users there, and is being deployed to a similar community at UC Davis.

In the remainder of this section we argue that clusters are an excellent fit for Internet services, provided the challenges we describe for cluster software development can be surmounted. In Section 2 we describe the proposed layered architecture for building new services, and a programming model for creating services that maps well onto the architecture. We show how TranSend and HotBot map onto this architecture, using HotBot to justify specific design decisions within the architecture. Sections 3 and 4 describe the TranSend implementation and its measured performance, including experiments on its scalability and fault tolerance properties. Section 5 discusses related work and the continuing evolution of this work, and we summarize our observations and contributions in Section 6.

1.2 Advantages of Clusters

Particularly in the area of Internet service deployment, clusters provide three primary benefits over single larger machines, such as SMPs: incremental scalability, high availability, and the cost/performance and maintenance benefits of commodity PC's. We elaborate on each of these in turn.

Scalability: Clusters are well suited to Internet service workloads, which are highly parallel (many independent simultaneous users) and for which the grain size typically corresponds to at most a few CPU-seconds on a commodity PC. For these workloads, large clusters can dwarf the power of the largest machines. For example, Inktomi's HotBot cluster contains 60 nodes with 120 processors, 30 GB of physical memory, and hundreds of commodity disks. Wal-Mart uses a cluster from TeraData with 768 processors and 16 terabytes of online storage.

Furthermore, the ability to grow clusters incrementally over time is a tremendous advantage in areas such as Internet service deployment, where capacity planning depends on a large number of unknown variables. Incremental scalability replaces capacity planning with relatively fluid reactionary scaling. Clusters correspondingly eliminate the "forklift upgrade", in which you must throw out the current machine (and related investments) and replace it via forklift with an even larger one.

High Availability: Clusters have natural redundancy due to the independence of the nodes: Each node has its own busses, power supply, disks, etc., so it is "merely" a matter of software to mask (possibly multiple simultaneous) transient faults. A natural extension of this capability is to temporarily disable a subset of nodes and then upgrade them in place ("hot upgrade"). Such capabilities are essential for network services, whose users have come to expect 24-hour uptime despite the inevitable reality of hardware and software faults due to rapid system evolution.

Commodity Building Blocks: The final set of advantages of clustering follows from the use of commodity building blocks rather than high-end, low-volume machines. The obvious advantage is cost/performance, since memory, disks, and nodes can all track the leading edge; for example, we changed the building block every time we grew the HotBot cluster, each time picking the reliable high volume previous-generation commodity units, helping to ensure stability and robustness. Furthermore, since many commodity vendors compete on service (particularly for PC hardware), it is easy to get high-quality configured nodes in 48 hours or less. In contrast, large SMPs typically have a lead time of 45 days, are more cumbersome to purchase, install, and upgrade, and are supported by a single vendor, so it is much harder to get help when difficulties arise. Once again, it is a "simple matter of software" to tie together a collection of possibly heterogeneous commodity building blocks.

To summarize, clusters have significant advantages in scalability, growth, availability, and cost. Although fundamental, these advantages are not easy to realize.

1.3 Challenges of Cluster Computing

There are a number of areas in which clusters are at a disadvantage relative to SMP's. In this section we describe some of these challenges and how they influenced the architecture we will propose in Section 2.

Administration: Administration is a serious concern for systems of many nodes. We leverage ideas in prior work [1], which describes how a unified monitoring/reporting framework with data visualization support was an effective tool for simplifying cluster administration.

Component vs. system replication: Each commodity PC in a cluster is not usually powerful enough to support an entire service, but can probably support some components of the service. Component-level rather than whole-system replication therefore allows commodity PCs to serve as the unit of incremental scaling, provided the software can be naturally decomposed into loosely coupled modules. We address this challenge by proposing an architecture in which each component has well-circumscribed functional responsibilities and is largely "interchangeable" with other components of the same type. For example, a cache node can run anywhere that a disk is available, and a worker that performs a specific kind of data compression can run anywhere that significant CPU cycles are available.

Partial failures: Component-level replication leads directly to the fundamental issue separating clusters from SMPs: the need to handle partial failures (i.e., the ability to survive and adapt to failures of subsets of the system). Traditional workstations and SMPs never face this issue, since the machine is either up or down.

Shared state: Unlike SMPs, clusters have no shared state. Although much work has been done to emulate global shared state through software distributed shared memory [33,34,36], we can improve performance and reduce complexity if we can avoid or minimize the need for shared state across the cluster.

These last two concerns, partial failure and shared state, lead us to focus on the sharing semantics actually required by network services.

1.4 BASE Semantics

We believe that the design space for network services can be partitioned according to the data semantics that each service demands. At one extreme is the traditional transactional database model with the ACID properties (atomicity, consistency, isolation, durability) [26], providing the strongest semantics at the highest cost and complexity. ACID makes no guarantees regarding availability; indeed, it is preferable for an ACID service to be unavailable than to function in a way that relaxes the ACID constraints. ACID semantics are well suited for Internet commerce transactions, billing users, or maintaining user profile information for personalized services.

For other Internet services, however, the primary value to the user is not necessarily strong consistency or durability, but rather high availability of data:

- **Stale** data can be temporarily tolerated as long as all copies of data eventually reach consistency after a short time (e.g., DNS servers do not reach consistency until entry timeouts expire [41]).
- **Soft state**, which can be regenerated at the expense of additional computation or file I/O, is exploited to improve performance; data is not durable.
- **Approximate** answers (based on stale data or incomplete soft state) delivered quickly may be more valuable than exact answers delivered slowly.

We refer to the data semantics resulting from the combination of these techniques as *BASE*—Basically Available, Soft State, Eventual Consistency. By definition, any data semantics that are not strictly ACID are BASE. BASE semantics allow us to handle partial failure in clusters with less complexity and cost. Like pioneering systems such as Grapevine [9] , BASE reduces the complexity of the service implementation, essentially trading consistency for simplicity; like later systems such as Bayou [21] that allow trading consistency for availability, BASE provides opportunities for better performance. For example, where ACID requires durable and consistent state across partial failures, BASE semantics often allows us to avoid communication and disk activity or to postpone it until a more convenient time.

In practice, it is simplistic to categorize every service as either ACID or BASE; instead, different components of services demand varying data semantics. Directories such as Yahoo! [64] maintain a database of soft state with BASE semantics, but keep user customization profiles in an ACID database. Transformation proxies [23,57] interposed between clients and servers transform Internet content on-the-fly; the transformed content is BASE data that can be regenerated by computation, but if the service bills the user per session, the billing should certainly be delegated to an ACID database.

We focus on services that have an ACID component, but manipulate primarily BASE data. Web servers, search/aggregation servers [58], caching proxies [14,44], and transformation proxies are all examples of such services; our framework supports a superset of these services by providing integrated support for the requirements of all four. As we will show, BASE semantics greatly simplify the implementation of fault tolerance and availability and permit performance optimizations within our framework that would be precluded by ACID.

2 Cluster-Based Scalable Service Architecture

In this section we propose a system architecture and service-programming model for building scalable network services on clusters. The architecture attempts to address both the challenges of cluster computing and the challenges of deploying network services, while exploiting clusters' strengths. We view our contributions as follows:

- A proposed system architecture for scalable network services that exploits the strengths of cluster computing, as exemplified by cluster-based servers such as TranSend and HotBot.
- Separation of the *content* of network services (i.e., what the services do) from their implementation, by encapsulating the "scalable network service" (SNS) requirements of high availability, scalability, and fault tolerance in a reusable layer with narrow interfaces.
- A programming model based on composition of stateless worker modules into new services. The model maps well onto our system architecture and numerous existing services map directly onto it.
- Detailed measurements of a production service that instantiates the architecture and validates our performance and reliability claims.

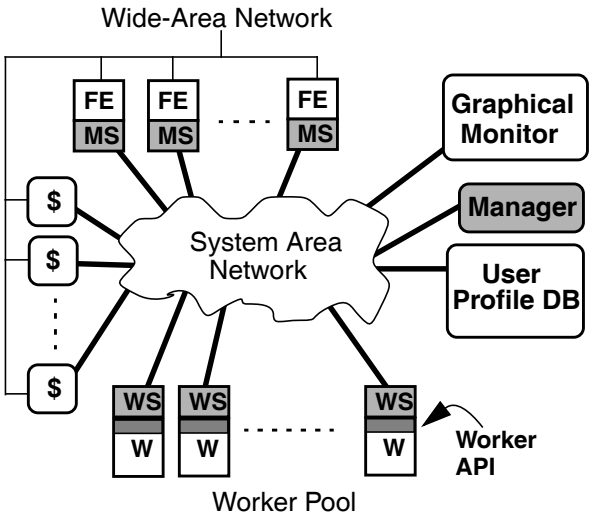


Figure 1: Architecture of a generic SNS. Components include front ends (FE), a pool of workers (W) some of which may be caches ($), a user profile database, a graphical monitor, and a fault-tolerant load manager, whose functionality logically extends into the manager stubs (MS) and worker stubs (WS).

In the remainder of this section we review the benefits and challenges of cluster computing and propose a network service architecture that exploits these observations and allows encapsulation of the SNS requirements. We then describe a programming model that minimizes service development effort by allowing implementation of new services entirely at the higher layers.

2.1 Proposed Functional Organization of an SNS

The above observations lead us to the software-component block diagram of a generic SNS shown Figure 1. Each physical workstation in a network of workstations (NOW [2]) supports one or more software components in the figure, but each component in the diagram is confined to one node. In general, the components whose tasks are naturally parallelizable are replicated for scalability, fault tolerance, or both. In our measurements (Section 4), we will argue that the performance demands on the non-replicated components are not significant for the implementation of a large class of services, and that the practical bottlenecks are bandwidth into and out of the system and bandwidth in the system area network (SAN).

Front Ends provide the interface to the SNS as seen by the outside world (e.g., HTTP server). They "shepherd" incoming requests by matching them up with the appropriate user profile from the customization database, and queueing them for service by one or more workers. Front ends maximize system throughput by maintaining state for many simultaneous outstanding requests, and can be replicated for both scalability and availability.

The **Worker Pool** consists of caches and service-specific modules that implement the actual service (data transformation/filtering, content aggregation, etc.) Each type of module may be instantiated zero or more times, depending on offered load.

The **Customization Database** stores user profiles that allow mass customization of request processing.

Service: Service-specific code
- Workers that present human interface to what TACC modules do, including device-specific presentation
- User interface to control the service

TACC: Transformation,Aggregation, Caching, Customization
- API for composition of stateless data transformation and content aggregation modules
- Uniform caching of original, post-aggregation and post-transformation data
- Transparent access to Customization database

SNS: Scalable Network Service support
- Incremental and absolute scalability
- Worker load balancing and overflow management
- Front-end availability, fault tolerance mechanisms
- System monitoring and logging

Figure 2: Scalable Network Service Layered Model

The **Manager** balances load across workers and spawns additional workers as offered load fluctuates or faults occur. When necessary, it may assign work to machines in the *overflow pool*, a set of backup machines (perhaps on desktops) that can be harnessed to handle load bursts and provide a smooth transition during incremental growth. The overflow pool is discussed in Section 2.2.3.

The **Graphical Monitor** for system management supports tracking and visualization of the system's behavior, asynchronous error notification via email or pager, and temporary disabling of system components for hot upgrades.

The **System-Area Network** provides a low-latency, high-bandwidth interconnect, such as switched 100-Mb/s Ethernet or Myrinet [43]. Its main goal is to prevent the interconnect from becoming the bottleneck as the system scales.

2.2 Separating Service Content From Implementation: A Reusable SNS Support Layer

Layered software models allow layers to be isolated from each other and allow existing software in one layer to be reused in different implementations. We observe that the components in the above architecture can be grouped naturally into three layers of functionality, as shown in Figure 2: SNS (scalable network service implementation), TACC (transformation, aggregation, caching, customization), and Service. The key contributions of our architecture are the reusability of the SNS layer, and the ability to add simple, stateless "building blocks" at the TACC layer and compose them in the Service layer. We discuss TACC in Section 2.3. The SNS layer, which we describe here, provides scalability, load balancing, fault tolerance, and high availability; it comprises the front ends, manager, SAN, and monitor in Figure 1.

2.2.1 Scalability

Components in our SNS architecture may be replicated for fault tolerance or high availability, but we also use replication to achieve scalability. When the offered load to the system saturates the capacity of some component class, more instances of that component can be launched on incrementally added nodes. The duties of our replicated components are largely independent of each other (because of the nature of the internet services' workload), which means the amount of additional resources required is a linear function of the increase in offered load. Although the components are mostly independent, they do have some dependence on the shared, non-replicated system components: the SAN, the resource manager, and possibly the user profile database. Our measurements in Section 4 confirm that even for very large systems, these shared components do not become a bottleneck.

The static partitioning of functionality between front ends and workers reflects our desire to keep workers as simple as possible, by localizing in the front ends the control decisions associated with satisfying user requests. In addition to managing the network state for outstanding requests, a front end encapsulates service-specific worker dispatch logic, accesses the profile database to pass the appropriate parameters to the workers, notifies the end user in a service-specific way (e.g., constructing an HTML page describing the error) when one or more workers fails unrecoverably, provides the user interface to the profile database, and so forth. This division of responsibility allows workers to remain simple and stateless, and allows the behavior of the service as a whole to be defined almost entirely in the front end. If the workers are analogous to processes in a Unix pipeline, the front end is analogous to an interactive shell.

2.2.2 Centralized Load Balancing

Load balancing is controlled by a centralized policy implemented in the manager. The manager collects load information from the workers, synthesizes load balancing hints based on the policy, and periodically transmits the hints to the front ends, which make local scheduling decisions based on the most recent hints. The load balancing and overflow policies are left to the system operator. We describe our experiments with load balancing and overflow in Section 4.5.

The decision to centralize rather than distribute load balancing is intentional: If the load balancer can be made fault tolerant, and if we can ensure it does not become a performance bottleneck, centralization makes it easier to implement and reason about the behavior of the load balancing policy. In Section 3.1.3 we discuss the evolution that led to this design decision and its implications for performance, fault tolerance, and scalability.

2.2.3 Prolonged Bursts and Incremental Growth

Although we would like to assume that there is a well-defined average load and that arriving traffic follows a Poisson distribution, burstiness has been demonstrated for Ethernet traffic [35], file system traffic [27], and Web requests [18], and is confirmed by our traces of web traffic (discussed later). In addition, Internet services can experience relatively rare but prolonged bursts of high load: after the recent landing of Pathfinder on Mars, its web site served over 220 million hits in a 4-day period [45]. Often, it is during such bursts that uninterrupted operation is most critical.

Our architecture includes the notion of an overflow pool for absorbing these bursts. The overflow machines are not dedicated to the service, and normally do not have workers running on them, but the manager can spawn workers on the overflow machines on demand when unexpected load bursts arrive, and release the machines when the burst subsides. In an institutional or corporate setting, the overflow pool could consist of workstations on individuals' desktops. Because worker nodes are already interchangeable, workers do not need to know whether they are running on a dedicated or an overflow node, since load balancing is handled externally. In addition to absorbing sustained bursts, the ability to temporarily harness overflow machines eases incremental growth: when the overflow machines are being recruited unusually often, it is time to purchase more dedicated nodes for the service.

2.2.4 Soft State for Fault Tolerance and Availability

The technique of constructing robust entities by relying on cached soft state refreshed by periodic messages from peers has been enormously successful in wide-area TCP/IP networks [5,20,39], another arena in which transient component failure is a

fact of life. Correspondingly, our SNS components operate in this manner, and monitor one another using *process peer fault tolerance*[1]: when a component fails, one of its peers restarts it (on a different node if necessary), while cached stale state carries the surviving components through the failure. After the component is restarted, it gradually rebuilds its soft state, typically by listening to multicasts from other components. We give specific examples of this mechanism in Section 3.1.3.

We use timeouts as an additional fault-tolerance mechanism, to infer certain failure modes that cannot be otherwise detected. If the condition that caused the timeout can be automatically resolved (e.g., if workers lost because of a SAN partition can be restarted on still-visible nodes), the manager performs the necessary actions. Otherwise, the SNS layer reports the suspected failure condition, and the service layer determines how to proceed (e.g., report the error or fall back to a simpler task that does not require the failed worker).

2.2.5 Narrow Interface to Service-Specific Workers

To allow new services to reuse all these facilities, the manager and front ends provide a narrow API, shown as the Manager Stubs and Worker Stubs in Figure 1, for communicating with the workers, the manager, and the graphical system monitor. The worker stub provides mechanisms for workers to implement some required behaviors for participating in the system (e.g., supplying load data to assist the manager in load balancing decisions and reporting detectable failures in their own operation). The worker stub hides fault tolerance, load balancing, and multithreading considerations from the worker code, which may use all the facilities of the operating system, need not be thread-safe, and can, in fact, crash without taking the system down. The minimal restrictions on worker code allow worker authors to focus instead on the *content* of the service, even using off-the-shelf code (as we have in TranSend) to implement the worker modules.

The manager stub linked to the front ends provides support for implementing the dispatch logic that selects which worker type(s) are needed to satisfy a request; since the dispatch logic is independent of the core load balancing and fault tolerance mechanisms, a variety of services can be built using the same set of workers.

2.3 TACC: A Programming Model for Internet Services

Having encapsulated the "SNS requirements" into a separate software layer, we now require a programming model for building the services themselves in higher layers. We focus on a particular subset of services, based on *transformation, aggregation, caching, and customization* of Internet content (TACC). Transformation is an operation on a single data object that changes its content; examples include filtering, transcoding, re-rendering, encryption, and compression. Aggregation involves collecting data from several objects and collating it in a prespecified way; for example, collecting all listings of cultural events from a prespecified set of Web pages, extracting the date and event information from each, and composing the result into a dynamically-generated "culture this week" page. Our initial implementation allows Unix-pipeline-like chaining of an arbitrary number of stateless transformations and aggregations; this results in a very general programming model that subsumes transformation proxies [22], proxy filters [67], customized information aggregators [59,13], and search engines. The selection of which workers to invoke for a particular request is service-specific and controlled outside the workers themselves; for example, given a collection of workers that convert images between pairs of encodings, a correctly chosen sequence of transformations can be used for general image conversion.

Customization represents a fundamental advantage of the Internet over traditional wide-area media such as television. Many online services, including the Wall Street Journal, the Los Angeles Times, and C/Net, have deployed "personalized" versions of their service as a way to increase loyalty and the quality of the service. Such mass customization requires the ability to track users and keep profile data for each user, although the content of the profiles differs across services. The customization database, a traditional ACID database, maps a user identification token (such as an IP address or cookie) to a list of key-value pairs for each user of the service. A key strength of the TACC model is that the appropriate profile information is automatically delivered to workers along with the input data for a particular user request; this allows the same workers to be reused for different services. For example, an image-compression worker can be run with one set of parameters to reduce image resolution for faster Web browsing, and a different set of parameters to reduce image size and bit depth for handheld devices. We have found composable, customizable workers to be a powerful building block for developing new services, and we discuss our experience with TACC and its continuing evolution in Section 5.

Caching is important because recomputing or storing data has become cheaper than moving it across the Internet. For example, a study of the UK National web cache has shown that even a small cache (400MB) can reduce the load on the network infrastructure by 40% [61], and SingNet, the largest ISP in Singapore, has saved 40% of its telecom charges using web caching [60]. In the TACC model, caches can store post-transformation (or post-aggregation) content and even intermediate-state content, in addition to caching original Internet content.

Many existing services are subsumed by the TACC model and fit well with it. (In Section 5.4 we describe some that do not.) For example, the HotBot search engine collects search results from a number of database partitions and collates the results. Transformation involves converting the input data from one form to another. In TranSend, graphic images can be scaled and filtered through a low-pass filter to tune them for a specific client or to reduce their size. A key strength of our architecture is the ease of composition of tasks; this affords considerable flexibility in the transformations and aggregations the service can perform, without requiring workers to understand service-specific dispatch logic, load balancing, etc., any more than programs in a Unix pipeline need to understand the implementation of the pipe mechanism.

We claim that a large number of interesting services can be implemented entirely at the service and TACC layers, and that relatively few services will benefit from direct modification to the SNS layer unless they have very specific low-level performance needs. In Section 5.1 we describe our experience adding functionality at both the TACC and service layers.

3 Service Implementation

This section focuses on the implementation of TranSend, a scalable Web distillation proxy, and compares it with HotBot. The goals of this section are to demonstrate how each component shown in Figure 1 maps into the layered architecture, to discuss relevant implementation details and trade-offs, and to provide the necessary context for the measurements we report in the next section.

1: Not to be confused with *process pairs*, a different fault-tolerance mechanism for hard-state processes, discussed in [6]. Process peers are similar to the fault tolerance mechanism explored in the early "Worm" programs [55] and to "Robin Hood/Friar Tuck" fault tolerance: "Each ghost-job would detect the fact that the other had been killed, and would start a new copy of the recently slain program within a few milliseconds. The only way to kill both ghosts was to kill them simultaneously (very difficult) or to deliberately crash the system." [50]

3.1 TranSend SNS Components

3.1.1 Front Ends

TranSend runs on a cluster of SPARCstation 10 and 20 machines, interconnected by switched 10 Mb/s Ethernet and connected to the dialup IP pool by a single 10 Mb/s segment. The TranSend front end presents an HTTP interface to the client population. A thread is assigned to each arriving TCP connection. Request processing involves fetching Web data from the caching subsystem (or from the Internet on a cache miss), pairing up the request with the user's customization preferences, sending the request and preferences to a pipeline of one or more *distillers* (the TranSend lossy-compression workers) to perform the appropriate transformation, and returning the result to the client. Alternatively, if an appropriate distilled representation is available in the cache, it can be sent directly to the client. A large thread pool allows the front end to sustain throughput and maximally exploit parallelism despite the large number of potentially long, blocking operations associated with each task, and provides a clean programming model. The production TranSend runs with a single front-end of about 400 threads.

3.1.2 Load Balancing Manager

Client-side JavaScript support [46] balances load across multiple front ends and masks transient front end failures, although other mechanisms such as round-robin DNS [12] or commercial routers [16] could also be used. For internal load balancing, TranSend uses a centralized manager whose responsibilities include tracking the location of distillers, spawning new distillers on demand, balancing load across distillers of the same class, and providing the assurance of fault tolerance and system tuning. We argue for a centralized as opposed to distributed manager because it is easier to change the load balancing policy and reason about its behavior; the next section discusses the fault-tolerance implications of this decision.

The manager periodically beacons its existence on an IP multicast group to which the other components subscribe. The use of IP multicast provides a level of indirection and relieves components of having to explicitly locate each other. When the front end has a task for a distiller, the manager stub code contacts the manager, which locates an appropriate distiller, spawning a new one if necessary. The manager stub caches the new distiller's location for future requests.

The worker stub attached to each distiller accepts and queues requests on behalf of the distiller and periodically reports load[2] information to the manager. The manager aggregates load information from all distillers, computes weighted moving averages, and piggybacks the resulting information on its beacons to the manager stub. The manager stub (at the front end) caches the information in these beacons and uses lottery scheduling [63] to select a distiller for each request. The cached information provides a backup so that the system can continue to operate (using slightly stale load data) even if the manager crashes. Eventually, the fault tolerance mechanisms (discussed in Section 3.1.3) restart the manager and the system returns to normal.

To allow the system to scale as the load increases, the manager can automatically spawn a new distiller on an unused node if it detects excessive load on distillers of a particular class. (The spawning and load balancing policies are described in detail in Section 4.5.) Another mechanism used for adjusting to bursts in load is overflow: if all the nodes in the system are used up, the manager can resort to starting up temporary distillers on a set of overflow nodes. Once the burst subsides, the distillers may be reaped.

3.1.3 Fault Tolerance and Crash Recovery

In the original prototype for the manager, information about distillers was kept as hard state, using a log file and crash recovery protocols similar to those used by ACID databases. Resilience against crashes was via process-*pair* fault tolerance, as in [6]: the primary manager process was mirrored by a secondary whose role was to maintain a current copy of the primary's state, and take over the primary's tasks if it detects that the primary has failed. In this scenario, crash recovery is seamless, since all state in the secondary process is up-to-date.

However, by moving entirely to BASE semantics, we were able to simplify the manager greatly and increase our confidence in its correctness. In TranSend, all state maintained by the manager is explicitly designed to be soft state. When a distiller starts up, it registers itself with the manager, whose existence it discovers by subscribing to a well-known multicast channel. If the distiller crashes before de-registering itself, the manager detects the broken connection; if the manager crashes and restarts, the distillers detect beacons from the new manager and re-register themselves. Timeouts are used as a backup mechanism to infer failures. Since all state is soft and is periodically beaconed, no explicit crash recovery or state mirroring mechanisms are required to regenerate lost state. Similarly, the front end does not require any special crash recovery code, since it can reconstruct its state as it receives the next few beacons from the manager.

With this use of soft state, each "watcher" process only needs to detect that its peer is alive (rather than mirroring the peer's state) and, in some cases, be able to restart the peer (rather than take over the peer's duties). Broken connections, timeouts, or loss of beacons are used to infer component failures and restart the failed process. The manager, distillers, and front ends are process peers:

- The manager reports distiller failures to the manager stubs, which update their caches of where distillers are running.
- The manager detects and restarts a crashed front end.
- The front end detects and restarts a crashed manager.

This process peer functionality is encapsulated within the manager stub code. Simply by linking against the stub, front ends are automatically recruited as process peers of the manager.

3.1.4 User Profile Database

The service interface to TranSend allows each user to register a series of customization settings, using either HTML forms or a Java/JavaScript combination applet. The actual database is implemented using *gdbm* because it is freely available and its performance is adequate for our needs: user preference reads are much more frequent than writes, and the reads are absorbed by a write-through cache in the front end.

3.1.5 Cache Nodes

TranSend runs Harvest object cache workers [10] on four separate nodes. Harvest suffers from three functional/performance deficiencies, two of which we resolved.

First, although a collection of Harvest caches can be treated as "siblings", by default all siblings are queried on each request, so that the cache service time would increase as the load increases even if more cache nodes were added. Therefore, for both scalability and improved fault tolerance, the manager stub can manage a number of separate cache nodes as a single virtual cache, hashing the key space across the separate caches and automatically re-hashing when cache nodes are added or removed. Second, we modified Harvest to allow data to be injected into it, allowing distillers (via

2: In the current implementation, distiller load is characterized in terms of the queue length at the distiller, optionally weighted by the expected cost of distilling each item.

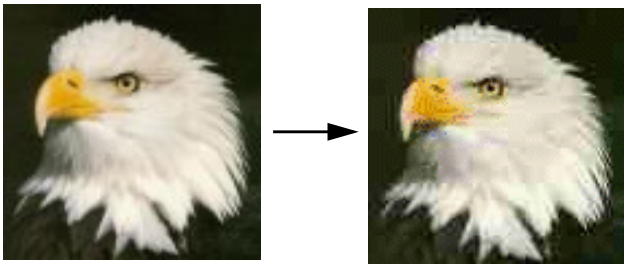


Figure 3: Scaling this JPEG image by a factor of 2 in each dimension and reducing JPEG quality to 25 results in a size reduction from 10KB to 1.5KB.

Figure 4: User Interface for manipulating preferences.

the worker stub) to store post-transformed or intermediate-state data into the large virtual cache. Finally, because the interface to each cache node is HTTP, a separate TCP connection is required for each cache request. We did not repair this deficiency due to the complexity of the Harvest code, and as a result some of the measurements reported in Section 4.4 are slightly pessimistic.

Caching in TranSend is only an optimization. All cached data can be thrown away at the cost of performance—cache nodes are workers whose only job is the management of BASE data.

3.1.6 Datatype-Specific Distillers

The second group of workers is the distillers, which perform transformation and aggregation. As a result, we were able to leverage a large amount of off-the-shelf code for our distillers. We have built three parameterizable distillers for TranSend: scaling and low-pass filtering of JPEG images using the off-the-shelf *jpeg-6a* library [29], GIF-to-JPEG conversion followed by JPEG degradation[3], and a Perl HTML "munger" that marks up inline image references with distillation preferences, adds extra links next to distilled images so that users can retrieve the original content, and adds a "toolbar" (Figure 4) to each page that allows users to control various aspects of TranSend's operation. The user interface for TranSend is thus controlled by the HTML distiller, under the direction of the user preferences from the front end.

Each of these distillers took approximately 5-6 hours to implement, debug, and optimize. Although pathological input data occasionally causes a distiller to crash, the process-peer fault tolerance guaranteed by the SNS layer means that we don't have to worry about eliminating all such possible bugs and corner cases from the system.

3.1.7 Graphical Monitor

Our extensible Tcl/Tk [48] graphical monitor presents a unified view of the system as a *single virtual entity.* Components of the system report state information to the monitor using a multicast group, allowing multiple monitors to run at geographically dispersed locations for remote management. The monitor can page or email the system operator if a serious error occurs, for example, if it stops receiving reports from some component.

The benefits of visualization are not just cosmetic: We can immediately detect by looking at the visualization panel what state the system as a whole is in, whether any component is currently causing a bottleneck (such as cache-miss time, distillation queueing delay, interconnect), what resources the system is using, and other such figures of interest.

3.1.8 How TranSend Exploits BASE

Distinguishing ACID vs. BASE semantics in the design of service components greatly simplifies TranSend's fault-tolerance and improves its availability. Only the user-profile database is ACID; everything else exploits some aspect of BASE semantics, both in manipulating application data (i.e., Web content) and in the implementation of the system components themselves.

Stale load balancing data: The load balancing data in the manager stub is slightly stale between updates from the manager, which arrive a few seconds apart. The use of stale data for the load balancing and overflow decisions improves performance and helps to hide faults, since using cached data avoids communicating with the source. Timeouts are used to recover from cases where stale data causes an incorrect load balancing choice. For example, if a request is sent to a worker that no longer exists, the request will time out and another worker will be chosen. From the standpoint of performance, as we will show in our measurements, the use of slightly stale data is not a problem in practice.

Soft state: The two advantages of soft state are improved performance from avoiding (blocking) commits and trivial recovery. Transformed content is cached and can be regenerated from the original (which may be also cached).

Approximate answers: Users of TranSend request objects that are named by the object URL and the user preferences, which are used to derive distillation parameters. However, if the system is too heavily loaded to perform distillation, it can return a somewhat different version from the cache; if the user clicks the "Reload" button later, they will get the distilled representation they asked for if the system now has sufficient resources to perform the distillation. If the required distiller has temporarily or permanently failed, the system can return the original content. In all cases, an approximate answer delivered quickly is more useful than the exact answer delivered slowly.

3.2 HotBot Implementation

In this section we highlight the principal differences between the implementations of TranSend and HotBot.The original Inktomi work, which is the basis of HotBot, predates the layered model and scalable server architecture presented here and uses ad hoc rather than generalized mechanisms in some places.

Front ends and service interface: HotBot runs on a mixture of single- and multiple-CPU SPARCstation server nodes, interconnected by Myrinet [43]. The HTTP front ends in HotBot run 50-80 threads per node and handle the presentation and customization of results based on user preferences and browser type. The presentation is performed using a form of "dynamic HTML" based on Tcl macros [54].

Load balancing: HotBot workers statically partition the search-engine database for load balancing. Thus each worker handles a subset of the database proportional to its CPU power, and every query goes to all workers in parallel.

Failure management: Unlike the workers in TranSend, HotBot worker nodes are not interchangeable, since each worker uses a local disk to store its part of the database. The original Inktomi nodes cross-mounted databases, so that there were always multiple nodes that could reach any database partition. Thus, when a node when down, other nodes would automatically take over responsibility for that data, maintaining 100% data availability with graceful degradation in performance.

3: We chose this approach after discovering that the JPEG representation is smaller and faster to operate on for most images, and produces aesthetically superior results.

Component	TranSend	HotBot
Load balancing	Dynamic, by queue lengths at worker nodes	Static partitioning of read-only data
Application layer	Composable TACC workers	Fixed search service application
Service layer	Worker dispatch logic, HTML / JavaScript UI	Dynamic HTML generation, HTML UI
Failure management	Centralized but fault-tolerant using process-peers	Distributed to each node
Worker placement	FE's and caches bound to their nodes	All workers bound to their nodes
User profile (ACID) database	Berkeley DB with read caches	Parallel Informix server
Caching	Harvest caches store pre- and post-transformation Web data	integrated cache of recent searches, for incremental delivery

Table 1: Main differences between TranSend and HotBot.

Since the database partitioning distributes documents randomly and it is acceptable to lose part of the database temporarily, HotBot moved to a model in which RAID storage handles disk failures, while fast restart minimizes the impact of node failures. For example, with 26 nodes the loss of one machine results in the database dropping from 54M to about 51M documents, which is still significantly larger than other search engines (such as Alta Vista at 30M).

The success of the fault management of HotBot is exemplified by the fact that during February 1997, HotBot was physically moved (from Berkeley to San Jose) without ever being down, by moving half of the cluster at a time and changing DNS resolution in the middle. Although various parts of the database were unavailable at different times during the move, the overall service was still up and useful—user feedback indicated that few people were affected by the transient changes.

User profile database: We expect commercial systems to use a real database for ACID components. HotBot uses Informix with primary/backup failover for the user profile and ad revenue tracking database, with each front end linking in an Informix SQL client. However, all other HotBot data is BASE, and as in TranSend, timeouts are used to recover from stale cluster-state data.

3.3 Summary

The TranSend implementation quite closely maps into the layered architecture presented in Section 2, while the HotBot implementation differs in the use of a distributed manager, static load balancing by data partitioning, and workers that are tied to particular machines. The careful separation of responsibility into different components of the system, and the layering of components according to the architecture, made the implementation complexity manageable.

4 Measurements of the TranSend Implementation

We took measurements of TranSend using a cluster of 15 Sun SPARC Ultra-1 workstations connected by 100 Mb/s switched Ethernet and isolated from external load or network traffic. For measurements requiring Internet access, the access was via a 10 Mb/s switched Ethernet network connecting our workstation to the outside world. In the following subsections we analyze the size distribution and burstiness characteristics of TranSend's expected workload, describe the performance of two throughput-critical components (the cache nodes and data-transformation workers) in isolation, and report on experiments that stress TranSend's fault tolerance, responsiveness to bursts, and scalability.

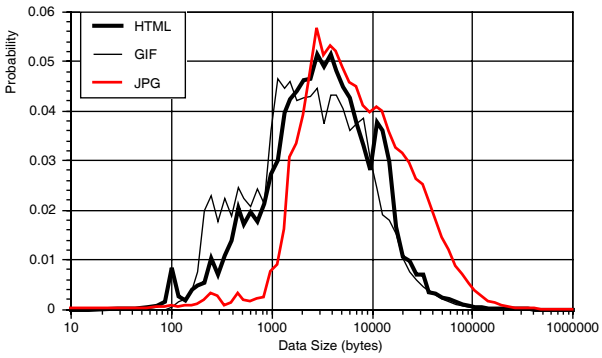


Figure 5: Distribution of content lengths for HTML, GIF, and JPEG files. The spikes to the left of the main GIF and JPEG distributions are error messages mistaken for image data, based on file name extension. Average content lengths: HTML - 5131 bytes, GIF - 3428 bytes, JPEG - 12070 bytes.

4.1 HTTP Traces and the Playback Engine

Many of the performance tests are based upon HTTP trace data that we gathered from our intended user population, namely the 25,000 UC Berkeley dialup IP users, up to 600 of whom may be connected via a bank of 14.4K or 28.8K modems. The modems' connection to the Internet passes through a single 10 Mb/s Ethernet segment; we placed a tracing machine running an IP packet filter on this segment for a month and a half, and unobtrusively gathered a trace of approximately 20 million (anonymized) HTTP requests. GIF, HTML, and JPEG were by far the three most common MIME types observed in our traces (50%, 22%, and 18%, respectively), and hence our three implemented distillers cover these common cases. Data for which no distiller exists is passed unmodified to the user.

Figure 5 illustrates the distribution of sizes occurring for these three MIME types. Although most content accessed on the web is small (considerably less than 1 KB), the average byte transferred is part of large content (3-12 KB). This means that the users' modems spend most of their time transferring a few, large files. It is the goal of TranSend to eliminate this bottleneck by distilling this large content into smaller, but still useful representations; data under 1 KB is transferred to the client unmodified, since distillation of such small content rarely results in a size reduction.

Figure 5 also reveals a number of interesting properties of the individual data types. The GIF distribution has two plateaus—one for data sizes under 1KB (which correspond to icons, bullets, etc.) and one for data sizes over 1KB (which correspond to photos or cartoons). Our 1KB distillation threshold therefore exactly separates these two classes of data, and deals with each correctly. JPEGs do not show this same distinction: the distribution falls of rapidly under the 1KB mark.

In order to realistically stress test TranSend, we created a high performance trace playback engine. The engine can generate requests at a constant (and dynamically tunable) rate, or it can faithfully play back a trace according to the timestamps in the trace file. We thus had fine-grained control over both the amount and nature of the load offered to our implementation during our experimentation.

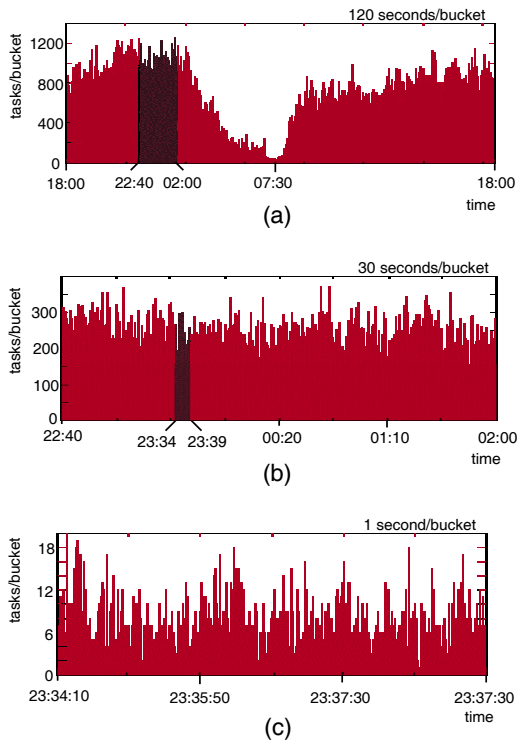


Figure 6: The number of requests per second for traced dialup IP users, showing burstiness across different time scales. (a) 24 hours with 2 minute buckets, 5.8 req/s avg., 12.6 req/s max. (b) 3 hr 20 min with 30 second buckets, 5.6 req/s avg., 10.3 req/s peak. (c) 3 min 20 sec, 8.1 req/s avg., 20 req/s peak.

4.2 Burstiness

Burstiness is a fundamental property of a great variety of computing systems, and can be observed across all time scales [18,27,35]. Our HTTP traces show that the offered load to our implementation will contain bursts—Figure 6 shows the request rate observed from the user base across a 24 hour, 3.5 hour, and 3.5 minute time interval. The 24 hour interval exhibits a strong 24 hour cycle that is overlaid with shorter time-scale bursts. The 3.5 hour and 3.5 minute intervals reveal finer grained bursts.

We described in Section 2.2.3 how our architecture allows an arbitrary subset of machines to be managed as an overflow pool during temporary but prolonged periods of high load. The overflow pool can also be used to absorb bursts on shorter time scales. We argue that there are two possible administrative avenues for managing the overflow pool:

1. Select an average desired utilization level for the dedicated worker pool. Since we can observe a daily cycle, this amounts to drawing a line across Figure 6a (i.e., picking a number of tasks/sec) such that the fraction of black under the line is the desired utilization level.
2. Select an acceptable percentage of time that the system will resort to the overflow pool. This amounts to drawing a line across Figure 6a such that the fraction of columns that cross the line is this percentage.[4]

Since we have measured the average number of requests/s that a distiller of a given class can handle, the number of tasks /s that we picked (from step 1 or 2 above) dictates how many distillers will need to be in the dedicated (non-overflow) pool.

4.3 Distiller Performance

If the system is behaving well, the distillation of images is the most computationally expensive task performed by TranSend. We measured

4: Note that the utilization level cannot necessarily be predicted given a certain acceptable percentage, and vice-versa.

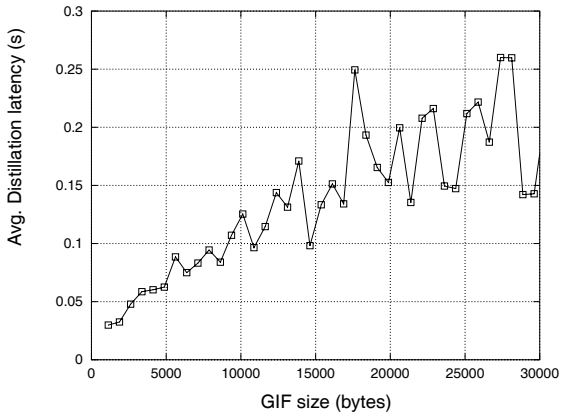


Figure 7: Average distillation latency vs. GIF size, based on GIF data gathered from the dialup IP trace.

the performance of our distillers by timing distillation latency as a function of input data size, calculated across approximately 100,000 items from the dialup IP trace file. Figure 7 shows that for the GIF distiller, there is an approximately linear relationship between distillation time and input size, although a large variation in distillation time is observed for any particular data size. The slope of this relationship is approximately 8 milliseconds per kilobyte of input. Similar results were observed for the JPEG and HTML distillers, although the HTML distiller is far more efficient.

4.4 Cache Partition Performance

In [10], a detailed performance analysis of the Harvest caching system is presented. We summarize the results here:

- The average cache hit takes 27 ms to service, including network and OS overhead, implying a maximum average service rate from each partitioned cache instance of 37 requests per second. TCP connection and tear-down overhead is attributed to 15 ms of this service time.
- 95% of all cache hits take less than 100 ms to service, implying cache hit rate has low variation.
- The miss penalty (i.e., the time to fetch data from the Internet) varies widely, from 100 ms through 100 seconds. This implies that should a cache miss occur, it is likely to dominate the end-to-end latency through the system, and therefore much effort should be expended to minimize cache miss rate.

As a supplement to these results, we ran a number of cache simulations to explore the relationship between user population size, cache size, and cache hit rate, using LRU replacement. We observed that the size of the user population greatly affects the attainable hit rate. Cache hit rate increases monotonically as a function of cache size, but plateaus out at a level that is a function of the user population size. For the user population observed across the traces (approximately 8000 people over the 1.5 month period), six gigabytes of cache space (in total, partitioned over all instances) gave us a hit rate of 56%. Similarly, we observed that for a given cache size, increasing the size of the user population increases the hit rate in the cache (due to an increase in locality across the users), until the point at which the sum of the users' working sets exceeds the cache size, causing the cache hit rate to fall.

From these results, we can deduce that the capacity of a single front end will be limited by the high cache miss penalties. The number of simultaneous, outstanding requests at a front end is equal to $N \times T$, where N is the number of requests arriving per second, and T is the average service time of a request. A high cache miss penalty implies that T will be large. Because two TCP connections (one between the client and front end, the other between the front end and a cache partition) and one thread context are maintained in the front end for each outstanding request, implying that front ends are vulnerable to state management and context switching overhead. As an example, for offered loads of 15 requests per second to a front end, we have observed 150-350 outstanding requests and therefore up to 700 open TCP connections and 300

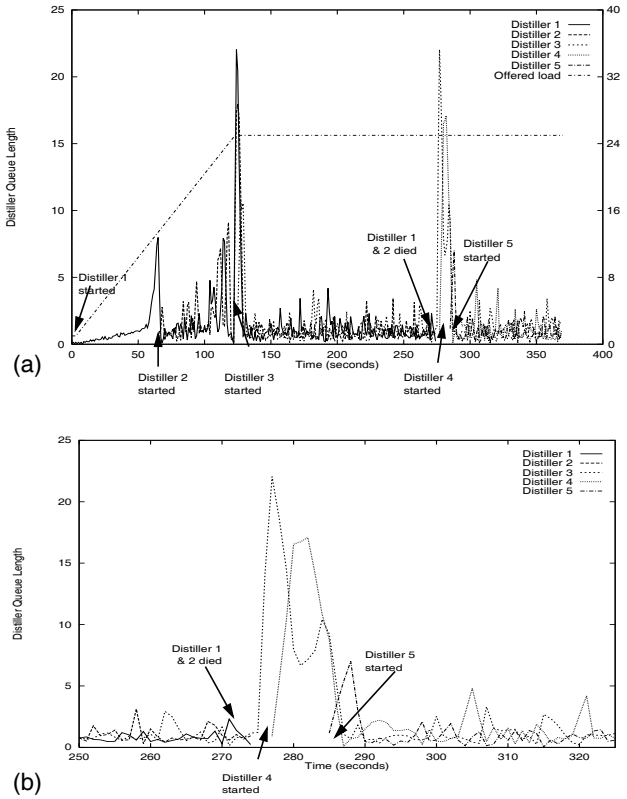


Figure 8: Distiller queue lengths observed over time as the load presented to the system fluctuates, and as distillers are manually brought down. (b) is an enlargement of (a).

active thread contexts at any given time. As a result, the front end spends more than 70% of its time in the kernel (as reported by the *top* utility) under this load. Eliminating this overhead is the subject of ongoing research.

4.5 Self Tuning and Load Balancing

TranSend uses queue lengths at the distillers as a metric for load balancing. As queue lengths grow due to increased load, the moving average of the queue length maintained by the manager starts increasing; when the average crosses a configurable threshold H, the manager spawns a new distiller to absorb the load. The threshold H maps to the greatest delay the user is willing to tolerate when the system is under high load. To allow the new distiller to stabilize the system, the spawning mechanism is disabled for D seconds; the parameter D represents a tradeoff between stability (rate of spawning and reaping distillers) and user-perceptible delay.

Figure 8(a) shows the variation in distiller queue lengths over time. The system was bootstrapped with one front end and the manager. On-demand spawning of the first distiller was observed as soon as load was offered. With increasing load, the distiller queue gradually increased until the manager decided to spawn a second distiller, which reduced the queue length of the first distiller and balanced the load across both distillers within five seconds. Continued increase in load caused a third distiller to start up, which again reduced and balanced the queue lengths within five seconds.

Figure 8(b) shows an enlarged view of the graph in Figure 8(a). During the experiment, we manually killed the first two distillers, causing the load on the remaining distiller to rapidly increase. The manager immediately reacted and started up a new distiller. Even after D seconds, the manager discovered that the system was overloaded and started up one more distiller, causing the load to stabilize.

When we first ran this experiment, we noticed rapid oscillations in queue lengths. Inspection revealed that since the front end's manager stubs only periodically received distiller queue length reports, they were making load balancing decisions based on stale data. To repair this, we changed the manager stub to keep a running estimate of the change in distiller queue lengths between successive reports; these estimates were sufficient to eliminate the oscillations. The data in Figure 8 reflects the modified load balancing functionality.

4.6 Scalability

To demonstrate the scalability of the system, we needed to eliminate two bottlenecks that limit the load we could offer: the overhead associated with having a very large number of open file descriptors, and the bottleneck 10Mb/s Ethernet connecting our cluster to the Internet. To do this, we prepared a trace file that repeatedly requested a fixed number of JPEG images, all approximately 10KB in size, based on the distributions we observed (Section 4.1). These images would then remain resident in the cache partitions, eliminating cache miss penalty and the resulting buildup of file descriptors in the front end. We recognize that although a non-zero cache miss penalty does not introduce any additional network, stable storage, or computational burden on the system, it does result in an increase in the amount of state in the front end, which as we mentioned in Section 4.4 limits the performance of a single front end. On the other hand, by turning off caching of distilled images, we force our system to re-distill the image every time it was requested, and in that respect our measurements are pessimistic relative to the system's normal mode of operation.

Our strategy for the experiment was as follows:

1. Begin with a minimal instance of the system: one front end, one distiller, the manager, and some fixed number of cache partitions. (Since for these experiments we repeatedly requested the same subset of images, the cache was effectively not tested.)
2. Increase the offered load until some system component saturates (e.g., distiller queues grow too long, front ends cannot accept additional connections, etc.).
3. Add more resources to the system to eliminate this saturation (in many cases the system does this automatically, as when it recruits overflow nodes to run more workers), and record the amount of resources added as a function of the increase in offered load, measured in requests per second.
4. Continue until the saturated resource cannot be replenished (i.e., we run out of hardware), or until adding more of the saturated resource no longer results in a linear or close-to-linear improvement in performance.

Table 2 presents the results of this experiment. At 24 requests per second, as the offered load exceeded the capacity of the single available distiller, the manager automatically spawned one additional distiller, and then subsequent distillers as necessary. At 87 requests per second, the Ethernet segment leading into the front end saturated, requiring a new front end to be spawned. We were unable to test the system at rates higher than 159 requests per second, as all of our cluster's machines were hosting distillers, front ends, or playback engines. We did observe nearly perfectly linear growth of the system over the scaled range: a distiller can handle approximately 23 requests per second, and a 100 Mb/s Ethernet segment into a front-end can handle approximately 70 requests per second.[5] We were unable to saturate the front end, the cache parti-

5: We believe that TCP connection setup and processing overhead is the dominating factor. Using a more efficient TCP implementation such as Fast Sockets [52] may alleviate this limitation, although more investigation is needed.

Requests/ Second	# Front Ends	# Distillers	Element that saturated
0-24	1	1	distillers
25-47	1	2	distillers
48-72	1	3	distillers
73-87	1	4	FE Ethernet
88-91	2	4	distillers
92-112	2	5	distillers
113-135	2	6	distillers & FE Ethernet
136-159	3	7	distillers

Table 2: Results of the scalability experiment

tions, or fully saturate the interior SAN during this experiment. We draw two conclusions from this result:

- Even with a commodity 100 Mb/s SAN, linear scaling is limited primarily by bandwidth into the system rather than bandwidth inside the system.
- Although we run TranSend on four SPARC 10's, a single Ultra-1 class machine would suffice to serve the entire dialup IP population of UC Berkeley (25,000 users officially, over 8000 of whom surfed during the trace).

Ultimately, the scalability of our system is limited by the shared or centralized components of the system, namely the user profile database, the manager, and the SAN. In our experience, neither the database nor the manager have ever been close to saturation. The main task of the manager (in steady state) is to accumulate load announcements from all distillers and multicast this information to the front ends. We conducted an experiment to test the capability of the manager to handle these load announcements. Nine hundred distillers were created on four machines. Each of these distillers generated a load announcement packet for the manager every half a second. The manager was easily able to handle this aggregate load of 1800 announcements per second. With each distiller capable of processing over 20 front end requests per second, the manager is computationally capable of sustaining a total number of distillers equivalent to 18000 requests per second. This number is nearly three orders of magnitude greater than the peak load ever seen on UC Berkeley's modem pool which is comparable to a modest-sized ISP. Similarly, HotBot's ACID database (parallel Informix server), used for ad revenue tracking and user profiles, can serve about 400 requests per second, significantly greater than HotBot's load.

On the other hand, SAN saturation is a potential concern for communication-intensive workloads such as TranSend's. The problem of optimizing component placement given a specific network topology, technology, and workload is an important topic for future research. As a preliminary exploration of how TranSend behaves as the SAN saturates, we repeated the scalability experiments using a 10 Mb/s switched Ethernet. As the network was driven closer to saturation, we noticed that most of our (unreliable) multicast traffic was being dropped, crippling the ability of the manager to balance load and the ability of the monitor to report system conditions.

One possible solution to this problem is the addition of a low-speed utility network to isolate control traffic from data traffic, allowing the system to more gracefully handle (and perhaps avoid) SAN saturation. Another possibility is to use a higher-performance SAN interconnect: a Myrinet [43] microbenchmark run on the HotBot implementation measured 32 MBytes/s all-pairs traffic between 40 nodes, far greater than the traffic experienced during the normal use of the system, suggesting that Myrinet will support systems of at least several tens of nodes.

5 Discussion

In previous sections we presented detailed measurements of a scalable network service implementation that confirmed the effectiveness of our layered architecture. In this section, we discuss some of the more interesting and novel aspects of our architecture, reflect on further potential applications of this research, and compare our work with others' efforts.

5.1 Extensibility: New Workers and Composition

One of our goals was to make the system easily extensible at the TACC and Service layers by making it easy to create workers and chain them together. Our HTML and JPEG distillers consist almost entirely of off-the-shelf code, and each took an afternoon to write. Debugging the pathological cases for the HTML distiller was spread out over a period of days—since the system masked transient faults by bypassing original content "around" the faulting distiller, we could only deduce the existence of bugs by noticing (using the Monitor display) that the HTML distiller had been restarted several times over a period of hours.

The other aspect of extensibility is the ease with which new services can be added by composing workers and modifying the service presentation interface. We now discuss several examples of new services in various stages of construction, indicating what must be changed in the TACC and Service layers for each. The services share the following common features, which make them amenable to implementation using our framework:

- Compute-intensive transformation or aggregation
- Computation is parallelizable with granularity of a few CPU seconds
- Substantial value added by mass customization
- Data manipulated has BASE semantics

We restrict our discussion here to services that can be implemented using the HTTP proxy model (i.e., transparent interposition of computation between Web clients and Web servers). The following applications have all been prototyped using TranSend.

Keyword Filtering: The keyword filter aggregator is very simple (about 10 lines of Perl). It allows users to specify a Perl regular expression as customization preference. This regular expression is then applied to all HTML before delivery. A simple example filter marks all occurrences of the chosen keywords with large, bold, red typeface.

Bay Area Culture Page: This service retrieves scheduling information from a number of cultural pages on the web, and collates the results into a single, comprehensive calendar of upcoming events, bounded by dates stored as part of each user's profile. The service is implemented as a single aggregator in the TACC layer, and is composed with the unmodified TranSend service layer, delivering the benefits of distillation automatically. This service exploits BASE "approximate answers" semantics at the application layer: extremely general, layout-independent heuristics are used to extract scheduling information from the cultural pages. About 10-20% of the time, the heuristics spuriously pick up non-date text (and the accompanying non-descriptions of events), but the service is still useful and users simply ignore spurious results. Early experience with services such as this one suggest that our SNS architecture may be a promising platform for deploying certain kinds of simple network agents.

TranSend Metasearch: The metasearch service is similar to the Bay Area Culture Page in that it collates content from other sources in the Internet. This content, however, is dynamically produced—an aggregator accepts a search string from a user, queries a number of popular search engines, and collates the top results from each into a single result page. Commercial metasearch engines already exist [58], but the TranSend metasearch engine was implemented using 3 pages of Perl code in roughly 2.5 hours, and inherits scalability, fault tolerance, and high availability from the SNS layer.

Anonymous Rewebber: Just as anonymous remailer chains [24] allow email authors to anonymously disseminate their content, an *anonymous rewebber network* allows web authors to anonymously publish their content. The rewebber described in [25] was implemented in one week using our TACC architecture. The rewebber's workers perform encryption and decryption, its user profile database maintains public key information for anonymous servers, and its cache stores decrypted versions of frequently accessed pages. Since encryption and decryption of distinct pages requested by independent users is both computationally intensive and highly parallelizable, this service is a natural fit for our architecture.

Real Web Access for PDAs and Smart Phones: We have already extended TranSend to support graphical Web browsing on the USR PalmPilot [62], a typical "thin client" device. Previous attempts to provide Web browsing on such devices have foundered on the severe limitations imposed by small screens, limited computing capability, and austere programming environments, and virtually all have fallen back to simple text-only browsing. But the ability of our architecture to move complexity into the service workers rather than the client allows us to approach this problem from a different perspective. We have built TranSend workers that output simplified markup and scaled-down images ready to be "spoon fed" to an extremely simple browser client, given knowledge of the client's screen dimensions and font metrics. This greatly simplifies client-side code since no HTML parsing, layout, or image processing is necessary, and as a side benefit, the smaller and more efficient data representation reduces transmission time to the client.

5.2 Economic Feasibility

Given the improved quality of service provided by TranSend, an interesting question is the additional cost required to operate this service. From our performance data, a US$5000 Pentium Pro server should be able to support about 750 modems, or about 15,000 subscribers (assuming a 20:1 subscriber to modem ratio). Amortized over 1 year, the marginal cost per user is an amazing 25 cents/month.

If we include the savings to the ISP due to a cache hit rate of 50% or more, as we observed in our cache experiments, then we can eliminate the equivalent of 1-2 T1 lines per TranSend installation, which reduces operating costs by about US$3000 per month. Thus, we expect that the server would pay for itself in only two months. In this argument we have ignored the cost of administration, which is nontrivial, but we believe administration costs for TranSend would be minimal— we run TranSend at Berkeley with essentially no administration except for feature upgrades and bug fixes, both of which are performed without bringing the service down.

5.3 Related Work

Content transformation by proxy: Filtering and on-the-fly compression have become particularly popular for HTTP [31], whose proxy mechanism was originally intended for users behind security firewalls. The mechanism has been used to shield clients from the effects of poor (especially wireless) networks [22,37], perform filtering [67] and anonymization, and perform value-added transformations on content, including Kanji transcoding [56], Kanji-to-GIF conversion [65], application-level stream transducing [13,59], and personalized agent services for Web browsing [7].

Fault tolerance and high availability: The Worm programs [55] are an early example of process-peer fault tolerance. Tandem Computer and others explored a related mechanism, process-pair fault tolerance, [6] in which a secondary (backup) process ran in parallel with the primary and maintained a mirror of the primary's internal state by processing the same message traffic as the primary, allowing it to immediately replace the primary in the event of failure. Tandem also advocated the use of simple "building blocks" to ensure high availability. The Open Group SHAWS project [49] plans to build scalable highly available web servers using a fault tolerance toolkit called CORDS, but that project is still in progress.

BASE: Grapevine [9] was an important early example of trading consistency for simplicity; Bayou [21] later explored trading consistency for availability in application-specific ways, providing an operational spectrum between ACID and BASE for a distributed database. The use of soft state to provide improved performance and increase fault tolerance robustness has been well explored in the wide-area Internet, in the context of IP packet routing [39], multicast routing [20], and wireless TCP optimizations such as TCP Snoop [5]; the lessons learned in those areas strongly influenced our design philosophy for the TACC server architecture.

Load balancing and scaling: WebOS [66] and SWEB++ [3] have exploited the extensibility of client browsers via Java and JavaScript to enhance scalability of network-based services by dividing labor between the client and server. We note that our system does not preclude, and in fact benefits from, exploiting intelligence and computational resources at the client, as we do for the TranSend user interface and coarse-grained load balancing. However, as discussed in the Introduction, we expect the utility of centralized, highly-available services to continue to increase, and this cannot occur without the growth path provided by linear incremental scalability in the SNS sense.

5.4 Future Work

Our past work on adaptation via distillation [23,22] described how distillation could be dynamically tuned to match the behavior of the user's network connection, and we have successfully demonstrated adaptation to network changes by combining our original WWW proxy prototype with the Event Notification mechanisms developed by Welling and Badrinath [4], and plan to leverage these mechanisms to provide an adaptive solution for Web access from wireless clients.

We have not investigated how well our proposed architecture works outside the Internet-server domain. In particular, we do not believe it will work well for write-intensive services where the writes carry hard state or where strong consistency is desired, such as commerce servers, file systems, or online voting systems.

The programming model for TACC services is still embryonic. We plan to develop it into a well-defined programming environment with an SDK, and we will encourage our colleagues to author services of their own using our system.

Previous research into operating systems support for busy Internet servers [32, 42] has identified inadequacies in OS implementations and the set of abstractions available to applications. We plan to investigate similar issues related specifically to cluster-

based middleware services, as motivated by our observations in Section 4.4.

6 Conclusions

We proposed a layered architecture for cluster-based scalable network services. We identified challenges of cluster-based computing, and showed how our architecture addresses these challenges. The architecture is reusable: authors of new network services write and compose stateless workers that transform, aggregate, cache, and customize (TACC) Internet content, but are shielded from the software complexity of automatic scaling, high availability, and failure management. We argued that a large class of network services can get by with BASE, a weaker-than-ACID data semantics that results from the combination of trading consistency for availability and exploiting soft state for performance and failure management.

We discussed in depth the design and implementation of two cluster-based scalable network services: the TranSend distillation Web proxy and the HotBot search engine. Using extensive client traces, we conducted detailed performance measurements of TranSend. While gathering these measurements, we scaled TranSend up to 10 Ultra-1 workstations serving 159 web requests per second, and demonstrated that a single such workstation is sufficient to serve the needs of the entire 600 modem UC Berkeley dialup IP bank.

Since the class of cluster-based scalable network services we have identified can substantially increase the value of Internet access to end users while remaining cost-efficient to deploy and administer, we believe that cluster-based value-added network services will become an important Internet-service paradigm.

7 Acknowledgments

This paper has benefited from the detailed and perceptive comments of our reviewers, especially our shepherd Hank Levy. We thank Randy Katz and Eric Anderson for their detailed readings of early drafts of this paper, and David Culler for his ideas on TACC's potential as a model for cluster programming. Ken Lutz and Eric Fraser configured and administered the test network on which the TranSend scaling experiments were performed. Cliff Frost of the UC Berkeley Data Communications and Networks Services group allowed us to collect traces on the Berkeley dialup IP network and has worked with us to deploy and promote TranSend within Berkeley. Undergraduate researchers Anthony Polito, Benjamin Ling, and Andrew Huang implemented various parts of TranSend's user profile database and user interface. Ian Goldberg and David Wagner helped us debug TranSend, especially through their implementation of the rewebber.

8 References

[1] E. Anderson and David A. Patterson. *Extensible, Scalable Monitoring For Clusters of Computers.* Proc. 1997 Large Installation System Administration Confere (LISA XI), to appear.

[2] T. E. Anderson et al. *The Case for NOW (Networks of Workstations).* IEEE Micro, February 1995.

[3] D. Andresen, T. Yang, O. Egecioglu, O. H. Ibarra, and T. R. Smith. *Scalability Issues for High Performance Digital Libraries on the World Wide Web.*Proceedings of ADL '96, Forum on Research and Technology Advances in Digital Libraries, IEEE, Washington D.C., May 1996

[4] B. R. Badrinath and G. Welling. *A Framework for Environment Aware Mobile Applications*. International Conference on Distributed Computing Systems, May 1997 (to appear)

[5] H. Balakrishnan, S. Seshan, E. Amir, R. Katz. *Improving TCP/IP Performance over Wireless Networks.* Proeedings. of the 1st ACM Conference on Mobile Computing and Networking, Berkeley, CA, November 1995.

[6] J. F. Bartlett. *A NonStop Kernel.* Proc. 8th SOSP and Operating Systems Review 15(5), December 1981

[7] Rob Barrett, Paul Maglio, and Daniel Kellem. *How to Personalize the Web.* Proc. CHI 97.

[8] Berkeley Home IP Service FAQ. http://ack.berkeley.edu/dcns/modems/hip/hip_faq.html.

[9] A.D. Birrell et al. *Grapevine: An Exercise in Distributed Computing.* Communications of the ACM 25(4), Feb. 1984.

[10] C.M. Bowman et al. Harvest: *A Scalable, Customizable Discovery and Access System.* Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, August 1994

[11] Tim Bray. *Measuring the Web.* Proc. WWW-5, Paris, May 1996.

[12] T. Brisco. *RFC 1764: DNS Support for Load Balancing*, April 1995.

[13] C. Brooks, M.S. Mazer, S. Meeks and J. Miller. *Application-Specific Proxy Servers as HTTP Stream Transducers.* Proc. WWW-4, Boston, May 1996. http://www.w3.org/pub/Conferences/WWW4/Papers/56.

[14] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz and K. J. Worrell. *A Hierarchical Internet Object Cache*. Proceedings of the 1996 Usenix Annual Technical Conference, 153-163, January 1996.

[15] D.Clark. *Policy Routing in Internet Protocols.* Internet Request for Comments 1102, May 1989,

[16] Cisco Systems. *Local Director*. http://www.cisco.com/warp/public/751/lodir/index.html.

[17] F. J. Corbató and V. A. Vyssotsky. Introduction and Overview of the Multics System. *AFIPS Conference Proceedings*, 27, 185-196, (1965 Fall Joint Computer Conference), 1965. http://www.lilli.com/fjcc1.html

[18] M.E. Crovella and A. Bestavros. *Explaining World Wide Web Traffic Self-Similarity*. Tech Rep. TR-95-015, Computer Science Department, Boston University, October 1995.

[19] P. B. Danzig, R. S. Hall and M. F. Schwartz. *A Case for Caching File Objects Inside Internetworks*. Proceedings of SIGCOMM '93. 239-248, September 1993.

[20] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. *An Architecture for Wide-Area Multicast Routing.* Proceedings of SIGCOMM '94, University College London, London, U.K., September 1994.

[21] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, B. Welch. *The Bayou Architecture: Support for Data Sharing Among Mobile Users.*

[22] A. Fox and E. A. Brewer. *Reducing WWW Latency and Bandwidth Requirements via Real-Time Distillation.* Proc. WWW-5, Paris, May 1996.

[23] A. Fox, S. D. Gribble, E. Brewer and E. Amir. *Adapting to Network and Client Variation Via On-Demand Dynamic Distillation.* Proceedings of ASPLOS-VII, Boston, October 1996.

[24] Ian Goldberg, David Wagner, and Eric Brewer. *Privacy-enhancing Technologies for the Internet*. Proc. of IEEE Spring COMPCON, 1997

[25] Ian Goldberg and David Wagner. *TAZ Servers and the Rewebber Network: Enabling Anonymous Publishing on the World Wide Web*. Unpublished manuscript, May 1997, available at http://www.cs.berkeley.edu/~daw/cs268/.

[26] J. Gray. *The Transaction Concept: Virtues and Limitations*. Proceedings of VLDB. Cannes, France, September 1981, 144-154.

[27] S.D. Gribble, G.S. Manku, and E. Brewer. *Self-Similarity in File-Systems: Measurement and Applications*. Unpublished, available at http://www.cs.berkeley.edu/~gribble/papers/papers.html

[28] T.R. Halfhill. *Inside the Web PC*. Byte Magazine, March 1996.

[29] Independent JPEG Group. *jpeg6a* library.

[30] Inktomi Corporation: *The Inktomi Technology Behind HotBot*. May 1996. http://www.inktomi.com/whitepap.html.

[31] Internet Engineering Task Force. *Hypertext Transfer Protocol—HTTP 1.1.* RFC 2068, March 1, 1997.

[32] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. *Server Operating Systems*. Proceedings of the SIGOPS European Workshop, September 1996.

[33] P. Keleher, A. Cox, and W. Zwaenepoel. *Lazy Release Consistency for Software Distributed Shared Memory*. Proceedings of the 19th Annual Symposium on Computer Architecture. May, 1992.

[34] P. Keleher, A. Cox, S. Swarkadas, and W. Zwaenepoel. *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems*. Proceedings of the 1994 Winter USENIX Conference, January, 1994.

[35] W. Leland, M.S. Taqqu, W. Willinger, and D.V. Wilson. *On the Self-Similar Nature of Ethernet Traffic (extended version).* IEEE/ACM Transactions on Network v2, February 1994.

[36] K. Li. *Shared Virtual Memory on Loosely Coupled Microprocessors*. PhD Thesis, Yale University, September 1986.

[37] M. Liljeberg et al. *Enhanced Services for World Wide Web in Mobile WAN Environment*. University of Helsinki CS Technical Report No. C-1996-28, April 1996.

[38] Bruce A. Mah. *An Empirical Model of HTTP Network Traffic*. Proc. INFOCOM 97, Kobe, Japan, April 1997.

[39] J. McQuillan, I Richer, E. Rosen. *The New Routing Algorithm for the ARPANET.* IEEE Transactions on Communications COM-28, No. 5, pp. 711-719 , May 1980.

[40] M. Meeker and C. DePuy. *The Internet Report*. Morgan Stanley Equity Research, April 1996. http://www.mas.com/misc/inet/morganh.html

[41] P.V. Mockapetris and K.J. Dunlap. *Development of the Domain Name System.* ACM SIGCOMM Computer Communication Review, 1988.

[42] Jeffrey C. Mogul. *Operating Systems Support for Busy Internet Servers*. Proceedings of HotOS-V, Orcas Island, Washington, May 1995.

[43] Myricom Inc. *Myrinet: A Gigabit Per Second Local Area Network*. IEEE-Micro, Vol.15, No.1, February 1995, pp. 29-36.

[44] National Laboratory for Applied Network Research. *The Squid Internet Object Cache*. http://squid.nlanr.net.

[45] National Aeronautics and Space Administration. *The Mars Pathfinder Mission Home Page*. http://mpfwww.jpl.nasa.gov/default1.html.

[46] Netscape Communications Corporation. *Netscape Proxy Automatic Configuration*. http://home.netscape.com/eng/mozilla/2.02/relnotes/unix-2.02.html#Proxies.

[47] Nokia Communicator 9000 Press Release. Available at http://www.club.nokia.com/support/9000/press.html.

[48] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[49] Open Group Research Institute. *Scalable Highly Available Web Server Project (SHAWS).* http://www.osf.org/RI/PubProjPgs/SFTWWW.htm

[50] Eric S. Raymond, ed. *The New Hackers' Dictionary*. Cambridge, MA: MIT Press, 1991. Also http://www.ccil.org/jargon/jargon.html.

[51] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, J. Ried. *GroupLens: An Open Architecture for Collaborative Filtering of Netnews*. Proceedings of 1994 Conference on Computer Supported Cooperative Work, Chapel Hill, NC.

[52] S. H. Rodrigues and T. E. Anderson. *High-Performance Local-Area Communication Using Fast Sockets.* Proc. 1997 Winter USENIX, Anaheim, CA.

[53] Jacques A.J. Roufs. *Perceptual Image Quality: Concept and Measurement.* Philips Journal of Research, 47:3-14, 1992.

[54] A. Sah, K. E. Brown and E. Brewer. *Programming the Internet from the Server-Side with Tcl and Audience1*. Proceedings of Tcl96, July 1996.

[55] J. F. Shoch and J. A. Hupp. *The "Worm" Programs—Early Experience with a Distributed System*. CACM 25(3):172-180, March 1982.

[56] Y. Sato. DeleGate Server. Documentation available at http://www.aubg.edu:8080/cii/src/delegate3.0.17/doc/Manual.txt.

[57] B. Schilit and T. Bickmore. *Digestor: Device-Independent Access to the World Wide Web.* Proc. WWW-6, Santa Clara, CA, April 1997.

[58] E. Selberg, O. Etzioni and G. Lauckhart. *Metacrawler: About Our Service*. http://www.metacrawler.com/about.html.

[59] M.A. Schickler, M.S. Mazer, and C. Brooks. *Pan-Browser Support for Annotations and Other Meta-Information on the World Wide Web.* Proc. WWW-5, Paris, May 1996. http://www5conf.inria.fr/fich_html/papers/P15/Overview.html

[60] SingNet (Singapore ISP). *Heretical Caching Effort for SingNet Customers*. http://www.singnet.com.sg/cache/proxy

[61] N. Smith. *The UK National Web Cache - The State of the Art. Proc*. WWW-5, paris, May 1996. http://www5conf.inria.fr/fich_html/papers/P45/Overview.html

[62] US Robotics Palm Pilot home page - http://www.usr.com/palm/.

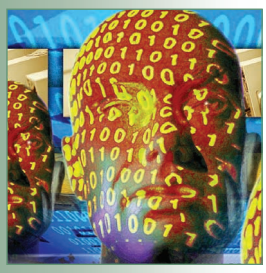
[63] C. Waldspurger and W. Weihl. *Lottery Scheduling: Flexible Proportional Share Resource Management*. Proceedings of the First OSDI, November 1994.

[64] Yahoo!, Inc. http://www.yahoo.com

[65] Kao-Ping Yee. Shoduoka Mediator Service. http://www.lfw.org/shodouka.

[66] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. *Using Smart Clients to Build Scalable Services.* Proceedings of Winter 1997 USENIX, January 1997.

[67] B. Zenel. *A Proxy Based Filtering Mechanism for the Mobile Environment*. Ph.D. Thesis Proposal, Department of Computer Science, Columbia University, March 1996.



Copyright 2001, IEEE Computer
Do not distribute.
See http://www.computer.org/internet/ic2001/w4046abs.htm

Lessons from Giant-Scale Services

Giant Web services require new tools and methods for issues of scale, availability, and evolution.

Eric A. Brewer
University of California, Berkeley, and Inktomi Corporation

Web portals and ISPs such as AOL, Microsoft Network, and Yahoo have grown more than tenfold in the past five years. Despite their scale, growth rates, and rapid evolution of content and features, these sites and other "giant-scale" services like instant messaging and Napster must be always available. Many other major Web sites such as eBay, CNN, and Wal-Mart, have similar availability requirements. In this article, I look at the basic model for such services, focusing on the key real-world challenges they face – high availability, evolution, and growth – and developing some principles for attacking these problems.

This is very much an "experience" article. Few of the points I make here are addressed in the literature, and most of my conclusions take the form of principles and approaches rather than absolute quantitative evaluations. This is due partly to my focus on high-level design, partly to the newness of the area, and partly to the proprietary nature of some of the information (which represents 15-20 very large sites). Nonetheless, the lessons are easy to understand and apply, and they simplify the design of large systems.

The Basic Model

I focus on "infrastructure services" – Internet-based systems that provide instant messaging, wireless services such as iMode, and so on. These services primarily reside remotely from the user, although they might include local access software, such as a browser. My discussion is limited primarily to single-site, single-owner, well-connected clusters, which may be part of a larger service, as in the case of e-mail servers. I do not cover wide-area issues such as network partitioning, low or intermittent bandwidth, or multiple administrative domains. There are many other important challenges that I do not address here, including service monitoring and configuration, network quality of service (QoS), security, and logging and log analysis.

This article is essentially about bridging the gap between the basic building blocks of giant-scale services and the real-world scalability and availability they require. It focuses on high availability and the related issues of replication, graceful degradation, disaster tolerance, and online evolution.

Database management systems (DBMs) are an important part of many large sites,

 1089-7801/01/$10.00 © 2001 IEEE

but I intentionally ignore them here because they are well studied elsewhere and because the issues in this article are largely orthogonal to the use of databases.

Advantages

The basic model that giant-scale services follow provides some fundamental advantages:

- *Access anywhere, anytime*. A ubiquitous infrastructure facilitates access from home, work, airport, and so on.
- *Availability via multiple devices*. Because the infrastructure handles most of the processing, users can access services with devices such as set-top boxes, network computers, and smart phones, which can offer far more functionality for a given cost and battery life.
- *Groupware support*. Centralizing data from many users allows service providers to offer group-based applications such as calendars, teleconferencing systems, and group-management systems such as Evite (http://www.evite.com/).
- *Lower overall cost*. Although hard to measure, infrastructure services have a fundamental cost advantage over designs based on stand-alone devices. Infrastructure resources can be multiplexed across active users, whereas end-user devices serve at most one user (active or not). Moreover, end-user devices have very low utilization (less than 4 percent), while infrastructure resources often reach 80 percent utilization. Thus, moving anything from the device to the infrastructure effectively improves efficiency by a factor of 20. Centralizing the administrative burden and simplifying end devices also reduce overall cost, but are harder to quantify.
- *Simplified service updates*. Perhaps the most powerful long-term advantage is the ability to upgrade existing services or offer new services without the physical distribution required by traditional applications and devices. Devices such as Web TVs last longer and gain usefulness over time as they benefit automatically from every new Web-based service.

Components

Figure 1 shows the basic model for giant-scale sites. The model is based on several assumptions. First, I assume the service provider has limited control over the clients and the IP network. Greater control might be possible in some cases, however, such as with intranets. The model also assumes that queries drive the service. This is true for most common protocols including HTTP, FTP, and variations of RPC. For example, HTTP's basic primitive, the "get" command, is by definition a query. My third assumption is that read-only queries greatly outnumber updates (queries that affect the persistent data store). Even sites that we tend to think of as highly transactional, such as e-commerce or financial sites, actually have this type of "read-mostly" traffic[1]: Product evaluations (reads) greatly outnumber purchases (updates), for example, and stock quotes (reads) greatly outnumber stock trades (updates). Finally, as the sidebar, "Clusters in Giant-Scale Services" (next page) explains, all giant-scale sites use clusters.

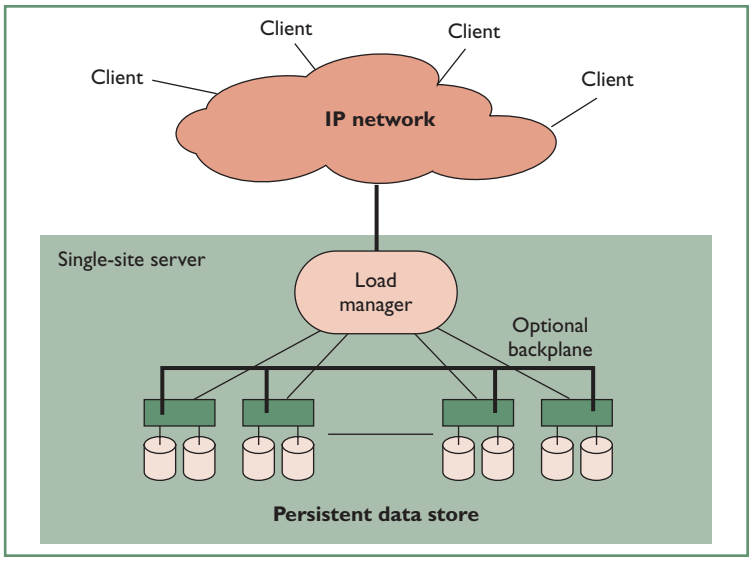


Figure 1. The basic model for giant-scale services. Clients connect via the Internet and then go through a load manager that hides down nodes and balances traffic.

The basic model includes six components:

- *Clients*, such as Web browsers, standalone e-mail readers, or even programs that use XML and SOAP initiate the queries to the services.
- The best-effort *IP network*, whether the public Internet or a private network such as an intranet, provides access to the service.
- The *load manager* provides a level of indirection between the service's external name and the servers' physical names (IP addresses) to preserve the external name's availability in the presence of server faults. The load manager balances load among active servers. Traffic might flow through proxies or firewalls before the load manager.
- *Servers* are the system's workers, combining CPU, memory, and disks into an easy-to-replicate unit.

Clusters in Giant-Scale Services

Table A. Example clusters for giant-scale services.

Service	Nodes	Queries	Nodes
AOL Web cache	>1,000	10B/day	4-CPU DEC 4100s
Inktomi search engine	>1,000	>80M/day	2-CPU Sun Workstations
Geocities	>300	>25M/day	PC Based
Anonymous Web-based e-mail	>5,000	>1B/day	FreeBSD PCs

Clusters are collections of commodity servers that work together on a single problem. Giant-scale services use clusters to meet scalability requirements that exceed those of any single machine. Table A shows some representative clusters and their traffic.

Other examples include some of Exodus Communications' data centers, which house several thousand nodes that support hundreds of different services; and AOL's recently added US$520 million data center, which is larger than three football fields and filled almost entirely with clusters. Although I assume that all giant-scale services use clusters, it is useful to review the driving forces behind their use.

- *Absolute scalability*. A successful network service must scale to support a substantial fraction of the world's population. Users spend increasingly more time online each day, and analysts predict that about 1.1 billion people will have some form of infrastructure access within the next 10 years.
- *Cost and performance*. Although a traditionally compelling reason for using clusters, hardware cost and performance are not really issues for giant-scale services. No alternative to clusters can match the required scale, and hardware cost is typically dwarfed by bandwidth and operational costs.
- *Independent components*. Users expect 24-hour service from systems that consist of thousands of hardware and software components. Transient hardware failures and software faults due to rapid system evolution are inevitable, but clusters simplify the problem by providing (largely) independent faults.
- *Incremental scalability*. Clusters should allow for scaling services as needed to account for the uncertainty and expense of growing a service. Nodes have a three-year depreciation lifetime, for example, and should generally be replaced only when they no longer justify their rack space compared to new nodes. Given Moore's law, a unit of rack space should quadruple in computing power over three years, and actual increases appear to be faster due to improvements in packaging and disk size.

Given these advantages, we view commodity nodes as the basic building blocks for giant-scale services. The goal is then to exploit these advantages, particularly independent faults and incremental scalability, into our higher-level goals of availability, graceful degradation, and ease of evolution.

- The *persistent data store* is a replicated or partitioned "database" that is spread across the servers' disks. It might also include network-attached storage such as external DBMSs or systems that use RAID storage.
- Many services also use a *backplane*. This optional system-area-network handles inter-server traffic such as redirecting client queries to the correct server or coherence traffic for the persistent data store.

Nearly all services have several other service-specific auxiliary systems such as user-profile databases, ad servers, and site management tools, but we can ignore these in the basic model.

Load Management

Load management has seen rapid improvement since around 1995. The original approach used "round-robin DNS," which distributes different IP addresses for a single domain name among clients in a rotating fashion. Although this approach balances load well, it does not hide inactive servers; a client with a down node's address will continue to try to use it until the DNS mapping expires, which might take several hours. (Short "time-to-live" settings have shorter outages, but require the client to make more DNS lookups.) Many browsers exacerbate the problem by mishandling DNS expiration.

Fortunately, many vendors now sell "layer-4" switches to solve load management (for example, Cisco's CSS 11800 content switch, F5 Networks' Big-IP enterprise switch, and Foundry Networks' ServerIron Web switch). Such transport-layer switches understand TCP and port numbers, and can make decisions based on this information. Many of these are "layer-7" (application layer) switches, in that they understand HTTP requests and can actually parse URLs at wire speed. The switches typically come in pairs that support hot failover — the ability for one switch to take over for another automatically — to avoid the obvious single point of failure, and can handle throughputs

above 20 Gbits per second. They detect down nodes automatically, usually by monitoring open TCP connections, and thus dynamically isolate down nodes from clients quite well.

Two other load-management approaches are typically employed in combination with layer-4 switches. The first uses custom "front-end" nodes that act as service-specific layer-7 routers (in software).[2] Wal-Mart's site uses this approach, for example, because it helps with session management: Unlike switches, the nodes track session information for each user.

The final approach includes clients in the load-management process when possible. This general "smart client" end-to-end approach goes beyond the scope of a layer-4 switch.[3] It greatly simplifies switching among different physical sites, which in turn simplifies disaster tolerance and overload recovery. Although there is no generic way to do this for the Web, it is common with other systems. In DNS, for instance, clients know about an alternative server and can switch to it if the primary disappears; with cell phones this approach is implemented as part of roaming; and application servers in the middle tier of three-tier database systems understand database failover.

Figures 2 and 3 illustrate systems at opposite ends of the complexity spectrum: a simple Web farm and a server similar to the Inktomi search engine cluster. These systems differ in load management, use of a backplane, and persistent data store.

The Web farm in Figure 2 uses round-robin DNS for load management. The persistent data store is implemented by simply replicating all content to all nodes, which works well with a small amount of content. Finally, because all servers can handle all queries, there is no coherence traffic and no need for a backplane. In practice, even simple Web farms often have a second LAN (backplane) to simplify manual updates of the replicas. In this version, node failures reduce system capacity, but not data availability.

In Figure 3, a pair of layer-4 switches manages the load within the site. The "clients" are actually other programs (typically Web servers) that use the smart-client approach to failover among different physical clusters, primarily based on load.

Because the persistent store is partitioned across servers, possibly without replication, node failures could reduce the store's effective size and overall capacity. Furthermore, the nodes are no longer identical, and some queries might need to be directed to specific nodes. This is typically accomplished using a layer-7 switch to parse URLs, but some systems, such as clustered Web caches, might also use the backplane to route requests to the correct node.[4]

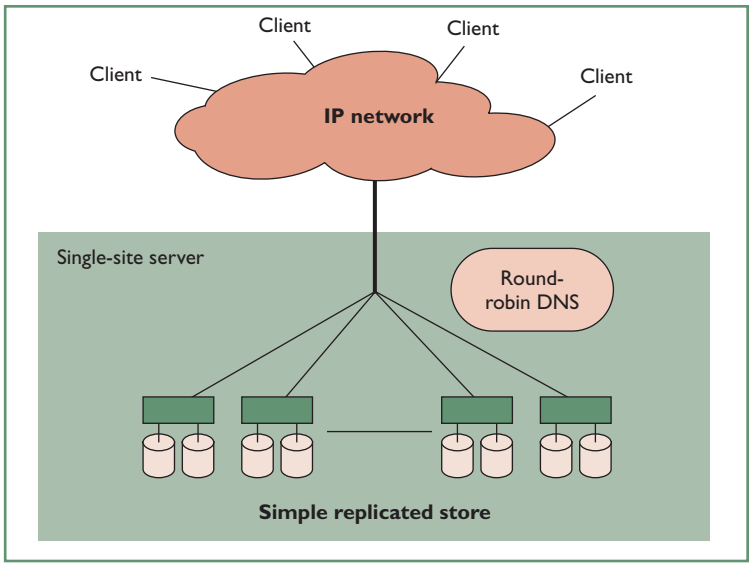


Figure 2. A simple Web farm. Round-robin DNS assigns different servers to different clients to achieve simple load balancing. Persistent data is fully replicated and thus all nodes are identical and can handle all queries.

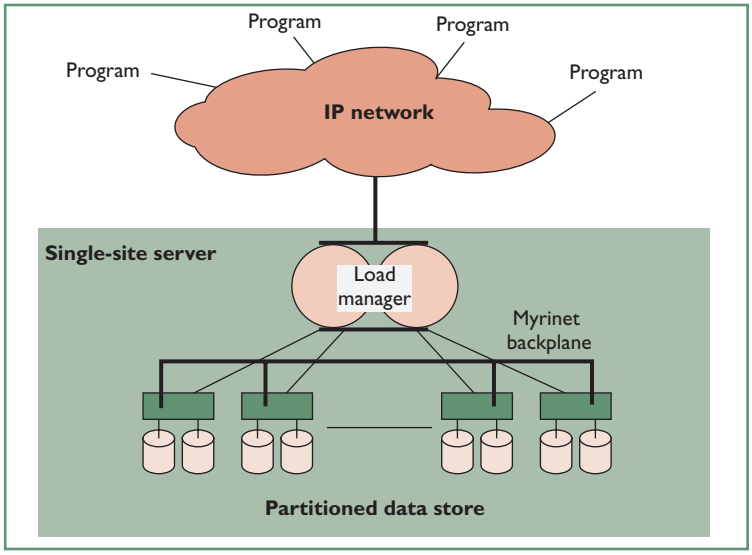


Figure 3. Search engine cluster. The service provides support to other programs (Web servers) rather than directly to end users. These programs connect via layer-4 switches that balance load and hide faults. Persistent data is partitioned across the servers, which increases aggregate capacity but implies there is some data loss when a server is down. A backplane allows all nodes to access all data.

High Availability

High availability is a major driving requirement behind giant-scale system design. Other infra-

Figure 4. 100-node 200-CPU cluster. Key design points include extreme symmetry, internal disks, no people, no monitors, and no visible cables.

structures — such as the telephone, rail, and water systems — aim for perfect availability, a goal that should apply to IP-based infrastructure services as well. All these systems plan for component failures and natural disasters, but information systems must also deal with constantly evolving features and unpredictable growth.

Figure 4 shows a cluster designed for high availability: It features extreme symmetry, no people, very few cables, almost no external disks, and no monitors. Each of these design features reduces the number of failures in practice. In addition, Inktomi manages the cluster from offsite, and contracts limit temperature and power variations.

Availability Metrics

The traditional metric for availability is *uptime*, which is the fraction of time a site is handling traffic. Uptime is typically measured in nines, and traditional infrastructure systems such as the phone system aim for four or five nines ("four nines" implies 0.9999 uptime, or less than 60 seconds of downtime per week). Two related metrics are mean-time-between-failure (MTBF) and mean-time-to-repair (MTTR). We can think of uptime as:

$$\text{uptime} = (\text{MTBF} - \text{MTTR})/\text{MTBF} \quad (1)$$

Following this equation, we can improve uptime either by reducing the frequency of failures or reducing the time to fix them. Although the former is more pleasing aesthetically, the latter is much easier to accomplish with evolving systems. For example, to see if a component has an MTBF of one week requires well more than a week of testing under heavy realistic load. If the component fails, you have to start over, possibly repeating the process many times. Conversely, measuring the MTTR takes minutes or less and achieving a 10-percent improvement takes orders of magnitude less total time because of the very fast debugging cycle. In addition, new features tend to reduce MTBF but have relatively little impact on MTTR, which makes it more stable. Thus, giant-scale systems should focus on improving MTTR and simply apply best effort to MTBF.

We define *yield* as the fraction of queries that are completed:

$$\text{yield} = \text{queries completed}/\text{queries offered} \quad (2)$$

Numerically, this is typically very close to uptime, but it is more useful in practice because it directly maps to user experience and because it correctly reflects that not all seconds have equal value. Being down for a second when there are no queries has no impact on users or yield, but reduces uptime. Similarly, being down for one second at peak and off-peak times generates the same uptime, but vastly different yields because there might be an order-of-magnitude difference in load between the peak second and the minimum-load second. Thus we focus on yield rather than uptime.

Because these systems are typically based on queries, we can also measure query completeness — how much of the database is reflected in the answer. We define this fraction as the *harvest* of the query:

$$\text{harvest} = \text{data available}/\text{complete data} \quad (3)$$

A perfect system would have 100 percent yield and 100 percent harvest. That is, every query would complete and would reflect the entire database. Similarly, we can extend this idea to the fraction of features available; systems such as eBay can have some features, such as seller profiles, unavailable while the rest of the site works perfectly.

The key insight is that we can influence whether faults impact yield, harvest, or both. Replicated systems tend to map faults to reduced capacity (and to yield at high utilizations), while partitioned systems tend to map faults to reduced harvest, as parts of the database temporarily disappear, but the capacity in queries per second remains the same.

DQ Principle

The DQ Principle is simple:

$$\text{Data per query} \times \text{queries per second} \rightarrow \text{constant} \quad (4)$$

This is a principle rather than a literal truth, but it is remarkably useful for thinking about giant-scale systems. The intuition behind this principle is that the system's overall capacity tends to have a particular physical bottleneck, such as total I/O bandwidth or total seeks per second, which is tied to data movement. The DQ value is the total amount of data that has to be moved per second on average, and it is thus bounded by the underlying physical limitation. At the high utilization level typical of giant-scale systems, the DQ value approaches this limitation.

Various systems have used versions of this principle for a long time. For example, the query optimizer in System R used the number of I/O operations as the metric of optimization.[5] The DQ principle focuses more on bandwidth than on the number of I/Os (or seeks) — mainly because I have found that it correlates better with capacity and seems easier to affect gracefully. On a more intuitive level, service work corresponds to the amount of data moved because of data copying, presentation layers, and the many independent connections that tend to make these systems more network-bound than disk-bound. Also, the workload distribution tends to be fairly stable overall because of the scale (we are aggregating over thousands of connections), which means the ratio of bandwidth to I/Os is fairly consistent. Thus, either would be a good metric for most systems.

The DQ value is measurable and tunable. Adding nodes or implementing software optimizations increases the DQ value, while the presence of faults reduces it. The absolute value is not typically that important, but the relative value under various changes is very predictive. Thus, the first step is to define the specific DQ metric for the service by defining the target workload and using a load generator to measure a given combination of hardware, software, and database size against this workload. Every node has a DQ capacity that depends on this combination. Different nodes might have different values if they were bought separately; knowing the relative DQ values tells you how unevenly to do partitioning or load balancing.

Given the metric and load generator, it is then easy to measure the (relative) DQ impact of faults, database sizes, and hardware and software upgrades. Overall, DQ normally scales linearly with the number of nodes, which means a small test cluster is a good predictor for DQ changes on the production system. For example, Inktomi has been able to use four-node clusters to predict performance improvements on 100-node clusters due to software optimizations. The DQ impact can and should be evaluated for all proposed hardware and software changes prior to deployment. Similarly, the best case under multiple node faults is a linear reduction in overall DQ.

We can translate future traffic predictions into future DQ requirements and thus into hardware and software targets. This lets us convert traffic predictions into capacity planning decisions, taking into account any expected hardware or software improvements.

In terms of availability, these principles are especially valuable for analyzing fault impact. As I mentioned, the best we can do is to degrade DQ linearly with the number of (node) faults. The design goal for high availability is thus to control how DQ reductions affect our three availability metrics. The obvious limit of this principle is that it is for data-intensive sites. Computation-bound sites (such as simulation engines) or sites with long-latency external actions (such as chat sites) behave differently. The vast majority of the top 100 sites are data intensive, however, which is not surprising given that the Internet is primarily a communication medium.

Absolute value is not typically that important, but relative value is very predictive.

Replication vs. Partitioning

Replication is a traditional technique for increasing availability, and I will compare it to partitioning from the perspective of DQ and our availability metrics. Consider a two-node cluster: The replicated version has traditionally been viewed as "better" because it maintains 100 percent harvest under a fault, whereas the partitioned version drops to 50 percent harvest. Dual analysis shows that the replicated version drops to 50 percent yield, however, while the partitioned version remains at 100 percent yield. Furthermore, both versions have the same initial DQ value and lose 50 percent of it under one fault: Replicas maintain D and reduce Q (and thus yield), while partitions keep Q constant and reduce D (and thus harvest).

Table 1. Overload due to failures.

Failures	Lost capacity	Redirected load	Overload factor
1	$\frac{1}{n}$	$\frac{1}{n-1}$	$\frac{n}{n-1}$
k	$\frac{k}{n}$	$\frac{k}{n-k}$	$\frac{n}{n-k}$

The traditional view of replication silently assumes that there is enough excess capacity to prevent faults from affecting yield. We refer to this as the *load redirection problem* because under faults, the remaining replicas have to handle the queries formerly handled by the failed nodes. Under high utilization, this is unrealistic.

Table 1 generalizes this analysis to replica groups with n nodes. Losing two of five nodes in a replica group, for example, implies a redirected load of 2/3 extra load (two loads spread over three remaining nodes) and an overload factor for those nodes of 5/3 or 166 percent of normal load.

The key insight is that replication on disk is cheap, but accessing the replicated data requires DQ points. For true replication you need not only another copy of the data, but also twice the DQ value. Conversely, partitioning has no real savings over replication. Although you need more copies of the data with replication, the real cost is in the DQ bottleneck rather than storage space, and the DQ constant is independent of whether the database is replicated or partitioned. An important exception to this is that replication requires more DQ points than partitioning for heavy write traffic, which is rare in giant-scale systems.

According to these principles, you should always use replicas above some specified throughput. In theory, you can always partition the database more thinly and avoid the extra replicas, but with no DQ difference, it makes more sense to replicate the data once the partitions are a convenient size. You will enjoy more control over harvest and support for disaster recovery, and it is easier to grow systems via replication than by repartitioning onto more nodes.

We can vary the replication according to the data's importance, and generally control which data is lost in the presence of a fault. For example, for the cost of some extra disk space we can replicate key data in a partitioned system. Under normal use, one node handles the key data and the rest provide additional partitions. If the main node fails, we can make one of the other nodes serve the key data. We still lose $1/n$ of the data, but it will always be one of the less important partitions. This combined approach preserves the key data, but also allows us to use our "replicated" DQ capacity to serve other content during normal operation.

Finally, we can exploit randomization to make our lost harvest a random subset of the data, (as well as to avoid hot spots in partitions). Many of the load-balancing switches can use a (pseudo-random) hash function to partition the data, for example. This makes the average and worst-case losses the same because we lose a random subset of "average" value. The Inktomi search engine uses partial replication; e-mail systems use full replication; and clustered Web caches use no replication. All three use randomization.

Graceful Degradation

It would be nice to believe that we could avoid saturation at a reasonable cost simply by good design, but this is unrealistic for three major reasons.

- The peak-to-average ratio for giant-scale systems seems to be in the range of 1.6:1 to 6:1, which can make it expensive to build out capacity well above the (normal) peak.
- Single-event bursts, such as online ticket sales for special events, can generate far above-average traffic. In fact, moviephone.com added 10x capacity to handle tickets for "Star Wars: The Phantom Menace" and still got overloaded.
- Some faults, such as power failures or natural disasters, are not independent. Overall DQ drops substantially in these cases, and the remaining nodes become saturated.

Given these facts, mechanisms for graceful degradation under excess load are critical for delivering high availability. The DQ principle gives us new options for graceful degradation: We can either limit Q (capacity) to maintain D, or we can reduce D and increase Q. We can focus on harvest through admission control (AC), which reduces Q, or on yield through dynamic database reduction, which reduces D, or we can use a combination of the two. Temporarily cutting the effective database size in half, for instance, should roughly double our capacity. Some giant-scale services have just started applying the latter strategy.

The larger insight is that graceful degradation is simply the explicit process for managing the effect of saturation on availability; that is, we explicitly decide how saturation should affect uptime, harvest, and quality of service. Here are

some more sophisticated examples taken from real systems:

- *Cost-based AC*. Inktomi can perform AC based on the estimated query cost (measured in DQ). This reduces the average data required per query, and thus increases Q. Note that the AC policy affects both D and Q. Denying one expensive query could thus enable several inexpensive ones, giving us a net gain in harvest and yield. (Although I know of no examples, AC could also be done probabilistically — perhaps in the style of lottery scheduling, so that retrying hard queries eventually works.)
- *Priority- or value-based AC*. Datek handles stock trade requests differently from other queries and guarantees that they will be executed within 60 seconds, or the user pays no commission. The idea is to reduce the required DQ by dropping low-value queries, independently of their DQ cost.
- *Reduced data freshness*. Under saturation, a financial site can make stock quotes expire less frequently. This reduces freshness but also reduces the work per query, and thus increases yield at the expense of harvest. (The cached queries don't reflect the current database and thus have lower harvest.)

As you can see, the DQ principle can be used as a tool for designing how saturation affects our availability metrics. We first decide which metrics to preserve (or at least focus on), and then we use sophisticated AC to limit Q and possibly reduce the average D. We also use aggressive caching and database reduction to reduce D and thus increase capacity.

Disaster Tolerance

Disaster tolerance is a simple combination of managing replica groups and graceful degradation. A disaster can be defined as the complete loss of one or more replicas. Natural disasters can affect all the replicas at one physical location, while other disasters such as fires or power failures might affect only one replica.

Under the giant-scale services model, the basic question is how many locations to establish and how many replicas to put at each. To examine the load redirection problem, I return to Table 1. With two replicas at each of three locations, for example, we expect to lose 2/6 of the replicas during a natural disaster, which implies that each of the remaining sites must handle 50 percent more traffic. This will almost certainly saturate the site, which will employ graceful-degradation techniques to recover. For Inktomi, the best plan would be to dynamically reduce D by 2/3 (to get 3/2 Q) on the remaining replicas. The current plan actually reduces D to 50 percent for any disaster, which is simpler, but not as aggressive.

Load management presents a harder problem for disaster recovery. Layer-4 switches do not help with the loss of whole clusters, so the external site name might become unavailable during a disaster. In that case, you must resort to DNS changes or smart clients to redirect traffic to the other replicas. As I mentioned, however, DNS might have a very slow failover response time of up to several hours. With a smart-client approach, clients can perform the higher-level redirection automatically and immediately, which seems the most compelling reason to use smart clients.

Online Evolution and Growth

One of the traditional tenets of highly available systems is minimal change. This directly conflicts with both the growth rates of giant-scale services and "Internet time" — the practice of frequent product releases. For these services, we must plan for continuous growth and frequent functionality updates. Worse still, the frequency of updates means that in practice, software is never perfect; hard-to-resolve issues such as slow memory leaks and nondeterministic bugs tend to remain unfixed. The prevailing design philosophy aims to make the overall system tolerant of individual node failures and to avoid cascading failures. Thus, "acceptable" quality comes down to software that provides a target MTBF, a low MTTR, and no cascading failures.

We can think of maintenance and upgrades as controlled failures, and the "online evolution" process — completing upgrades without taking down the site — is important because giant-scale services are updated so frequently. By viewing online evolution as a temporary, controlled reduction in DQ value, we can act to minimize the impact on our availability metrics. In general, online evolution requires a fixed amount of time per node, u, so that the total DQ loss for n nodes is:

$$\Delta DQ = n \cdot u \cdot \text{average DQ/node} = DQ \cdot u \qquad (5)$$

That is, the lost DQ value is simply the total DQ value multiplied by the upgrade time per node.

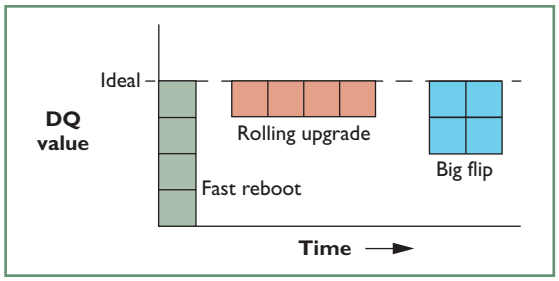


Figure 5. Three approaches to upgrades. The shaded regions map DQ loss (down from the ideal value) over time for a four-node cluster. Note that the area of the three regions is the same.

Upgrade time is essentially recovery time, and depends on the kind of change. Software upgrades are most common and can be handled quickly with a staging area, which allows both the new and old versions to coexist on the node. System administrators can do a controlled reboot (at worst) to upgrade the node within the normal MTTR. Without sufficient workspace, the downtime is much greater because the administrator must move the new version onto the node while the system is down. Staging also simplifies returning to the old version, which is helpful because live traffic tends to expose new bugs. Other upgrades, such as hardware, operating system, database schema, or repartitioning, require significantly more downtime.

Three main approaches are used for online evolution:

- *Fast reboot*. The simplest approach is to quickly reboot all nodes into the new version. The fast reboot is straightforward, but it guarantees some downtime. Thus, it is more useful to measure lost yield rather than downtime. By upgrading during off-peak hours, we can reduce the yield impact. In practice, this requires a staging area because the upgrades all occur at the same time and thus need to be highly automated.
- *Rolling upgrade*. In this approach, we upgrade nodes one at a time in a "wave" that rolls through the cluster. This minimizes the overall impact because only one node is down at a time. In a partitioned system, harvest will be reduced during the *n* upgrade windows, but in a replicated system we expect 100 percent yield and 100 percent harvest because we can update one replica at a time, and we probably have enough capacity (in off hours) to prevent lost yield. One disadvantage with the rolling upgrade is that the old and new versions must be compatible because they will coexist. Examples of incompatible versions include changes to schema, namespaces, or (intracluster) protocols. Neither of the other two approaches has this restriction.
- *Big flip*. The final approach is the most complicated. The basic idea is to update the cluster one half at a time by taking down and upgrading half the nodes at once. During the "flip," we atomically switch all traffic to the upgraded half using a layer-4 switch. After waiting for connections to the old half to dissipate (the switch only affects new connections), we then upgrade the second half and bring those nodes back into use. As with fast reboot, only one version runs at a time. Note that the 50 percent DQ loss can be translated into either 50 percent capacity for replicas (which might be 100 percent off-peak yield) or 50 percent harvest for partitions.

The big flip is quite powerful and can be used for all kinds of upgrades, including hardware, OS, schema, networking, and even physical relocation. When used for relocating a cluster, the big flip must typically use DNS changes, and the upgrade time includes physically moving the nodes, which means the window of 50 percent DQ loss might last several hours. This is manageable over a weekend, however, and Inktomi has done it twice: first, when moving about 20 nodes from Berkeley to Santa Clara, California, (about an hour apart) using DNS for the flip; and second, when moving about 100 nodes between two cages in the same data center using a switch for the flip.

All three approaches have the same DQ loss (for a given upgrade), so we can plot their effective DQ level versus time, which is shown in Figure 5. The three regions have the same area, but the DQ loss is spread over time differently. All three approaches are used in practice, but rolling upgrades are easily the most popular. The heavyweight big flip is reserved for more complex changes and is used only rarely. The approaches differ in their handling of the DQ loss and in how they deal with staging areas and version compatibility, but all three benefit from DQ analysis and explicit management of the impact on availability metrics.

Conclusions

From my experience with several giant-scale services, I have defined several tools for designing and analyzing giant-scale services. Starting with the basic architectural model leads to novel ways

to think about high availability, including the DQ principle and the use of harvest and yield. Experience has helped us develop several ways to control the impact of faults through combinations of replication and partitioning. These tools are useful for graceful degradation, disaster tolerance, and online evolution.

I've learned several major lessons for developing giant-scale services:

- *Get the basics right*. Start with a professional data center and layer-7 switches, and use symmetry to simplify analysis and management.
- *Decide on your availability metrics*. Everyone should agree on the goals and how to measure them daily. Remember that harvest and yield are more useful than just uptime.
- *Focus on MTTR at least as much as MTBF*. Repair time is easier to affect for an evolving system and has just as much impact.
- *Understand load redirection during faults*. Data replication is insufficient for preserving uptime under faults; you also need excess DQ.
- *Graceful degradation is a critical part of a high-availability strategy*. Intelligent admission control and dynamic database reduction are the key tools for implementing the strategy.
- *Use DQ analysis on all upgrades*. Evaluate all proposed upgrades ahead of time, and do capacity planning.
- *Automate upgrades as much as possible*. Develop a mostly automatic upgrade method, such as rolling upgrades. Using a staging area will reduce downtime, but be sure to have a fast, simple way to revert to the old version.

These techniques seem to get to the core issues in practice and provide ways to think about availability and fault tolerance that are predictable, analyzable, and measurable.

Current solutions are essentially ad hoc applications of these ideas. Techniques for online evolution and graceful degradation, for example, could be greatly automated. Similarly, future systems could provide more direct ways to integrate simple application information, such as a query's cost or value, into availability and load management mechanisms. Finally, better use of smart clients could simplify disaster recovery, online evolution, and graceful degradation.

References

1. A. Wolman et al., "On the Scale and Performance of Cooperative Web Proxy Caching," *Proc. 17th Symp. Operating Systems Principles 1999*, ACM Press, New York, 1999, pp. 16-31.
2. A. Fox et al., "Cluster-Based Scalable Network Services," *Proc. 16th Symp. Operating System Principles*, ACM Press, New York, 1997, pp. 78-91.
3. C. Yoshikawa et al., "Using Smart Clients to Build Scalable Services," *Proc. Usenix Annual Technical Conf.*, Usenix Assoc., Berkeley, Calif., Jan. 1997.
4. Traffic Server Technical Specification, Inktomi Corporation, Foster City, Calif., 2001; http://www.inktomi.com/products/cns/products/tscclass.html.
5. M.M. Astrahan et al., "System R: A Relational Approach to Database Management," *ACM Trans. Database Systems*, ACM Press, New York, June 1976, p. 97.

Eric A. Brewer is founder and Chief Scientist of Inktomi and an associate professor of computer science at UC Berkeley. He received a PhD in computer science from MIT. His research interests include mobile and wireless computing, scalable servers, and application- and system-level security. Brewer founded the Federal Search Foundation in 2000, which helped create the first U.S. government-wide portal, FirstGov.gov, for which he won the 2001 Federal CIO Council's Azimuth Award.

Readers can contact the author at brewer@cs.berkeley.edu.

For further information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib/.

Coming in 2002 ...

- **Peer-to-Peer Networking**
- **Usability and the World Wide Web**
- **Internet Telephony**

IEEE Internet Computing

Calls for papers are available at http://computer.org/call4ppr.htm

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica*, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan†
MIT Laboratory for Computer Science
chord@lcs.mit.edu
http://pdos.lcs.mit.edu/chord/

Abstract

A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. This paper presents *Chord*, a distributed lookup protocol that addresses this problem. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data item pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Results from theoretical analysis, simulations, and experiments show that Chord is scalable, with communication cost and the state maintained by each node scaling logarithmically with the number of Chord nodes.

1. Introduction

Peer-to-peer systems and applications are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality. A review of the features of recent peer-to-peer applications yields a long list: redundant storage, permanence, selection of nearby servers, anonymity, search, authentication, and hierarchical naming. Despite this rich set of features, the core operation in most peer-to-peer systems is efficient location of data items. The contribution of this paper is a scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures.

The *Chord protocol* supports just one operation: given a key, it maps the key onto a node. Depending on the application using Chord, that node might be responsible for storing a value associated with the key. Chord uses a variant of consistent hashing [11] to assign keys to Chord nodes. Consistent hashing tends to balance load, since each node receives roughly the same number of keys, and involves relatively little movement of keys when nodes join and leave the system.

Previous work on consistent hashing assumed that nodes were aware of most other nodes in the system, making it impractical to scale to large number of nodes. In contrast, each Chord node needs "routing" information about only a few other nodes. Because the routing table is distributed, a node resolves the hash function by communicating with a few other nodes. In the steady state, in an N-node system, each node maintains information only about $O(\log N)$ other nodes, and resolves all lookups via $O(\log N)$ messages to other nodes. Chord maintains its routing information as nodes join and leave the system; with high probability each such event results in no more than $O(\log^2 N)$ messages.

Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance. Chord is simple, routing a key through a sequence of $O(\log N)$ other nodes toward the destination. A Chord node requires information about $O(\log N)$ other nodes for *efficient* routing, but performance degrades gracefully when that information is out of date. This is important in practice because nodes will join and leave arbitrarily, and consistency of even $O(\log N)$ state may be hard to maintain. Only one piece information per node need be correct in order for Chord to guarantee correct (though slow) routing of queries; Chord has a simple algorithm for maintaining this information in a dynamic environment.

The rest of this paper is structured as follows. Section 2 compares Chord to related work. Section 3 presents the system model that motivates the Chord protocol. Section 4 presents the base Chord protocol and proves several of its properties, while Section 5 presents extensions to handle concurrent joins and failures. Section 6 demonstrates our claims about Chord's performance through simulation and experiments on a deployed prototype. Finally, we outline items for future work in Section 7 and summarize our contributions in Section 8.

2. Related Work

While Chord maps keys onto nodes, traditional name and location services provide a *direct* mapping between keys and values. A value can be an address, a document, or an arbitrary data item. Chord can easily implement this functionality by storing each key/value pair at the node to which that key maps. For this reason and to make the comparison clearer, the rest of this section assumes a Chord-based service that maps keys onto values.

DNS provides a host name to IP address mapping [15]. Chord can provide the same service with the name representing the key and the associated IP address representing the value. Chord requires no special servers, while DNS relies on a set of special root

*University of California, Berkeley. istoica@cs.berkeley.edu
†Authors in reverse alphabetical order.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'01, August 27-31, 2001, San Diego, California, USA.
Copyright 2001 ACM 1-58113-411-8/01/0008 ...$5.00.

servers. DNS names are structured to reflect administrative boundaries; Chord imposes no naming structure. DNS is specialized to the task of finding named hosts or services, while Chord can also be used to find data objects that are not tied to particular machines.

The Freenet peer-to-peer storage system [4, 5], like Chord, is decentralized and symmetric and automatically adapts when hosts leave and join. Freenet does not assign responsibility for documents to specific servers; instead, its lookups take the form of searches for cached copies. This allows Freenet to provide a degree of anonymity, but prevents it from guaranteeing retrieval of existing documents or from providing low bounds on retrieval costs. Chord does not provide anonymity, but its lookup operation runs in predictable time and always results in success or definitive failure.

The Ohaha system uses a consistent hashing-like algorithm for mapping documents to nodes, and Freenet-style query routing [18]. As a result, it shares some of the weaknesses of Freenet. Archival Intermemory uses an off-line computed tree to map logical addresses to machines that store the data [3].

The Globe system [2] has a wide-area location service to map object identifiers to the locations of moving objects. Globe arranges the Internet as a hierarchy of geographical, topological, or administrative domains, effectively constructing a static world-wide search tree, much like DNS. Information about an object is stored in a particular leaf domain, and pointer caches provide search short cuts [22]. The Globe system handles high load on the logical root by partitioning objects among multiple physical root servers using hash-like techniques. Chord performs this hash function well enough that it can achieve scalability without also involving any hierarchy, though Chord does not exploit network locality as well as Globe.

The distributed data location protocol developed by Plaxton *et al.* [19], a variant of which is used in OceanStore [12], is perhaps the closest algorithm to the Chord protocol. It provides stronger guarantees than Chord: like Chord it guarantees that queries make a logarithmic number hops and that keys are well balanced, but the Plaxton protocol also ensures, subject to assumptions about network topology, that queries never travel further in network distance than the node where the key is stored. The advantage of Chord is that it is substantially less complicated and handles concurrent node joins and failures well. The Chord protocol is also similar to Pastry, the location algorithm used in PAST [8]. However, Pastry is a prefix-based routing protocol, and differs in other details from Chord.

CAN uses a d-dimensional Cartesian coordinate space (for some fixed d) to implement a distributed hash table that maps keys onto values [20]. Each node maintains $O(d)$ state, and the lookup cost is $O(dN^{1/d})$. Thus, in contrast to Chord, the state maintained by a CAN node does not depend on the network size N, but the lookup cost increases faster than $\log N$. If $d = \log N$, CAN lookup times and storage needs match Chord's. However, CAN is not designed to vary d as N (and thus $\log N$) varies, so this match will only occur for the "right" N corresponding to the fixed d. CAN requires an additional maintenance protocol to periodically remap the identifier space onto nodes. Chord also has the advantage that its correctness is robust in the face of partially incorrect routing information.

Chord's routing procedure may be thought of as a one-dimensional analogue of the Grid location system [14]. Grid relies on real-world geographic location information to route its queries; Chord maps its nodes to an artificial one-dimensional space within which routing is carried out by an algorithm similar to Grid's.

Chord can be used as a lookup service to implement a variety of systems, as discussed in Section 3. In particular, it can help avoid single points of failure or control that systems like Napster possess [17], and the lack of scalability that systems like Gnutella display because of their widespread use of broadcasts [10].

3. System Model

Chord simplifies the design of peer-to-peer systems and applications based on it by addressing these difficult problems:

- **Load balance:** Chord acts as a distributed hash function, spreading keys evenly over the nodes; this provides a degree of natural load balance.
- **Decentralization:** Chord is fully distributed: no node is more important than any other. This improves robustness and makes Chord appropriate for loosely-organized peer-to-peer applications.
- **Scalability:** The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible. No parameter tuning is required to achieve this scaling.
- **Availability:** Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in a continuous state of change.
- **Flexible naming:** Chord places no constraints on the structure of the keys it looks up: the Chord key-space is flat. This gives applications a large amount of flexibility in how they map their own names to Chord keys.

The Chord software takes the form of a library to be linked with the client and server applications that use it. The application interacts with Chord in two main ways. First, Chord provides a `lookup(key)` algorithm that yields the IP address of the node responsible for the key. Second, the Chord software on each node notifies the application of changes in the set of keys that the node is responsible for. This allows the application software to, for example, move corresponding values to their new homes when a new node joins.

The application using Chord is responsible for providing any desired authentication, caching, replication, and user-friendly naming of data. Chord's flat key space eases the implementation of these features. For example, an application could authenticate data by storing it under a Chord key derived from a cryptographic hash of the data. Similarly, an application could replicate data by storing it under two distinct Chord keys derived from the data's application-level identifier.

The following are examples of applications for which Chord would provide a good foundation:

Cooperative Mirroring, as outlined in a recent proposal [6]. Imagine a set of software developers, each of whom wishes to publish a distribution. Demand for each distribution might vary dramatically, from very popular just after a new release to relatively unpopular between releases. An efficient approach for this would be for the developers to cooperatively mirror each others' distributions. Ideally, the mirroring system would balance the load across all servers, replicate and cache the data, and ensure authenticity. Such a system should be fully decentralized in the interests of reliability, and because there is no natural central administration.

Time-Shared Storage for nodes with intermittent connectivity. If a person wishes some data to be always available, but their

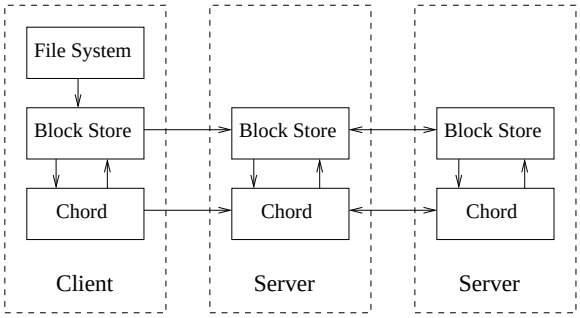


Figure 1: Structure of an example Chord-based distributed storage system.

machine is only occasionally available, they can offer to store others' data while they are up, in return for having their data stored elsewhere when they are down. The data's name can serve as a key to identify the (live) Chord node responsible for storing the data item at any given time. Many of the same issues arise as in the Cooperative Mirroring application, though the focus here is on availability rather than load balance.

Distributed Indexes to support Gnutella- or Napster-like keyword search. A key in this application could be derived from the desired keywords, while values could be lists of machines offering documents with those keywords.

Large-Scale Combinatorial Search, such as code breaking. In this case keys are candidate solutions to the problem (such as cryptographic keys); Chord maps these keys to the machines responsible for testing them as solutions.

Figure 1 shows a possible three-layered software structure for a cooperative mirror system. The highest layer would provide a file-like interface to users, including user-friendly naming and authentication. This "file system" layer might implement named directories and files, mapping operations on them to lower-level block operations. The next layer, a "block storage" layer, would implement the block operations. It would take care of storage, caching, and replication of blocks. The block storage layer would use Chord to identify the node responsible for storing a block, and then talk to the block storage server on that node to read or write the block.

4. The Base Chord Protocol

The Chord protocol specifies how to find the locations of keys, how new nodes join the system, and how to recover from the failure (or planned departure) of existing nodes. This section describes a simplified version of the protocol that does not handle concurrent joins or failures. Section 5 describes enhancements to the base protocol to handle concurrent joins and failures.

4.1 Overview

At its heart, Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them. It uses *consistent hashing* [11, 13], which has several good properties. With high probability the hash function balances load (all nodes receive roughly the same number of keys). Also with high probability, when an Nth node joins (or leaves) the network, only an $O(1/N)$ fraction of the keys are moved to a different location—this is clearly the minimum necessary to maintain a balanced load.

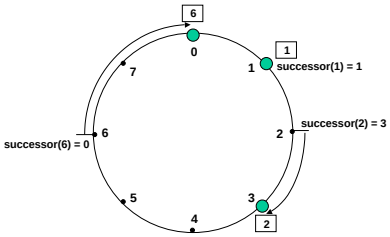


Figure 2: An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0.

Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node. A Chord node needs only a small amount of "routing" information about other nodes. Because this information is distributed, a node resolves the hash function by communicating with a few other nodes. In an N-node network, each node maintains information only about $O(\log N)$ other nodes, and a lookup requires $O(\log N)$ messages.

Chord must update the routing information when a node joins or leaves the network; a join or leave requires $O(\log^2 N)$ messages.

4.2 Consistent Hashing

The consistent hash function assigns each node and key an m-bit *identifier* using a base hash function such as SHA-1 [9]. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. We will use the term "key" to refer to both the original key and its image under the hash function, as the meaning will be clear from context. Similarly, the term "node" will refer to both the node and its identifier under the hash function. The identifier length m must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible.

Consistent hashing assigns keys to nodes as follows. Identifiers are ordered in an *identifier circle* modulo 2^m. Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. This node is called the *successor node* of key k, denoted by *successor*(k). If identifiers are represented as a circle of numbers from 0 to $2^m - 1$, then $successor(k)$ is the first node clockwise from k.

Figure 2 shows an identifier circle with $m = 3$. The circle has three nodes: 0, 1, and 3. The successor of identifier 1 is node 1, so key 1 would be located at node 1. Similarly, key 2 would be located at node 3, and key 6 at node 0.

Consistent hashing is designed to let nodes enter and leave the network with minimal disruption. To maintain the consistent hashing mapping when a node n joins the network, certain keys previously assigned to n's successor now become assigned to n. When node n leaves the network, all of its assigned keys are reassigned to n's successor. No other changes in assignment of keys to nodes need occur. In the example above, if a node were to join with identifier 7, it would capture the key with identifier 6 from the node with identifier 0.

The following results are proven in the papers that introduced consistent hashing [11, 13]:

THEOREM 1. *For any set of N nodes and K keys, with high probability:*

1. *Each node is responsible for at most $(1+\epsilon)K/N$ keys*

2. *When an $(N+1)^{st}$ node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands (and only to or from the joining or leaving node).*

When consistent hashing is implemented as described above, the theorem proves a bound of $\epsilon = O(\log N)$. The consistent hashing paper shows that ϵ can be reduced to an arbitrarily small constant by having each node run $O(\log N)$ "virtual nodes" each with its own identifier.

The phrase "with high probability" bears some discussion. A simple interpretation is that the nodes and keys are randomly chosen, which is plausible in a non-adversarial model of the world. The probability distribution is then over random choices of keys and nodes, and says that such a random choice is unlikely to produce an unbalanced distribution. One might worry, however, about an adversary who intentionally chooses keys to all hash to the same identifier, destroying the load balancing property. The consistent hashing paper uses "k-universal hash functions" to provide certain guarantees even in the case of nonrandom keys.

Rather than using a k-universal hash function, we chose to use the standard SHA-1 function as our base hash function. This makes our protocol deterministic, so that the claims of "high probability" no longer make sense. However, producing a set of keys that collide under SHA-1 can be seen, in some sense, as inverting, or "decrypting" the SHA-1 function. This is believed to be hard to do. Thus, instead of stating that our theorems hold with high probability, we can claim that they hold "based on standard hardness assumptions."

For simplicity (primarily of presentation), we dispense with the use of virtual nodes. In this case, the load on a node may exceed the average by (at most) an $O(\log N)$ factor with high probability (or in our case, based on standard hardness assumptions). One reason to avoid virtual nodes is that the number needed is determined by the number of nodes in the system, which may be difficult to determine. Of course, one may choose to use an a priori upper bound on the number of nodes in the system; for example, we could postulate at most one Chord server per IPv4 address. In this case running 32 virtual nodes per physical node would provide good load balance.

4.3 Scalable Key Location

A very small amount of routing information suffices to implement consistent hashing in a distributed environment. Each node need only be aware of its successor node on the circle. Queries for a given identifier can be passed around the circle via these successor pointers until they first encounter a node that succeeds the identifier; this is the node the query maps to. A portion of the Chord protocol maintains these successor pointers, thus ensuring that all lookups are resolved correctly. However, this resolution scheme is inefficient: it may require traversing all N nodes to find the appropriate mapping. To accelerate this process, Chord maintains additional routing information. This additional information is not essential for correctness, which is achieved as long as the successor information is maintained correctly.

As before, let m be the number of bits in the key/node identifiers. Each node, n, maintains a routing table with (at most) m entries, called the *finger table*. The ith entry in the table at node n contains the identity of the *first* node, s, that succeeds n by at least 2^{i-1} on the identifier circle, i.e., $s = successor(n + 2^{i-1})$, where $1 \leq i \leq m$ (and all arithmetic is modulo 2^m). We call node s the ith *finger* of node n, and denote it by $n.finger[i].node$ (see Table 1). A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Note that the first finger of n is its immediate successor on the circle; for convenience we often refer to it as the *successor* rather than the first finger.

In the example shown in Figure 3(b), the finger table of node 1 points to the successor nodes of identifiers $(1 + 2^0) \bmod 2^3 = 2$, $(1 + 2^1) \bmod 2^3 = 3$, and $(1 + 2^2) \bmod 2^3 = 5$, respectively. The successor of identifier 2 is node 3, as this is the first node that follows 2, the successor of identifier 3 is (trivially) node 3, and the successor of 5 is node 0.

Notation	Definition
finger$[k].start$	$(n + 2^{k-1}) \bmod 2^m$, $1 \leq k \leq m$
.interval	[*finger*$[k]$.*start*, *finger*$[k+1]$.*start*)
.node	first node $\geq n.$*finger*$[k]$.*start*
successor	the next node on the identifier circle; *finger*[1].*node*
predecessor	the previous node on the identifier circle

Table 1: Definition of variables for node n, using m-bit identifiers.

This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. Second, a node's finger table generally does not contain enough information to determine the successor of an arbitrary key k. For example, node 3 in Figure 3 does not know the successor of 1, as 1's successor (node 1) does not appear in node 3's finger table.

What happens when a node n does not know the successor of a key k? If n can find a node whose ID is closer than its own to k, that node will know more about the identifier circle in the region of k than n does. Thus n searches its finger table for the node j whose ID most immediately precedes k, and asks j for the node it knows whose ID is closest to k. By repeating this process, n learns about nodes with IDs closer and closer to k.

The pseudocode that implements the search process is shown in Figure 4. The notation *n.foo()* stands for the function *foo()* being invoked at and executed on node n. Remote calls and variable references are preceded by the remote node identifier, while local variable references and procedure calls omit the local node. Thus *n.foo()* denotes a remote procedure call on node n, while *n.bar*, without parentheses, is an RPC to lookup a variable *bar* on node n.

find_successor works by finding the immediate predecessor node of the desired identifier; the successor of that node must be the successor of the identifier. We implement *find_predecessor* explicitly, because it is used later to implement the join operation (Section 4.4).

When node n executes *find_predecessor*, it contacts a series of nodes moving forward around the Chord circle towards id. If node n contacts a node n' such that id falls between n' and the successor of n', *find_predecessor* is done and returns n'. Otherwise node n asks n' for the node n' knows about that most closely precedes id. Thus the algorithm always makes progress towards the precedessor of id.

As an example, consider the Chord ring in Figure 3(b). Suppose node 3 wants to find the successor of identifier 1. Since 1 belongs to the circular interval [7, 3), it belongs to 3.*finger*[3].*interval*; node 3 therefore checks the third entry in its finger table, which is 0. Because 0 precedes 1, node 3 will ask node 0 to find the successor of 1. In turn, node 0 will infer from its finger table that 1's successor is the node 1 itself, and return node 1 to node 3.

The finger pointers at repeatedly doubling distances around the circle cause each iteration of the loop in *find_predecessor* to halve the distance to the target identifier. From this intuition follows a theorem:

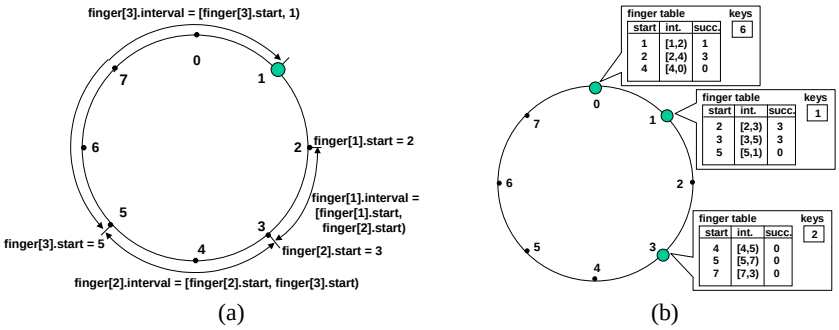


Figure 3: (a) The finger intervals associated with node 1. (b) Finger tables and key locations for a net with nodes 0, 1, and 3, and keys 1, 2, and 6.

THEOREM 2. *With high probability (or under standard hardness assumptions), the number of nodes that must be contacted to find a successor in an N-node network is $O(\log N)$.*

PROOF. Suppose that node n wishes to resolve a query for the successor of k. Let p be the node that immediately precedes k. We analyze the number of query steps to reach p.

Recall that if $n \neq p$, then n forwards its query to the closest predecessor of k in its finger table. Suppose that node p is in the ith finger interval of node n. Then since this interval is not empty, node n will finger some node f in this interval. The distance (number of identifiers) between n and f is at least 2^{i-1}. But f and p are both in n's i^{th} finger interval, which means the distance between them is at most 2^{i-1}. This means f is closer to p than to n, or equivalently, that the distance from f to p is at most half the distance from n to p.

If the distance between the node handling the query and the predecessor p halves in each step, and is at most 2^m initially, then within m steps the distance will be one, meaning we have arrived at p.

In fact, as discussed above, we assume that node and key identifiers are random. In this case, the number of forwardings necessary will be $O(\log N)$ with high probability. After $\log N$ forwardings, the distance between the current query node and the key k will be reduced to at most $2^m/N$. The expected number of node identifiers landing in a range of this size is 1, and it is $O(\log N)$ with high probability. Thus, even if the remaining steps advance by only one node at a time, they will cross the entire remaining interval and reach key k within another $O(\log N)$ steps. □

In the section reporting our experimental results (Section 6), we will observe (and justify) that the average lookup time is $\frac{1}{2}\log N$.

4.4 Node Joins

In a dynamic network, nodes can join (and leave) at any time. The main challenge in implementing these operations is preserving the ability to locate every key in the network. To achieve this goal, Chord needs to preserve two invariants:

1. Each node's successor is correctly maintained.
2. For every key k, node $successor(k)$ is responsible for k.

In order for lookups to be fast, it is also desirable for the finger tables to be correct.

This section shows how to maintain these invariants when a single node joins. We defer the discussion of multiple nodes joining simultaneously to Section 5, which also discusses how to handle

```
// ask node n to find id's successor
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor;

// ask node n to find id's predecessor
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor])
    n' = n'.closest_preceding_finger(id);
  return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
  for i = m downto 1
    if (finger[i].node ∈ (n, id))
      return finger[i].node;
  return n;
```

Figure 4: The pseudocode to find the successor node of an identifier id. Remote procedure calls and variable lookups are preceded by the remote node.

a node failure. Before describing the join operation, we summarize its performance (the proof of this theorem is in the companion technical report [21]):

THEOREM 3. *With high probability, any node joining or leaving an N-node Chord network will use $O(\log^2 N)$ messages to re-establish the Chord routing invariants and finger tables.*

To simplify the join and leave mechanisms, each node in Chord maintains a *predecessor pointer*. A node's predecessor pointer contains the Chord identifier and IP address of the immediate predecessor of that node, and can be used to walk counterclockwise around the identifier circle.

To preserve the invariants stated above, Chord must perform three tasks when a node n joins the network:

1. Initialize the predecessor and fingers of node n.
2. Update the fingers and predecessors of existing nodes to reflect the addition of n.
3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node n is now responsible for.

We assume that the new node learns the identity of an existing Chord node n' by some external mechanism. Node n uses n' to

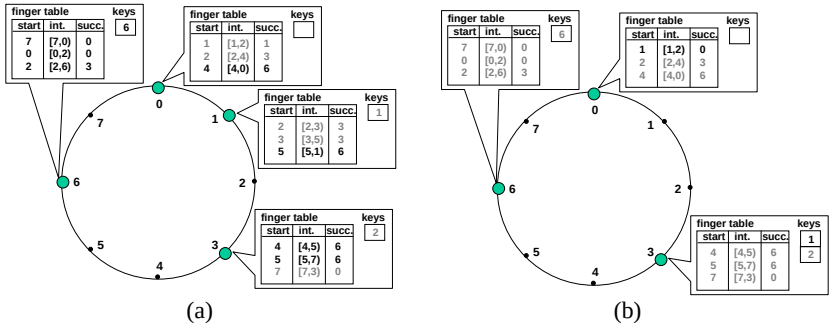


Figure 5: (a) Finger tables and key locations after node 6 joins. (b) Finger tables and key locations after node 3 leaves. Changed entries are shown in black, and unchanged in gray.

initialize its state and add itself to the existing Chord network, as follows.

Initializing fingers and predecessor: Node n learns its predecessor and fingers by asking n' to look them up, using the *init_finger_table* pseudocode in Figure 6. Naively performing *find_successor* for each of the m finger entries would give a run-time of $O(m \log N)$. To reduce this, n checks whether the ith finger is also the correct $(i+1)^{th}$ finger, for each i. This happens when *finger*$[i]$.*interval* does not contain any node, and thus *finger*$[i]$.*node* $\geq$ *finger*$[i+1]$.*start*. It can be shown that the change reduces the expected (and high probability) number of finger entries that must be looked up to $O(\log N)$, which reduces the overall time to $O(\log^2 N)$.

As a practical optimization, a newly joined node n can ask an immediate neighbor for a copy of its complete finger table and its predecessor. n can use the contents of these tables as hints to help it find the correct values for its own tables, since n's tables will be similar to its neighbors'. This can be shown to reduce the time to fill the finger table to $O(\log N)$.

Updating fingers of existing nodes: Node n will need to be entered into the finger tables of some existing nodes. For example, in Figure 5(a), node 6 becomes the third finger of nodes 0 and 1, and the first and the second finger of node 3.

Figure 6 shows the pseudocode of the *update_finger_table* function that updates existing finger tables. Node n will become the ith finger of node p if and only if (1) p precedes n by at least 2^{i-1}, and (2) the ith finger of node p succeeds n. The first node, p, that can meet these two conditions is the immediate predecessor of $n - 2^{i-1}$. Thus, for a given n, the algorithm starts with the ith finger of node n, and then continues to walk in the counter-clock-wise direction on the identifier circle until it encounters a node whose i^{th} finger precedes n.

We show in the technical report [21] that the number of nodes that need to be updated when a node joins the network is $O(\log N)$ with high probability. Finding and updating these nodes takes $O(\log^2 N)$ time. A more sophisticated scheme can reduce this time to $O(\log N)$; however, we do not present it as we expect implementations to use the algorithm of the following section.

Transferring keys: The last operation that has to be performed when a node n joins the network is to move responsibility for all the keys for which node n is now the successor. Exactly what this entails depends on the higher-layer software using Chord, but typically it would involve moving the data associated with each key to the new node. Node n can become the successor only for keys that were previously the responsibility of the node immediately follow-

```
#define successor finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
  if (n')
    init_finger_table(n');
    update_others();
    // move keys in (predecessor, n] from successor
  else // n is the only node in the network
    for i = 1 to m
      finger[i].node = n;
    predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for i = 1 to m - 1
    if (finger[i + 1].start ∈ [n, finger[i].node))
      finger[i + 1].node = finger[i].node;
    else
      finger[i + 1].node =
          n'.find_successor(finger[i + 1].start);


// update all nodes whose finger
// tables should refer to n
n.update_others()
  for i = 1 to m
    // find last node p whose i^th finger might be n
    p = find_predecessor(n - 2^(i-1));
    p.update_finger_table(n, i);

// if s is i^th finger of n, update n's finger table with s
n.update_finger_table(s, i)
  if (s ∈ [n, finger[i].node))
    finger[i].node = s;
    p = predecessor; // get first node preceding n
    p.update_finger_table(s, i);
```

Figure 6: Pseudocode for the node join operation.

ing n, so n only needs to contact that one node to transfer responsibility for all relevant keys.

5. Concurrent Operations and Failures

In practice Chord needs to deal with nodes joining the system concurrently and with nodes that fail or leave voluntarily. This section describes modifications to the basic Chord algorithms described in Section 4 to handle these situations.

5.1 Stabilization

The join algorithm in Section 4 aggressively maintains the finger tables of all nodes as the network evolves. Since this invariant is difficult to maintain in the face of concurrent joins in a large network, we separate our correctness and performance goals. A basic "stabilization" protocol is used to keep nodes' successor pointers up to date, which is sufficient to guarantee correctness of lookups. Those successor pointers are then used to verify and correct finger table entries, which allows these lookups to be fast as well as correct.

If joining nodes have affected some region of the Chord ring, a lookup that occurs before stabilization has finished can exhibit one of three behaviors. The common case is that all the finger table entries involved in the lookup are reasonably current, and the lookup finds the correct successor in $O(\log N)$ steps. The second case is where successor pointers are correct, but fingers are inaccurate. This yields correct lookups, but they may be slower. In the final case, the nodes in the affected region have incorrect successor pointers, or keys may not yet have migrated to newly joined nodes, and the lookup may fail. The higher-layer software using Chord will notice that the desired data was not found, and has the option of retrying the lookup after a pause. This pause can be short, since stabilization fixes successor pointers quickly.

Our stabilization scheme guarantees to add nodes to a Chord ring in a way that preserves reachability of existing nodes, even in the face of concurrent joins and lost and reordered messages. Stabilization by itself won't correct a Chord system that has split into multiple disjoint cycles, or a single cycle that loops multiple times around the identifier space. These pathological cases cannot be produced by any sequence of ordinary node joins. It is unclear whether they can be produced by network partitions and recoveries or intermittent failures. If produced, these cases could be detected and repaired by periodic sampling of the ring topology.

Figure 7 shows the pseudo-code for joins and stabilization; this code replaces Figure 6 to handle concurrent joins. When node n first starts, it calls $n.join(n')$, where n' is any known Chord node. The $join$ function asks n' to find the immediate successor of n. By itself, $join$ does not make the rest of the network aware of n.

Every node runs *stabilize* periodically (this is how newly joined nodes are noticed by the network). When node n runs *stabilize*, it asks n's successor for the successor's predecessor p, and decides whether p should be n's successor instead. This would be the case if node p recently joined the system. *stabilize* also notifies node n's successor of n's existence, giving the successor the chance to change its predecessor to n. The successor does this only if it knows of no closer predecessor than n.

As a simple example, suppose node n joins the system, and its ID lies between nodes n_p and n_s. n would acquire n_s as its successor. Node n_s, when notified by n, would acquire n as its predecessor. When n_p next runs *stabilize*, it will ask n_s for its predecessor (which is now n); n_p would then acquire n as its successor. Finally, n_p will notify n, and n will acquire n_p as its predecessor. At this point, all predecessor and successor pointers are correct.

```
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);

// periodically verify n's immediate successor,
// and tell the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';

// periodically refresh finger table entries.
n.fix_fingers()
    i = random index > 1 into finger[];
    finger[i].node = find_successor(finger[i].start);
```

Figure 7: Pseudocode for stabilization.

As soon as the successor pointers are correct, calls to *find_predecessor* (and thus *find_successor*) will work. Newly joined nodes that have not yet been fingered may cause *find_predecessor* to initially undershoot, but the loop in the lookup algorithm will nevertheless follow successor (*finger*[1]) pointers through the newly joined nodes until the correct predecessor is reached. Eventually *fix_fingers* will adjust finger table entries, eliminating the need for these linear scans.

The following theorems (proved in the technical report [21]) show that all problems caused by concurrent joins are transient. The theorems assume that any two nodes trying to communicate will eventually succeed.

THEOREM 4. *Once a node can successfully resolve a given query, it will always be able to do so in the future.*

THEOREM 5. *At some time after the last join all successor pointers will be correct.*

The proofs of these theorems rely on an invariant and a termination argument. The invariant states that once node n can reach node r via successor pointers, it always can. To argue termination, we consider the case where two nodes both think they have the same successor s. In this case, each will attempt to notify s, and s will eventually choose the closer of the two (or some other, closer node) as its predecessor. At this point the farther of the two will, by contacting s, learn of a better successor than s. It follows that every node progresses towards a better and better successor over time. This progress must eventually halt in a state where every node is considered the successor of exactly one other node; this defines a cycle (or set of them, but the invariant ensures that there will be at most one).

We have not discussed the adjustment of fingers when nodes join because it turns out that joins don't substantially damage the performance of fingers. If a node has a finger into each interval, then these fingers can still be used even after joins. The distance halving argument is essentially unchanged, showing that $O(\log N)$ hops suffice to reach a node "close" to a query's target. New joins influence the lookup only by getting in between the old predecessor and successor of a target query. These new nodes may need to be scanned linearly (if their fingers are not yet accurate). But unless a

tremendous number of nodes joins the system, the number of nodes between two old nodes is likely to be very small, so the impact on lookup is negligible. Formally, we can state the following:

THEOREM 6. *If we take a stable network with N nodes, and another set of up to N nodes joins the network with no finger pointers (but with correct successor pointers), then lookups will still take $O(\log N)$ time with high probability.*

More generally, so long as the time it takes to adjust fingers is less than the time it takes the network to double in size, lookups should continue to take $O(\log N)$ hops.

5.2 Failures and Replication

When a node n fails, nodes whose finger tables include n must find n's successor. In addition, the failure of n must not be allowed to disrupt queries that are in progress as the system is re-stabilizing.

The key step in failure recovery is maintaining correct successor pointers, since in the worst case *find_predecessor* can make progress using only successors. To help achieve this, each Chord node maintains a "successor-list" of its r nearest successors on the Chord ring. In ordinary operation, a modified version of the *stabilize* routine in Figure 7 maintains the successor-list. If node n notices that its successor has failed, it replaces it with the first live entry in its successor list. At that point, n can direct ordinary lookups for keys for which the failed node was the successor to the new successor. As time passes, *stabilize* will correct finger table entries and successor-list entries pointing to the failed node.

After a node failure, but before stabilization has completed, other nodes may attempt to send requests through the failed node as part of a *find_successor* lookup. Ideally the lookups would be able to proceed, after a timeout, by another path despite the failure. In many cases this is possible. All that is needed is a list of alternate nodes, easily found in the finger table entries preceding that of the failed node. If the failed node had a very low finger table index, nodes in the successor-list are also available as alternates.

The technical report proves the following two theorems that show that the successor-list allows lookups to succeed, and be efficient, even during stabilization [21]:

THEOREM 7. *If we use a successor list of length $r = O(\log N)$ in a network that is initially stable, and then every node fails with probability 1/2, then with high probability find_successor returns the closest living successor to the query key.*

THEOREM 8. *If we use a successor list of length $r = O(\log N)$ in a network that is initially stable, and then every node fails with probability 1/2, then the expected time to execute find_successor in the failed network is $O(\log N)$.*

The intuition behind these proofs is straightforward: a node's r successors all fail with probability $2^{-r} = 1/N$, so with high probability a node will be aware of, so able to forward messages to, its closest living successor.

The successor-list mechanism also helps higher layer software replicate data. A typical application using Chord might store replicas of the data associated with a key at the k nodes succeeding the key. The fact that a Chord node keeps track of its r successors means that it can inform the higher layer software when successors come and go, and thus when the software should propagate new replicas.

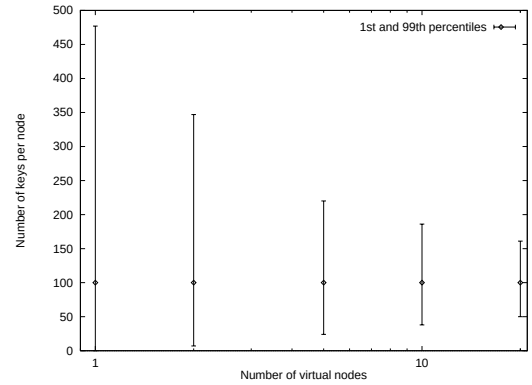


Figure 9: The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node. The network has 10^4 real nodes and stores 10^6 keys.

6. Simulation and Experimental Results

In this section, we evaluate the Chord protocol by simulation. The simulator uses the lookup algorithm in Figure 4 and a slightly older version of the stabilization algorithms described in Section 5. We also report on some preliminary experimental results from an operational Chord-based system running on Internet hosts.

6.1 Protocol Simulator

The Chord protocol can be implemented in an *iterative* or *recursive* style. In the iterative style, a node resolving a lookup initiates all communication: it asks a series of nodes for information from their finger tables, each time moving closer on the Chord ring to the desired successor. In the recursive style, each intermediate node forwards a request to the next node until it reaches the successor. The simulator implements the protocols in an iterative style.

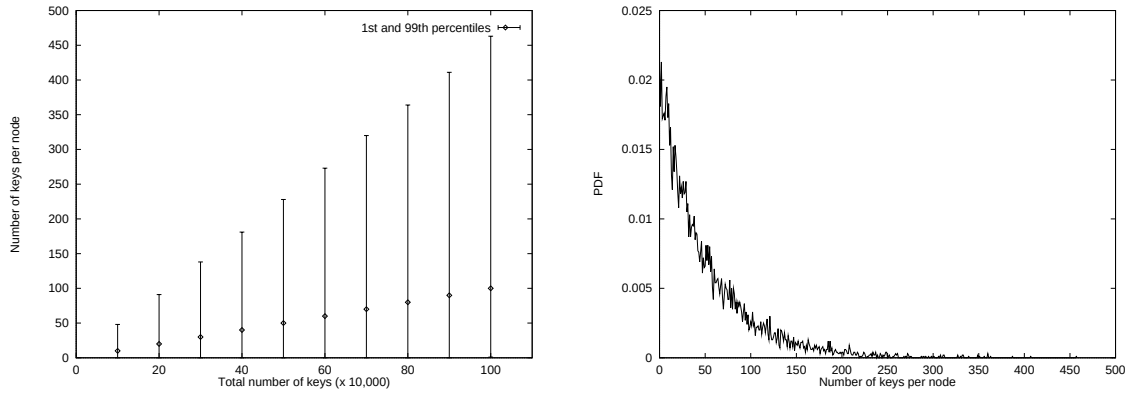
6.2 Load Balance

We first consider the ability of consistent hashing to allocate keys to nodes evenly. In a network with N nodes and K keys we would like the distribution of keys to nodes to be tight around N/K.

We consider a network consisting of 10^4 nodes, and vary the total number of keys from 10^5 to 10^6 in increments of 10^5. For each value, we repeat the experiment 20 times. Figure 8(a) plots the mean and the 1st and 99th percentiles of the number of keys per node. The number of keys per node exhibits large variations that increase linearly with the number of keys. For example, in all cases some nodes store no keys. To clarify this, Figure 8(b) plots the probability density function (PDF) of the number of keys per node when there are 5×10^5 keys stored in the network. The maximum number of nodes stored by any node in this case is 457, or $9.1\times$ the mean value. For comparison, the 99th percentile is $4.6\times$ the mean value.

One reason for these variations is that node identifiers do not uniformly cover the entire identifier space. If we divide the identifier space in N equal-sized bins, where N is the number of nodes, then we might hope to see one node in each bin. But in fact, the probability that a particular bin does not contain any node is $(1-1/N)^N$. For large values of N this approaches $e^{-1} = 0.368$.

As we discussed earlier, the consistent hashing paper solves this problem by associating keys with virtual nodes, and mapping multiple virtual nodes (with unrelated identifiers) to each real node. Intuitively, this will provide a more uniform coverage of the identifier space. For example, if we allocate $\log N$ randomly chosen virtual nodes to each real node, with high probability each of the



(a) (b)

Figure 8: (a) The mean and 1st and 99th percentiles of the number of keys stored per node in a 10^4 node network. (b) The probability density function (PDF) of the number of keys per node. The total number of keys is 5×10^5.

N bins will contain $O(\log N)$ nodes [16]. We note that this does not affect the worst-case query path length, which now becomes $O(\log(N \log N)) = O(\log N)$.

To verify this hypothesis, we perform an experiment in which we allocate r virtual nodes to each real node. In this case keys are associated to virtual nodes instead of real nodes. We consider again a network with 10^4 real nodes and 10^6 keys. Figure 9 shows the 1st and 99th percentiles for $r = 1, 2, 5, 10$, and 20, respectively. As expected, the 99th percentile decreases, while the 1st percentile increases with the number of virtual nodes, r. In particular, the 99th percentile decreases from $4.8\times$ to $1.6\times$ the mean value, while the 1st percentile increases from 0 to $0.5\times$ the mean value. Thus, adding virtual nodes as an indirection layer can significantly improve load balance. The tradeoff is that routing table space usage will increase as each actual node now needs r times as much space to store the finger tables for its virtual nodes. However, we believe that this increase can be easily accommodated in practice. For example, assuming a network with $N = 10^6$ nodes, and assuming $r = \log N$, each node has to maintain a table with only $\log^2 N \simeq 400$ entries.

6.3 Path Length

The performance of any routing protocol depends heavily on the length of the path between two arbitrary nodes in the network. In the context of Chord, we define the path length as the number of nodes traversed during a lookup operation. From Theorem 2, with high probability, the length of the path to resolve a query is $O(\log N)$, where N is the total number of nodes in the network.

To understand Chord's routing performance in practice, we simulated a network with $N = 2^k$ nodes, storing 100×2^k keys in all. We varied k from 3 to 14 and conducted a separate experiment for each value. Each node in an experiment picked a random set of keys to query from the system, and we measured the path length required to resolve each query.

Figure 10(a) plots the mean, and the 1st and 99th percentiles of path length as a function of k. As expected, the mean path length increases logarithmically with the number of nodes, as do the 1st and 99th percentiles. Figure 10(b) plots the PDF of the path length for a network with 2^{12} nodes ($k = 12$).

Figure 10(a) shows that the path length is about $\frac{1}{2}\log_2 N$. The reason for the $\frac{1}{2}$ is as follows. Consider some random node and a random query. Let the distance in identifier space be considered in binary representation. The most significant (say i^{th}) bit of this distance can be corrected to 0 by following the node's i^{th} finger. If the next significant bit of the distance is 1, it too needs to be corrected by following a finger, but if it is 0, then no $i - 1^{st}$ finger is followed—instead, we move on the the $i-2^{nd}$ bit. In general, the number of fingers we need to follow will be the number of ones in the binary representation of the distance from node to query. Since the distance is random, we expect half the $\log N$ bits to be ones.

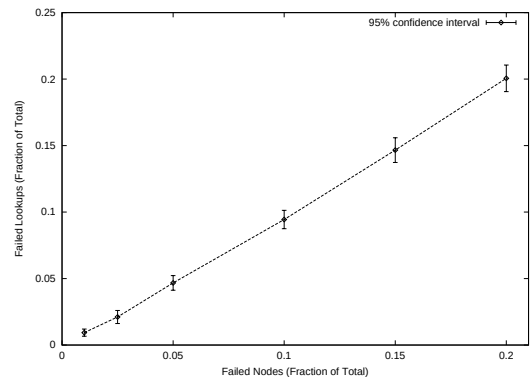
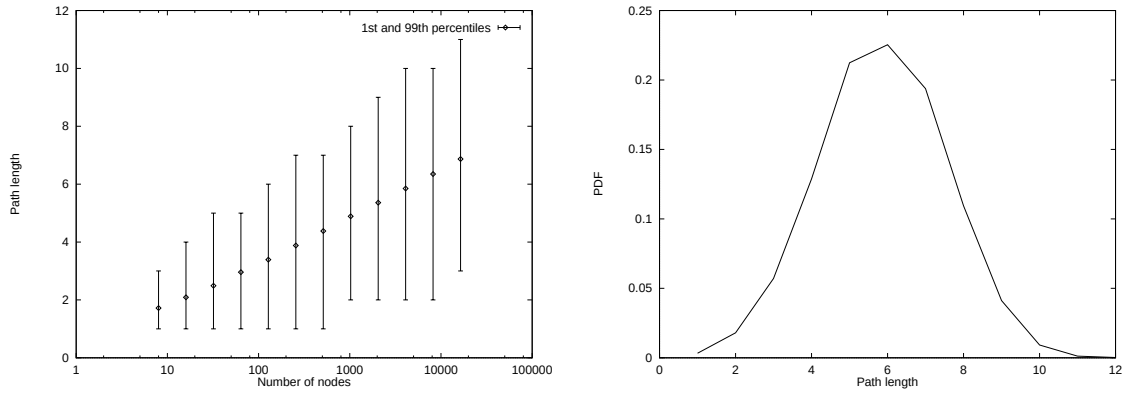


Figure 11: The fraction of lookups that fail as a function of the fraction of nodes that fail.

6.4 Simultaneous Node Failures

In this experiment, we evaluate the ability of Chord to regain consistency after a large percentage of nodes fail simultaneously. We consider again a 10^4 node network that stores 10^6 keys, and randomly select a fraction p of nodes that fail. After the failures occur, we wait for the network to finish stabilizing, and then measure the fraction of keys that could not be looked up correctly. A correct lookup of a key is one that finds the node that was originally responsible for the key, before the failures; this corresponds to a system that stores values with keys but does not replicate the values or recover them after failures.

Figure 11 plots the mean lookup failure rate and the 95% confidence interval as a function of p. The lookup failure rate is almost exactly p. Since this is just the fraction of keys expected to be lost due to the failure of the responsible nodes, we conclude that there is no significant lookup failure in the Chord network. For example, if the Chord network had partitioned in two equal-sized halves, we



(a) (b)

Figure 10: (a) The path length as a function of network size. (b) The PDF of the path length in the case of a 2^{12} node network.

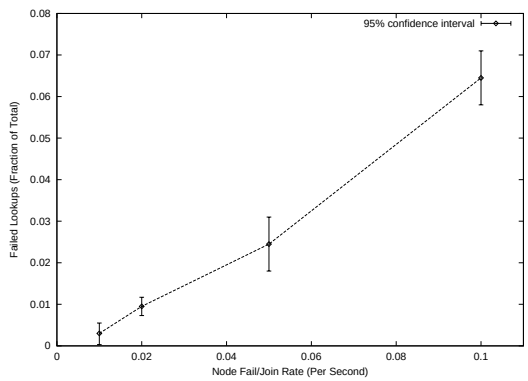


Figure 12: The fraction of lookups that fail as a function of the rate (over time) at which nodes fail and join. Only failures caused by Chord state inconsistency are included, not failures due to lost keys.

would expect one-half of the requests to fail because the querier and target would be in different partitions half the time. Our results do not show this, suggesting that Chord is robust in the face of multiple simultaneous node failures.

6.5 Lookups During Stabilization

A lookup issued after some failures but before stabilization has completed may fail for two reasons. First, the node responsible for the key may have failed. Second, some nodes' finger tables and predecessor pointers may be inconsistent due to concurrent joins and node failures. This section evaluates the impact of continuous joins and failures on lookups.

In this experiment, a lookup is considered to have succeeded if it reaches the current successor of the desired key. This is slightly optimistic: in a real system, there might be periods of time in which the real successor of a key has not yet acquired the data associated with the key from the previous successor. However, this method allows us to focus on Chord's ability to perform lookups, rather than on the higher-layer software's ability to maintain consistency of its own data. Any query failure will be the result of inconsistencies in Chord. In addition, the simulator does not retry queries: if a query is forwarded to a node that is down, the query simply fails. Thus, the results given in this section can be viewed as the worst-case scenario for the query failures induced by state inconsistency.

Because the primary source of inconsistencies is nodes joining and leaving, and because the main mechanism to resolve these inconsistencies is the stabilize protocol, Chord's performance will be sensitive to the frequency of node joins and leaves versus the frequency at which the stabilization protocol is invoked.

In this experiment, key lookups are generated according to a Poisson process at a rate of one per second. Joins and failures are modeled by a Poisson process with the mean arrival rate of R. Each node runs the stabilization routines at randomized intervals averaging 30 seconds; unlike the routines in Figure 7, the simulator updates all finger table entries on every invocation. The network starts with 500 nodes.

Figure 12 plots the average failure rates and confidence intervals. A node failure rate of 0.01 corresponds to one node joining and leaving every 100 seconds on average. For comparison, recall that each node invokes the stabilize protocol once every 30 seconds. In other words, the graph x axis ranges from a rate of 1 failure per 3 stabilization steps to a rate of 3 failures per one stabilization step. The results presented in Figure 12 are averaged over approximately two hours of simulated time. The confidence intervals are computed over 10 independent runs.

The results of figure 12 can be explained roughly as follows. The simulation has 500 nodes, meaning lookup path lengths average around 5. A lookup fails if its finger path encounters a failed node. If k nodes fail, the probability that one of them is on the finger path is roughly $5k/500$, or $k/100$. This would suggest a failure rate of about 3% if we have 3 failures between stabilizations. The graph shows results in this ball-park, but slightly worse since it might take more than one stabilization to completely clear out a failed node.

6.6 Experimental Results

This section presents latency measurements obtained from a prototype implementation of Chord deployed on the Internet. The Chord nodes are at ten sites on a subset of the RON test-bed in the United States [1], in California, Colorado, Massachusetts, New York, North Carolina, and Pennsylvania. The Chord software runs on UNIX, uses 160-bit keys obtained from the SHA-1 cryptographic hash function, and uses TCP to communicate between nodes. Chord runs in the iterative style. These Chord nodes are part of an experimental distributed file system [7], though this section considers only the Chord component of the system.

Figure 13 shows the measured latency of Chord lookups over a range of numbers of nodes. Experiments with a number of nodes larger than ten are conducted by running multiple independent

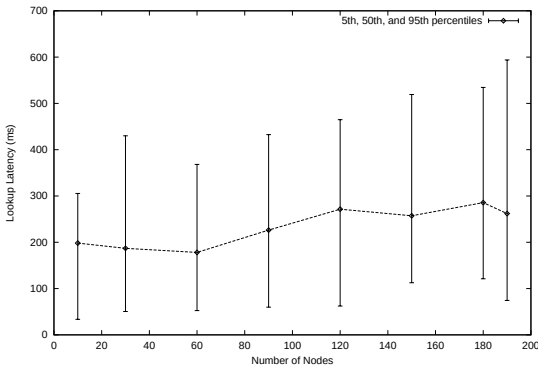


Figure 13: Lookup latency on the Internet prototype, as a function of the total number of nodes. Each of the ten physical sites runs multiple independent copies of the Chord node software.

copies of the Chord software at each site. This is different from running $O(\log N)$ virtual nodes at each site to provide good load balance; rather, the intention is to measure how well our implementation scales even though we do not have more than a small number of deployed nodes.

For each number of nodes shown in Figure 13, each physical site issues 16 Chord lookups for randomly chosen keys one-by-one. The graph plots the median, the 5th, and the 95th percentile of lookup latency. The median latency ranges from 180 to 285 ms, depending on number of nodes. For the case of 180 nodes, a typical lookup involves five two-way message exchanges: four for the Chord lookup, and a final message to the successor node. Typical round-trip delays between sites are 60 milliseconds (as measured by `ping`). Thus the expected lookup time for 180 nodes is about 300 milliseconds, which is close to the measured median of 285. The low 5th percentile latencies are caused by lookups for keys close (in ID space) to the querying node and by query hops that remain local to the physical site. The high 95th percentiles are caused by lookups whose hops follow high delay paths.

The lesson from Figure 13 is that lookup latency grows slowly with the total number of nodes, confirming the simulation results that demonstrate Chord's scalability.

7. Future Work

Based on our experience with the prototype mentioned in Section 6.6, we would like to improve the Chord design in the following areas.

Chord currently has no specific mechanism to heal partitioned rings; such rings could appear locally consistent to the stabilization procedure. One way to check global consistency is for each node n to periodically ask other nodes to do a Chord lookup for n; if the lookup does not yield node n, there may be a partition. This will only detect partitions whose nodes know of each other. One way to obtain this knowledge is for every node to know of the same small set of initial nodes. Another approach might be for nodes to maintain long-term memory of a random set of nodes they have encountered in the past; if a partition forms, the random sets in one partition are likely to include nodes from the other partition.

A malicious or buggy set of Chord participants could present an incorrect view of the Chord ring. Assuming that the data Chord is being used to locate is cryptographically authenticated, this is a threat to availability of data rather than to authenticity. The same approach used above to detect partitions could help victims realize that they are not seeing a globally consistent view of the Chord ring.

An attacker could target a particular data item by inserting a node into the Chord ring with an ID immediately following the item's key, and having the node return errors when asked to retrieve the data. Requiring (and checking) that nodes use IDs derived from the SHA-1 hash of their IP addresses makes this attack harder.

Even $\log N$ messages per lookup may be too many for some applications of Chord, especially if each message must be sent to a random Internet host. Instead of placing its fingers at distances that are all powers of 2, Chord could easily be changed to place its fingers at distances that are all integer powers of $1 + 1/d$. Under such a scheme, a single routing hop could decrease the distance to a query to $1/(1+d)$ of the original distance, meaning that $\log_{1+d} N$ hops would suffice. However, the number of fingers needed would increase to $\log N/(\log(1 + 1/d) \approx O(d \log N)$.

A different approach to improving lookup latency might be to use server selection. Each finger table entry could point to the first k nodes in that entry's interval on the ID ring, and a node could measure the network delay to each of the k nodes. The k nodes are generally equivalent for purposes of lookup, so a node could forward lookups to the one with lowest delay. This approach would be most effective with recursive Chord lookups, in which the node measuring the delays is also the node forwarding the lookup.

8. Conclusion

Many distributed peer-to-peer applications need to determine the node that stores a data item. The Chord protocol solves this challenging problem in decentralized manner. It offers a powerful primitive: given a key, it determines the node responsible for storing the key's value, and does so efficiently. In the steady state, in an N-node network, each node maintains routing information for only about $O(\log N)$ other nodes, and resolves all lookups via $O(\log N)$ messages to other nodes. Updates to the routing information for nodes leaving and joining require only $O(\log^2 N)$ messages.

Attractive features of Chord include its simplicity, provable correctness, and provable performance even in the face of concurrent node arrivals and departures. It continues to function correctly, albeit at degraded performance, when a node's information is only partially correct. Our theoretical analysis, simulations, and experimental results confirm that Chord scales well with the number of nodes, recovers from large numbers of simultaneous node failures and joins, and answers most lookups correctly even during recovery.

We believe that Chord will be a valuable component for peer-to-peer, large-scale distributed applications such as cooperative file sharing, time-shared available storage systems, distributed indices for document and service discovery, and large-scale distributed computing platforms.

Acknowledgments

We thank Frank Dabek for the measurements of the Chord prototype described in Section 6.6, and David Andersen for setting up the testbed used in those measurements.

9. References

[1] ANDERSEN, D. Resilient overlay networks. Master's thesis, Department of EECS, MIT, May 2001. `http://nms.lcs.mit.edu/projects/ron/`.

[2] BAKKER, A., AMADE, E., BALLINTIJN, G., KUZ, I., VERKAIK, P., VAN DER WIJK, I., VAN STEEN, M., AND TANENBAUM., A.

The Globe distribution network. In *Proc. 2000 USENIX Annual Conf. (FREENIX Track)* (San Diego, CA, June 2000), pp. 141–152.

[3] CHEN, Y., EDLER, J., GOLDBERG, A., GOTTLIEB, A., SOBTI, S., AND YIANILOS, P. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital libraries* (Berkeley, CA, Aug. 1999), pp. 28–37.

[4] CLARKE, I. A distributed decentralised information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.

[5] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, California, June 2000). `http://freenet.sourceforge.net`.

[6] DABEK, F., BRUNSKILL, E., KAASHOEK, M. F., KARGER, D., MORRIS, R., STOICA, I., AND BALAKRISHNAN, H. Building peer-to-peer systems with Chord, a distributed location service. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Elmau/Oberbayern, Germany, May 2001), pp. 71–76.

[7] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (To appear; Banff, Canada, Oct. 2001).

[8] DRUSCHEL, P., AND ROWSTRON, A. Past: Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS 2001)* (Elmau/Oberbayern, Germany, May 2001), pp. 65–70.

[9] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

[10] Gnutella. `http://gnutella.wego.com/`.

[11] KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (El Paso, TX, May 1997), pp. 654–663.

[12] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (Boston, MA, November 2000), pp. 190–201.

[13] LEWIN, D. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Department of EECS, MIT, 1998. Available at the MIT Library, `http://thesis.mit.edu/`.

[14] LI, J., JANNOTTI, J., DE COUTO, D., KARGER, D., AND MORRIS, R. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking* (Boston, Massachusetts, August 2000), pp. 120–130.

[15] MOCKAPETRIS, P., AND DUNLAP, K. J. Development of the Domain Name System. In *Proc. ACM SIGCOMM* (Stanford, CA, 1988), pp. 123–133.

[16] MOTWANI, R., AND RAGHAVAN, P. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.

[17] Napster. `http://www.napster.com/`.

[18] Ohaha, Smart decentralized peer-to-peer sharing. `http://www.ohaha.com/design.html`.

[19] PLAXTON, C., RAJARAMAN, R., AND RICHA, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA* (Newport, Rhode Island, June 1997), pp. 311–320.

[20] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM* (San Diego, CA, August 2001).

[21] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. Tech. Rep. TR-819, MIT LCS, March 2001. `http://www.pdos.lcs.mit.edu/chord/papers/`.

[22] VAN STEEN, M., HAUCK, F., BALLINTIJN, G., AND TANENBAUM, A. Algorithmic design of the Globe wide-area location service. *The Computer Journal 41*, 5 (1998), 297–310.

Scalable, Distributed Data Structures for Internet Service Construction

Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler
The University of California at Berkeley
{gribble,brewer,jmh,culler}@cs.berkeley.edu

Abstract

This paper presents a new persistent data management layer designed to simplify cluster-based Internet service construction. This self-managing layer, called a distributed data structure (DDS), presents a conventional single-site data structure interface to service authors, but partitions and replicates the data across a cluster. We have designed and implemented a distributed hash table DDS that has properties necessary for Internet services (incremental scaling of throughput and data capacity, fault tolerance and high availability, high concurrency, consistency, and durability). The hash table uses two-phase commits to present a coherent view of its data across all cluster nodes, allowing any node to service any task. We show that the distributed hash table simplifies Internet service construction by decoupling service-specific logic from the complexities of persistent, consistent state management, and by allowing services to inherit the necessary service properties from the DDS rather than having to implement the properties themselves. We have scaled the hash table to a 128 node cluster, 1 terabyte of storage, and an in-core read throughput of 61,432 operations/s and write throughput of 13,582 operations/s.

1 Introduction

Internet services are successfully bringing infrastructural computing to the masses. Millions of people depend on Internet services for applications like searching, instant messaging, directories, and maps, and also to safeguard and provide access to their personal data (such as email and calendar entries). As a direct consequence of this increasing user dependence, today's Internet services must possess many of the same properties as the telephony and power infrastructures. These *service properties* include the ability to scale to large, rapidly growing user populations, high availability in the face of partial failures, strictly maintaining the consistency of users' data, and operational manageability.

It is challenging for a service to achieve all of these properties, especially when it must manage large amounts of persistent state, as this state must remain available and consistent even if individual disks, processes, or processors crash. Unfortunately, the consequences of failing to achieve the properties are harsh, including lost data, angry users, and perhaps financial liability. Even worse, there appear to be few reusable Internet service construction platforms (or data management platforms) that successfully provide all of the properties.

Many projects and products propose using software platforms on clusters to address these challenges and to simplify Internet service construction [1, 2, 6, 15]. These platforms typically rely on commercial databases or distributed file systems for persistent data management, or they do not address data management at all, forcing service authors to implement their own service-specific data management layer. We argue that databases and file systems have not been designed with Internet service workloads, the service properties, and cluster environments specifically in mind, and as a result, they fail to provide the right scaling, consistency, or availability guarantees that services require.

In this paper, we bring scalable, available, and consistent data management capabilities to cluster platforms by designing and implementing a reusable, cluster-based storage layer, called a *distributed data structure (DDS)*, specifically designed for the needs of Internet services. A DDS presents a conventional single site in-memory data structure interface to applications, and durably manages the data behind this interface by distributing and replicating it across the cluster. Services inherit the aforementioned service properties by using a DDS to store and manage all persistent service state, shielding service authors from the complexities of scalable, available, persistent data storage, thus simplifying the process of implementing new Internet services.

We believe that given a small set of DDS types (such as a hash table, a tree, and an administrative log), authors will be able to build a large class of interesting and sophisticated servers. This paper describes the design, architecture, and implementation of one such distributed data structure (a distributed hash table built in Java). We evaluate

its performance, scalability and availability, and its ability to simplify service construction.

1.1 Clusters of Workstations

In [15], it is argued that clusters of workstations (commodity PC's with a high-performance network) are a natural platform for Internet services. Each cluster node is an independent failure boundary, which means that replicating computation and data can provide fault tolerance. A cluster permits incremental scalability: if a service runs out of capacity, a good software architecture allows nodes to be added to the cluster, linearly increasing the service's capacity. A cluster has natural parallelism: if appropriately balanced, all CPUs, disks, and network links can be used simultaneously, increasing the throughput of the service as the cluster grows. Clusters have high throughput, low latency redundant system area networks (SAN) that can achieve 1 Gb/s throughput with 10 to 100 μs latency.

1.2 Internet Service Workloads

Popular Internet services process hundreds of millions of tasks per day. A task is usually "small", causing a small amount of data to be transferred and computation to be performed. For example, according to press releases, Yahoo (`http://www.yahoo.com`) serves 625 million page views per day. Randomly sampled pages from the Yahoo directory average 7KB of HTML data and 10KB of image data. Similarly, AOL's web proxy cache (`http://www.aol.com`) handles 5.2 billion web requests per day, with an average response size of 5.5 KB. Services often take hundreds of milliseconds to process a given task, and their responses can take many seconds to flow back to clients over what are predominantly low bandwidth last-hop network links [19]. Given this high task throughput and non-negligible latency, a service may handle thousands of tasks simultaneously. Human users are typically the ultimate source of tasks; because users usually generate a small number of concurrent tasks (e.g., 4 parallel HTTP GET requests are typically spawned when a user requests a web page), the large set of tasks being handled by a service are largely independent.

2 Distributed Data Structures

A distributed data structure (DDS) is a self-managing storage layer designed to run on a cluster of workstations [2] and to handle Internet service workloads. A DDS has all of the previously mentioned service properties: high throughput, high concurrency, availability, incrementally scalability, and strict consistency of its data. Service authors see the interface to a DDS as a conventional data structure, such as a hash table, a tree, or a log. Behind this interface, the DDS platform hides all of the mechanisms used to access, partition, replicate, scale, and recover data. Because these complex mechanisms are hidden behind the simple DDS interface, authors only need to worry about service-specific logic when implementing a new service. The difficult issues of managing persistent state are handled by the DDS platform.

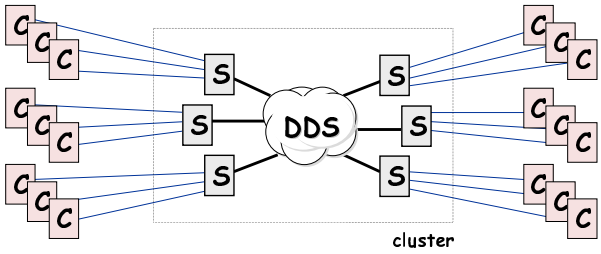


Figure 1: **High-level view of a DDS:** a DDS is a self-managing, cluster-based data repository. All service instances (S) in the cluster see the same consistent image of the DDS; as a result, any WAN client (C) can communicate with any service instance.

Figure 1 shows a high-level illustration of a DDS. All cluster nodes have access to the DDS and see the same consistent image of the DDS. As long as services keep all persistent state in the DDS, any service instance in the cluster can handle requests from any client, although we expect clients will have affinity to particular service instances to allow session state to accumulate.

The idea of having a storage layer to manage durable state is not new, of course; databases and file systems have done this for many decades. The novel aspects of a DDS are the level of abstraction that it presents to service authors, the consistency model it supports, the access behavior (concurrency and throughput demands) that it presupposes, and its many design and implementation choices that are made based on its expected runtime environment and the types of failures that it should withstand. A direct comparison between databases, distributed file systems, and DDS's helps to show this.

Relational database management systems (RDBMS): an RDBMS offers extremely strong durability and consistency guarantees, namely ACID properties derived from the use of transactions [18], but these ACID properties can come at high cost in terms of complexity and overhead. As a result, Internet services that rely on RDBMS backends typically go to great lengths to reduce the workload presented to the RDBMS, using techniques such as query caching in front ends [15, 21, 32]. RDBMS's offer a high degree of data independence, which is a powerful abstraction that adds addi-

tional complexity and performance overhead. The many layers of most RDBMS's (such as SQL parsing, query optimization, access path selection, etc.) permit users to decouple the logical structure of their data from its physical layout. This decoupling allows users to dynamically construct and issue queries over the data that are limited only by what can be expressed in the SQL language, but data independence can make parallelization (and therefore scaling) hard in the general case. From the perspective of the service properties, an RDBMS always chooses consistency over availability: if there are media or processor failures, an RDBMS can become unavailable until the failure is resolved, which is unacceptable for Internet services.

Distributed file systems: file systems have less strictly defined consistency models. Some (e.g., NFS [31]) have weak consistency guarantees, while others (e.g., Frangipani [33] or AFS [12]) guarantee a coherent filesystem image across all clients, with locking typically done at the granularity of files. The scalability of distributed file systems similarly varies; some use centralized file servers, and thus do not scale. Others such as xFS [3] are completely serverless, and in theory can scale to arbitrarily large capacities. File systems expose a relatively low level interface with little data independence; a file system is organized as a hierarchical directory of files, and files are variable-length arrays of bytes. These elements (directories and files) are directly exposed to file system clients; clients are responsible for logically structuring their application data in terms of directories, files, and bytes inside those files.

Distributed data structures (DDS): a DDS has a strictly defined consistency model: all operations on its elements are atomic, in that any operation completes entirely, or not at all. DDS's have one-copy equivalence, so although data elements in a DDS are replicated, clients see a single, logical data item. Two-phase commits are used to keep replicas coherent, and thus all clients see the same image of a DDS through its interface. Transactions across multiple elements or operations are not currently supported: as we will show later, many of our current protocol design decisions and implementation choices exploit the lack of transactional support for greater efficiency and simplicity. There are Internet services that require transactions (e.g. for e-commerce); we can imagine building a transactional DDS, but it is beyond the scope of this paper, and we believe that the atomic single-element updates and coherence provided by our current DDS are strong enough to support interesting services.

A DDS's interface is more structured and at a higher level than that of a file system. The granularity of an operation is a complete data structure element rather than an arbitrary byte range. The set of operations over the data in a DDS is fixed by a small set of methods exposed by the DDS API, unlike an RDBMS in which operations are defined by the set of expressible declarations in SQL. The query parsing and optimization stages of an RDBMS are completely obviated in a DDS, but the DDS interface is less flexible and offers less data independence.

In summary, by choosing a level of abstraction somewhere in between that of an RDBMS and a file system, and by choosing a well-defined and simple consistency model, we have been able to design and implement a DDS with all of the service properties. It has been our experience that the DDS interfaces, although not as general as SQL, are rich enough to successfully build sophisticated services.

3 Assumptions and Design Principles

In this section of the paper, we present the design principles that guided us while building our distributed hash table DDS. We also state a number of key assumptions we made regarding our cluster environment, failure modes that the DDS can handle, and the workloads it will receive.

Separation of concerns: the clean separation of service code from storage management simplifies system architecture by decoupling the complexities of state management from those of service construction. Because persistent service state is kept in the DDS, service instances can crash (or be gracefully shut down) and restart without a complex recovery process. This greatly simplifies service construction, as authors need only worry about service-specific logic, and not the complexities of data partitioning, replication, and recovery.

Appeal to properties of clusters: in addition to the properties listed in section 1.1, we require that our cluster is physically secure and well-administered. Given all of these properties, a cluster represents a carefully controlled environment in which we have the greatest chance of being able to provide all of the service properties. For example, its low latency SAN (10-100 μs instead of 10-100 ms for the wide-area Internet) means that two-phase commits are not prohibitively expensive. The SAN's high redundancy means that the probability of a network partition can be made arbitrarily small, and thus we need not consider partitions in our protocols. An uninterruptible power supply (UPS) and good system administration help to ensure that the probability of system-wide simultaneous hardware failure is extremely low; we can thus rely on data being available in more than one failure boundary (i.e., the physical memory or disk of more than one

node) while designing our recovery protocols.[1]

Design for high throughput and high concurrency: given the workloads presented in section 1.2, the control structure used to effect concurrency is critical. Techniques often used by web servers, such as process-per-task or thread-per-task, do not scale to our needed degree of concurrency. Instead, we use asynchronous, event-driven style of control flow in our DDS, similar to that espoused by modern high performance servers [5, 20] such as the Harvest web cache [8] and Flash web server [28]. A convenient side-effect of this style is that layering is inexpensive and flexible, as layers can be constructed by chaining together event handlers. Such chaining also facilitates interposition: a "middleman" event handler can be easily and dynamically patched between two existing handlers. In addition, if a server experiences a burst of traffic, the burst is absorbed in event queues, providing *graceful degradation* by preserving the throughput of the server but temporarily increasing latency. By contrast, thread-per-task systems degrade in both throughput and latency if bursts are absorbed by additional threads.

3.1 Assumptions

If one DDS node cannot communicate with another, we assume it is because this other node has stopped executing (due to a planned shutdown or a crash); we assume that network partitions do not occur inside our cluster, and that DDS software components are fail-stop. The need for no network partitions is addressed by the high redundancy of our network, as previously mentioned. We have attempted to induce fail-stop behavior in our software by having it terminate its own execution if it encounters an unexpected condition, rather than attempting to gracefully recover from such a condition. These strong assumptions have been valid in practice; we have never experienced an unplanned network partition in our cluster, and our software has always behaved in a fail-stop manner. We further assume that software failures in the cluster are independent. We replicate all durable data at more than one place in the cluster, but we assume that at least one replica is active (has not failed) at all times. We also assume some degree of synchrony, in that processes take a bounded amount of time to execute tasks, and that messages take a bounded amount of time to be delivered.

We make several assumptions about the workload presented to our distributed hash tables. A table's key space is the set of 64-bit integers; we assume that the population density over this space is even (i.e. the probability that a given key exists in the table is a function of the number of values in the table, but not of the particular key). We don't assume that all keys are accessed equiprobably, but rather that the "working set" of hot keys is larger than the number of nodes in our cluster. We then assume that a partitioning strategy that maps fractions of the keyspace to cluster nodes based on the nodes' relative processing speed will induce a balanced workload. Our current DDS design does not gracefully handle a small number of extreme hotspots (i.e., if a handful of keys receive most of the workload). If there are many such hotspots, however, then our partitioning strategy will probabilistically balance them across the cluster. Failure of these workload assumptions can result in load imbalances across the cluster, leading to a reduction in throughput.

Finally, we assume that tables are large and long lived. Hash table creations and destructions are relatively rare events: the common case is for hash tables to serve read, write, and remove operations.

4 Distributed Hash Tables: Architecture and Implementation

In this section, we present the architecture and implementation of a distributed hash table DDS. Figure 2 illustrates our hash table's architecture, which consists of the following components:

Client: a client consists of service-specific software running on a client machine that communicates across the wide area with one of many service instances running in the cluster. The mechanism by which the client selects a service instance is beyond the scope of this work, but it typically involves DNS round robin [7], a service-specific protocol, or level 4 or level 7 load-balancing switches on the edge of the cluster. An example of a client is a web browser, in which case the service would be a web server. Note that clients are completely unaware of DDS's: no part of the DDS system runs on a client.

Service: a service is a set of cooperating software processes, each of which we call a service instance. Service instances communicate with wide-area clients and perform some application-level function. Services may have soft state (state which may be lost and recomputed if necessary), but they rely on the hash table to manage all persistent state.

Hash table API: the hash table API is the boundary between a service instance and its "DDS library". The API provides services with `put()`, `get()`, `remove()`, `create()`, and `destroy()` operations on hash tables. Each operation is atomic, and all services see the same coherent image of all exist-

[1] We do have a checkpoint mechanism (discussed later) that permits us to recover in the case that any of these cluster properties fail, however all state changes that happen after the last checkpoint will be lost should this occur.

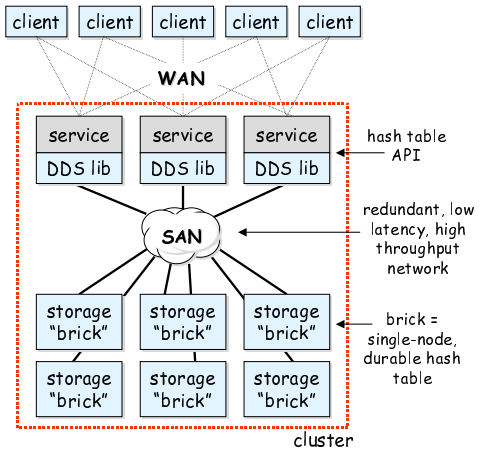


Figure 2: **Distributed hash table architecture:** each box in the diagram represents a software process. In the simplest case, each process runs on its own physical machine, however there is nothing preventing processes from sharing machines.

ing hash tables through this API. Hash table names are strings, hash table keys are 64 bit integers, and hash table values are opaque byte arrays; operations affect hash table values in their entirety.

DDS library: the DDS library is a Java class library that presents the hash table API to services. The library accepts hash table operations, and cooperates with the "bricks" to realize those operations. The library contains only soft state, including metadata about the cluster's current configuration and the partitioning of data in the distributed hash tables across the "bricks". The DDS library acts as the two-phase commit coordinator for state-changing operations on the distributed hash tables.

Brick: bricks are the only system components that manage durable data. Each brick manages a set of network-accessible single node hash tables. A brick consists of a buffer cache, a lock manager, a persistent chained hash table implementation, and network stubs and skeletons for remote communication. Typically, we run one brick per CPU in the cluster, and thus a 4-way SMP will house 4 bricks. Bricks may run on dedicated nodes, or they may share nodes with other components.

4.1 Partitioning, Replication, and Replica Consistency

A distributed hash table provides incremental scalability of throughput and data capacity as more nodes are added to the cluster. To achieve this, we horizontally partition tables to spread operations and data across bricks. Each brick thus stores some number of *partitions* of each table in the system, and when new nodes are added to the cluster, this partitioning is altered so that data is spread onto the new node. Because of our workload assumptions (section 3.1), this horizontal partitioning evenly spreads both load and data across the cluster.

Given that the data in the hash table is spread across multiple nodes, if any of those nodes fail, then a portion of the hash table will become unavailable. For this reason, each partition in the hash table is replicated on more than one cluster node. The set of replicas for a partition form a *replica group*; all replicas in the group are kept strictly coherent with each other. Any replica can be used to service a `get()`, but all replicas must be updated during a `put()` or `remove()`. If a node fails, the data from its partitions is available on the surviving members of the partitions' replica groups. Replica group membership is thus dynamic; when a node fails, all of its replicas are removed from their replica groups. When a node joins the cluster, it may be added to the replica groups of some partitions (such as in the case of recovery, described later).

To maintain consistency when state changing operations (`put()` and `remove()`) are issued against a partition, all replicas of that partition must be synchronously updated. We use an optimistic two-phase commit protocol to achieve consistency, with the DDS library serving as the commit coordinator and the replicas serving as the participants. If the DDS library crashes after *prepare* messages are sent, but before any *commit* messages are sent, the replicas will time out and abort the operation.

However, if the DDS library crashes after sending out any *commits*, then all replicas must commit. For the sake of availability, we do not rely on the DDS library to recover after a crash and issuing pending *commits*. Instead, replicas store short in-memory logs of recent state changing operations and their outcomes. If a replica times out while waiting for a *commit*, that replica communicates with all of its peers to find out if any have received a *commit* for that operation, and if so, the replica commits as well; if not, the replica aborts. Because all peers in the replica group that time out while waiting for a commit communicate with all other peers, if any receives a commit, then all will commit.

Any replica may abort during the first phase of the two-phase commit (e.g., if the replica cannot obtain a write lock on a key). If the DDS library receives any *abort* messages at the end of the first phase, it sends *aborts* to all replicas in the second phase. Replicas do not commit side-effects unless they receive a *commit* message in the second phase.

If a replica crashes during a two-phase commit, the DDS library simply removes it from its replica group and continues onward. Thus, all replica groups shrink over time; we rely on a recovery mech-

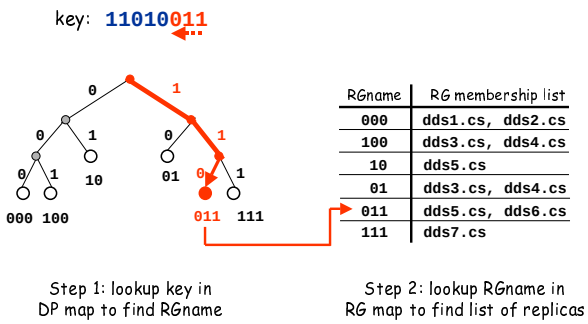


Figure 3: **Distributed hash table metadata maps:** this illustration highlights the steps taken to discover the set of replica groups which serve as the backing store for a specific hash table key. The key is used to traverse the DP map trie and retrieve the name of the key's replica group. The replica group name is then used looked up in the RG map to find the group's current membership.

anism (described later) for crashed replicas to rejoin the replica group. We made the significant optimization that the image of each replica must only be consistent through its brick's cache, rather than having a consistent on-disk image. This allows us to have a purely conflict-driven cache eviction policy, rather than having to force cache elements out to ensure on-disk consistency. An implication of this is that if all members of a replica group crash, that partition is lost. We assume nodes are independent failure boundaries (section 3.1); there must be no systematic software failure across nodes, and the cluster's power supply must be uninterruptible.

Our two-phase commit mechanism gives *atomic updates* to the hash table. It does not, however, give transactional updates. If a service wishes to update more than one element atomically, our DDS does not provide any help. Adding transactional support to our DDS infrastructure is a topic of future work, but this would require significant additional complexity such as distributed deadlock detection and undo/redo logs for recovery.

We do have a checkpoint mechanism in our distributed hash table that allows us to force the on-disk image of all partitions to be consistent; the disk images can then be backed up for disaster recovery. This checkpoint mechanism is extremely heavyweight, however; during the checkpointing of a hash table, no state-changing operations are allowed. We currently rely on system administrators to decide when to initiate checkpoints.

4.2 Metadata maps

To find the partition that manages a particular hash table key, and to determine the list of replicas in partitions' replica groups, the DDS libraries consult two metadata maps that are replicated on each node of the cluster. Each hash table in the cluster has its own pair of metadata maps.

The first map is called the *data partitioning (DP) map*. Given a hash table key, the DP map returns the name of the key's partition. The DP map thus controls the horizontal partitioning of data across the bricks. As shown in figure 3, the DP map is a trie over hash table keys; to find a key's partition, key bits are used to walk down the trie, starting from the least significant key bit until a leaf node is found. As the cluster grows, the DP trie subdivides in a "split" operation. For example, partition `10` in the DP trie of figure 3 could split into partitions `010` and `110`; when this happens, the keys in the old partition are shuffled across the two new partitions. The opposite of a split is a "merge"; if the cluster is shrunk, two partitions with a common parent in the trie can be merged into their parent. For example, partitions `000` and `100` in figure 3 could be merged into a single partition `00`.

The second map is called the *replica group (RG) membership map*. Given a partition name, the RG map returns a list of bricks that are currently serving as replicas in the partition's replica group. The RG maps are dynamic: if a brick fails, it is removed from all RG maps that contain it. A brick joins a replica group after finishing recovery. An invariant that must be preserved is that the replica group membership maps for all partitions in the hash table must have at least one member.

The maps are replicated on each cluster node, in both the DDS libraries and the bricks. The maps must be kept consistent, otherwise operations may be applied to the wrong bricks. Instead of enforcing consistency synchronously, we allow the libraries' maps to drift out of date, but lazily update them when they are used to perform operations. The DDS library piggybacks hashes of the maps[2] on operations sent to bricks; if a brick detects that either map used is out of date, the brick fails the operation and returns a "repair" to the library. Thus, all maps become eventually consistent as they are used. Because of this mechanism, libraries can be restarted with out of date maps, and as the library gets used its maps become consistent.

To `put()` a key and value into a hash table, the DDS library servicing the operation consults its DP map to determine the correct partition for the key. It then looks up that partition name in its RG map to find the current set of bricks serving as replicas, and finally performs a two-phase commit across these replicas. To do a `get()` of a key, a similar process is used, except that the DDS library can

[2] It is important to use large enough of a hash to make the probability of collision negligible; we currently use 32 bits.

select any of the replicas listed in the RG map to service the read. We use the locality-aware request distribution (LARD) technique [14] to select a read replica—LARD further partitions keys across replicas, in effect aggregating their physical caches.

4.3 Recovery

If a brick fails, all replicas on it become unavailable. Rather than making these partitions unavailable, we remove the failed brick from all replica groups and allow operations to continue on the surviving replicas. When the failed brick recovers (or an alternative brick is selected to replace it), it must "catch up" to all of the operations it missed. In many RDBMS's and file systems, recovery is a complex process that involves replaying logs, but in our system we use properties of clusters and our DDS design for vast simplifications.

Firstly, we allow our hash table to "say no"—bricks may return a failure for an operation, such as when a two-phase commit cannot obtain locks on all bricks (e.g., if two `puts()` to the same key are simultaneously issued), or when replica group memberships change during an operation. The freedom to say no greatly simplifies system logic, since we don't worry about correctly handling operations in these rare situations. Instead, we rely on the DDS library (or, ultimately, the service and perhaps even the WAN client) to retry the operation. Secondly, we don't allow any operation to finish unless all participating components agree on the metadata maps. If any component has an out-of-date map, operations fail until the maps are reconciled.

We make our partitions relatively small (~100MB), which means that we can transfer an entire partition over a fast system-area network (typically 100 Mb/s to 1 Gb/s) within 1 to 10 seconds. Thus, during recovery, we can incrementally copy entire partitions to the recovering node, obviating the need for the undo and redo logs that are typically maintained by databases for recovery. When a node initiates recovery, it grabs a write lease on one replica group member from the partition that it is joining; this write lease means that all state-changing operations on that partition will start to fail. Next, the recovering node copies the entire replica over the network. Then, it sends updates to the RG map to all other replicas in the group, which means that DDS libraries will start to lazily receive this update. Finally, it releases the write lock, which means that the previously failed operations will succeed on retry. The recovery of the partition is now complete, and the recovering node can begin recovery of other partitions as necessary.

There is an interesting choice of the rate at which partitions are transferred over the network during recovery. If this rate is fast, then the involved bricks will suffer a loss in read throughput during the recovery. If this rate is slow, then the bricks won't lose throughput, but the partition's mean time to recovery will increase. We chose to recover as quickly as possible, since in a large cluster only a small fraction of the total throughput of the cluster will be affected by the recovery.

A similar technique is used for DP map split and merge operations, except that all replicas must be modified and both the RG and DP maps are updated at the end of the operation.

4.3.1 Convergence of Recovery

A challenge for fault-tolerant systems is to remain consistent in the face of repeated failures; our recovery scheme described above has this property. In steady state operation, all replicas in a group are kept perfectly consistent. During recovery, state changing operations fail (but only on the recovering partition), implying that surviving replicas remain consistent and recovering nodes have a stable image from which to recover. We also ensure that a recovering node only joins the replica group after it has successfully copied over the entire partition's data but before it release its write lease. A remaining window of vulnerability in the system is if recovery takes longer than the write lease; if this seems imminent, the recovering node could aggressively renew its write lease, but we have not currently implemented this behavior.

If a recovering node crashes during recovery, its write lease will expire and the system will continue as normal. If the replica on which the lease was grabbed crashes, the recovering node must reinitiate recovery with another surviving member of the replica group. If all members of a replica group crash, data will be lost, as mentioned in Section 3.1.

4.4 Asynchrony

All components of the distributed hash table are built using an asynchronous, event-driven programming style. Each hash table layer is designed so that only a single thread ever executes in it at a time. This greatly simplified implementation by eliminating the need for data locks, and race conditions due to threads. Hash table layers are separated by FIFO queues, into which I/O completion events and I/O requests are placed. The FIFO discipline of these queues ensures fairness across requests, and the queues act as natural buffers that absorb bursts that exceed the system's throughput capacity.

All interfaces in the system (including the DDS library APIs) are split-phase and asynchronous. This means that a hash table `get()` doesn't block, but rather immediately returns with an identifier

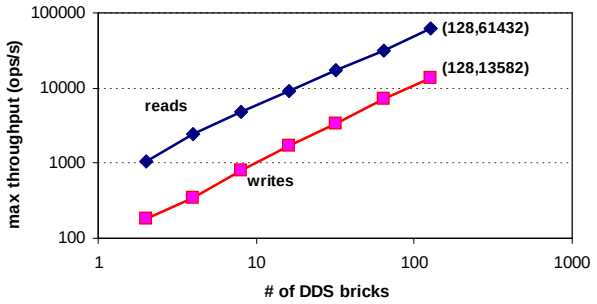


Figure 4: **Throughput scalability:** this benchmark shows the linear scaling of throughput as a function of the number of bricks serving in a distributed hash table; note that both axis have logarithmic scales. As we added more bricks to the DDS, we increased the number of clients using the DDS until throughput saturated.

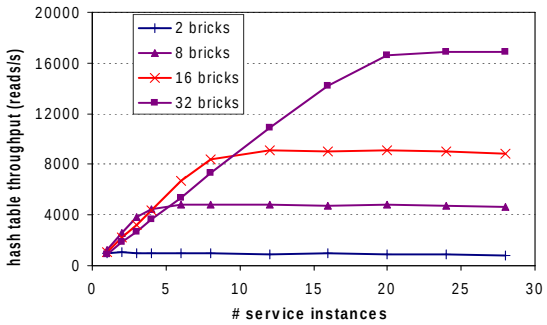


Figure 5: **Graceful degradation of reads:** this graph demonstrates that the read throughput from a distributed hash table remains constant even if the offered load exceeds the capacity of the hash table.

that can be matched up with a completion event that is delivered to a caller-specified upcall handler. This upcall handler can be application code, or it can be a queue that is polled or blocked upon.

5 Performance

In this section, we present performance benchmarks of the distributed hash table implementation that were gathered on a cluster of 28 2-way SMPs and 38 4-way SMPs (a total of 208 500 MHz Pentium CPUs). Each 2-way SMP has 500 MB of RAM, and each 4-way SMP has 1 GB. All are connected with either 100 Mb/s switched Ethernet (2-way SMPs) or 1 Gb/s switched Ethernet (4-way SMPs). The benchmarks are run using Sun's JDK 1.1.7v3, using the OpenJIT 1.1.7 JIT compiler and "green" (user-level) threads on top of Linux v2.2.5.

When running our benchmarks, we evenly spread hash table bricks amongst 4-way and 2-way SMPs, running at most one brick node per CPU in the cluster. Thus, 4-way SMPs would have at most 4 brick processes running on them, while 2-way SMPs would have at most 2. We also made use of these cluster nodes as load generators; because of this, we were only able to gather performance numbers to a maximum of a 128 brick distributed hash table, as we needed the remaining 80 CPUs to generate enough load to saturate such a large table.

5.1 In-Core Benchmarks

Our first set of benchmarks tested the in-core performance of the distributed hash table. By limiting the working set of keys that we requested to a size that fits in the aggregate physical memory of the bricks, this set of benchmarks investigates the overhead and throughput of the distributed hash table code independently of disk performance.

5.1.1 Throughput Scalability

This benchmark demonstrates that hash table throughput scales linearly with the number of bricks. The benchmark consists of several services that each maintain a pipeline of 100 operations (either `gets()` or `puts()`) to a single distributed hash table. We varied the number of bricks in the hash table; for each configuration, we slowly increased the number of services and measured the completion throughput flowing from the bricks. All configurations had 2 replicas per replica group, and each benchmark iteration consisted of reads or writes of 150-byte values. The benchmark was closed-loop: a new operation was immediately issued with a random key for each completed operation.

Figure 4 shows the maximum throughput sustained by the distributed hash table as a function of the number of bricks. Throughput scales linearly up to 128 bricks; we didn't have enough processors to scale the benchmark further. The read throughput achieved with 128 bricks is 61,432 reads per second (5.3 billion per day), and the write throughput with 128 bricks is 13,582 writes per second (1.2 billion per day); this performance is adequate to serve the hit rates of most popular web sites on the Internet.

5.1.2 Graceful Degradation for Reads

Bursts of traffic are a common phenomenon for all Internet services. If a traffic burst exceeds the service's capacity, the service should have the property of "graceful degradation": the throughput of the service should remain constant, with the excess traffic either being rejected or absorbed in buffers and served with higher latency. Figure 5 shows the throughput of a distributed hash table as a function of the number of simultaneous read requests issued to it; each service instance has a closed-loop pipeline of 100 operations. Each line on the graph represents a different number of bricks serving the

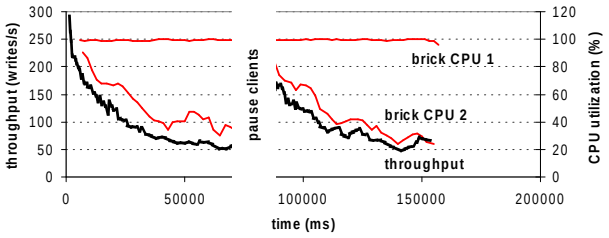


Figure 6: **Write imbalance leading to ungraceful degradation:** the bottom curve shows the throughput of a two-brick partition under overload, and the top two curves show the CPU utilization of those bricks. One brick is saturated, the other becomes only 30% busy.

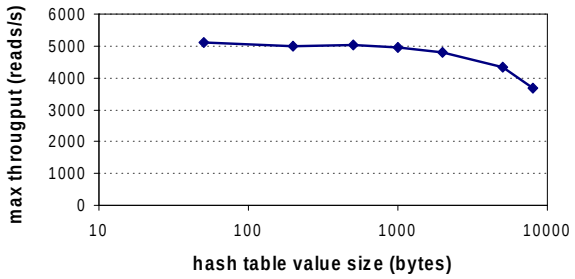


Figure 7: **Throughput vs. read size** the X axis shows the size of values read from the hash table, and the Y axis shows the maximum throughput sustained by an 8 brick hash table serving these values.

hash table. Each configuration is seen to eventually reach a maximum throughput as its bricks saturate. This maximum throughput is successfully sustained even as additional traffic is offered. The overload traffic is absorbed in the FIFO event queues of the bricks; all tasks are processed, but they experience higher latency as the queues drain from the burst.

5.1.3 Ungraceful Degradation for Writes

An unfortunate performance anomaly emerged when benchmarking `put()` throughput. As the offered load approached the maximum capacity of the hash table bricks, the total write throughput suddenly began to drop. On closer examination, we discovered that most of the bricks in the hash table were unloaded, but one brick in the hash table was completely saturated and had become the bottleneck in the closed-loop benchmark.

Figure 6 illustrates this imbalance. To generate it, we issued `puts()` to a hash table with a single partition and two replicas in its replica group. Each `put()` operation caused a two-phase commit across both replicas, and thus each replica saw the same set of network messages and performed the same computation (but perhaps in slightly different orders). We expected both replicas to perform identically, but instead one replica became more and more idle, and the throughput of the hash table dropped to match the CPU utilization of this idle replica.

Investigation showed that the busy replica was spending a significant amount of time in garbage collection. As more live objects populated that replica's heap, more time needed to be spent garbage collecting to reclaim a fixed amount of heap space, as more objects would be examined before a free object was discovered. Random fluctuations in arrival rates and garbage collection caused one replica to spend more time garbage collecting than the other. This replica became the system bottleneck, and more operations piled up in its queues, further amplifying this imbalance. Write traffic particularly exacerbated the situation, as objects created by the "prepare" phase must wait for at least one network round-trip time before a commit or abort command in the second phase is received. The number of live objects in each bricks' heap is thus proportional to the bandwidth-delay product of hash table `put()` operations. For read traffic, there is only one phase, and thus objects can be garbage collected immediately after read requests are satisfied.

We experimented with many JDKs, but consistently saw this effect. Some JDKs (such as JDK 1.2.2 on Linux 2.2.5) developed this imbalance for read traffic as well as write traffic. This sort of performance imbalance is fundamental to any system that doesn't perform admission control; if the task arrival rate temporarily exceeds the system's ability to handle them, then tasks will begin to pile up in the system. Because systems have finite resources, this inevitably causes performance degradation (thrashing). In our system, this degradation first materialized due to garbage collection. In other systems, this might happen due to virtual memory thrashing, to pick an example. We are currently exploring using admission control (at either the bricks or the hash table libraries) or early discard from bricks' queues to keep the bricks within their operational range, ameliorating this imbalance.

5.1.4 Throughput Bottlenecks

In figure 7, we varied the size of elements that we read out of an 8 brick hash table. Throughput was flat from 50 bytes through 1000 bytes, but then began to degrade. From this we deduced that per-operation overhead (such as object creation, garbage collection, and system call overhead) saturated the bricks' CPUs for elements smaller than 1000 bytes, and per-byte overhead (byte array copies, either in the TCP stack or in the JVM) saturated the bricks' CPUs for elements greater than 1000 bytes. At 8000 bytes, the throughput in and out of each 2-way SMP (running 2 bricks) was 60 Mb/s. For larger sized

hash table values, the 100 Mb/s switched network became the throughput bottleneck.

5.2 Out-of-core Benchmarks

Our next set of benchmarks tested performance for workloads that do not fit in the aggregate physical memory of the bricks. These benchmarks stress the single-node hash table's disk interaction, as well as the performance of the distributed hash table.

5.2.1 A Terabyte DDS

To test how well the distributed hash table scales in terms of data capacity, we populated a hash table with 1.28 terabytes of 8KB data elements. To do this, we created a table with 512 partitions in its DP map, but with only 1 replica per replica group (i.e., the table would not withstand node failures). We spread the 512 partitions across 128 brick nodes, and ran 2 bricks per node in the cluster. Each brick stored its data on a dedicated 12GB disk (all cluster nodes have 2 of these disks). The bricks each used 10GB worth of disk capacity, resulting in 1.28TB of data stored in the table.

To populate the 1.28TB hash table, we designed bulk loaders that generated writes to keys in an order that was carefully chosen to result in sequential disk writes. These bulk loaders understood the partitioning in the DP map and implementation details about the single-node tables' hash functions (which map keys to disk blocks). Using these loaders, it took 130 minutes to fill the table with 1.28 terabytes of data, achieving a total write throughput of 22,015 operations/s, or 1.4 MB/s per disk.

Comparatively, the in-core throughput benchmark presented in Section 5.1.1 obtained 13,582 operations/s for a 128 brick table, but that benchmark was configured with 2 replicas per replica group. Eliminating this replication would double the throughput of the in-core benchmark, resulting in a 27,164 operations/s. The bulk loading of the 1.28TB hash table was therefore only marginally slower in terms of the throughput sustained by each replica than the in-core benchmarks, which means that disk throughput was not the bottleneck.

5.2.2 Random Write and Read Throughput

However, we believe it is unrealistic and undesirable for hash table clients to have knowledge of the DP map and single-node tables' hash functions. We ran a second set of throughput benchmarks on another 1.28TB hash table, but populated it with random keys. With this workload, the table took 319 minutes to populate, resulting in a total write throughput of 8,985 operations/s, or 0.57 MB/s per disk. We similarly sustained a read throughput of 14,459 operations/s, or 0.93 MB/s per disk.[3]

This throughput is substantially lower than the throughput obtained during the in-core benchmarks because the random workload generated results in random read and write traffic to each disk. In fact, for this random workload, every `read()` issued to the distributed hash table results in a request for a random disk block from a disk. All disk traffic is seek dominated, and disk seeks become the overall bottleneck of the system.

We expect that there will be significant locality in DDS requests generated by Internet services, and given workloads with high locality, the DDS should perform nearly as well as the in-core benchmark results. However, it might be possible to significantly improve the write performance of traffic with little locality by using disk layout techniques similar to those of log-structured file systems [29]; we have not explored this possibility as of yet.

5.3 Availability and Recovery

To demonstrate availability in the face of node failures and the ability for the bricks to recover after a failure, we repeated the read benchmark with a hash table of 150 byte elements. The table was configured with a single 100MB partition and three replicas in that partition's replica group. Figure 8 shows the throughput of the hash table over time as we induced a fault in one of the replica bricks and later initiated its recovery. During recovery, the rate at which the recovered partition is copied was 12 MB/s, which is maximum sequential write bandwidth we could obtain from the bricks' disks.

At point (1), all three bricks were operational and the throughput sustained by the hash table was 450 operations per second. At point (2), one of the three bricks was killed. Performance immediately dropped to 300 operations per second, two-thirds of the original capacity. Fault detection was immediate: client libraries experienced broken transport connections that could not be reestablished. The performance overhead of the replica group map updates could not be observed. At point (3), recovery was initiated, and recovery completed at point (4). Between points (3) and (4), there was no noticeable performance overhead of recovery; this is because there was ample excess bandwidth on the network, and the CPU overhead of transferring the partition during recovery was negligible. It should be noted that between points (3) and (4), the recov-

[3] Write throughput is less than read throughput because a hash bucket must be read before it can written, in case there is already data stored in that bucket that must be preserved. There is therefore an additional read for every write, nearly halving the effective throughput for DDS writes.

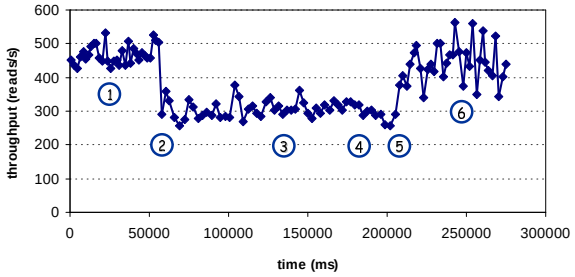


Figure 8: **Availability and Recovery:** this benchmark shows the read throughput of a 3-brick hash table as a deliberate single-node fault is induced, and afterwards as recovery is performed.

ering partition is not available for writes, because of the write lease grabbed during recovery. This partition is available for reads, however.

After recovery completed, performance briefly dropped at point (5). This degradation is due to the buffer cache warming on the recovered node. Once the cache became warm, performance resumed to the original 450 operations/s at point (6). An interesting anomaly at point (6) is the presence of noticeable oscillations in throughput; these were traced to garbage collection triggered by the "extra" activity of recovery. When we repeated our measurements, we would occasionally see this oscillation at other times besides immediately post-recovery. This sort of performance unpredictability due to garbage collection seems to be a pervasive problem; a better garbage collector or admission control might ameliorate this, but we haven't yet explored this.

6 Example Services

We have implemented a number of interesting services using our distributed hash table. The services' implementation was greatly simplified by using the DDS, and they trivially scaled by adding more service instances. An aspect of scalability not covered by using the hash table was the routing and load balancing of WAN client requests across service instances, but this is beyond the scope of this work.

Sanctio: Sanctio is an instant messaging gateway that provides protocol translation between popular instant messaging protocols (such as Mirabilis' ICQ and AOL's AIM), conventional email, and voice messaging over cellular telephones. Sanctio is a middleman between these protocols, routing and translating messages between the networks. In addition to protocol translation, Sanctio also can transform the message content. We have built a "web scraper" that allows us to compose AltaVista's BabelFish natural language translation service with Sanctio. We can thus perform language translation (e.g., English to French) as well as protocol translation; a Spanish speaking ICQ user can send a message to an English speaking AIM user, with Sanctio providing both language and protocol translation.

A user may be reached on a number of different addresses, one for each of the networks that Sanctio can communicate with. The Sanctio service must therefore keep a large table of bindings between users and their current transport addresses on these networks; we used the distributed hash table for this purpose. The expected workload on the DDS includes significant write traffic generated when users change networks or log in and out of a network. The data in the table must be kept consistent, otherwise messages will be routed to the wrong address.

Sanctio took 1 person-month to develop, most which was spent authoring the protocol translation code. The code that interacts with the distributed hash table took less than a day to write.

Web server: we have implemented a scalable web server using the distributed hash table. The server speaks HTTP to web clients, hashes requested URLs into 64 bit keys, and requests those keys from the hash table. The server takes advantage of the event-driven, queue-centric programming style to introduce CGI-like behavior by interposing on the URL resolution path. This web server was written in 900 lines of Java, 750 of which deals with HTTP parsing and URL resolution, and only 50 of which deals with interacting with the hash table DDS.

Others: We have built many other services as part of the Ninja project[4]. The "Parallelisms" service recommends related sites to user-specified URLs by looking up ontological entries in an inversion of the Yahoo web directory. We built a collaborative filtering engine for a digital music jukebox service [16]; this engine stores users' music preferences in a distributed hash table. We have also implemented a private key store and a composable user preference service, both of which use the distributed hash table for persistent state management.

7 Discussion

Our experience with the distributed hash table implementation has taught us many lessons about using it as a storage platform for scalable services. The hash table was a resounding success in simplifying the construction of interesting services, and these services inherited the scalability, availability, and data consistency of the hash table. Exploiting properties of clusters also proved to be remarkably useful. In our experience, most of the assumptions that we made regarding properties of a clusters and component failures (specifically the fail-stop behav-

[4] `http://ninja.cs.berkeley.edu/`

ior of our software and the probabilistic lack of network partitions in the cluster) were valid in practice.

One of our assumptions was initially problematic: we observed a case where there was a systematic failure of all replica group members inside a single replica group. This failure was caused by a software bug that enabled service instances to deterministically crash remote bricks by inducing a null pointer exception in the JVM. After fixing the associated bug in the brick, this situation never again arose. However, it serves as a reminder that systematic software bugs can in practice bring down the entire cluster at once. Careful software engineering and a good quality assurance cycle can help to ameliorate this failure mode, but we believe that this issue is fundamental to all systems that promise both availability and consistency.

As we scaled our distributed hash table, we noticed scaling bottlenecks that weren't associated with our own software. At 128 bricks, we approached the point at which the 100 Mb/s Ethernet switches would saturate; upgrading to 1 Gb/s switches throughout the cluster would delay this saturation. We also noticed that the combination of our JVM's user-level threads and the Linux kernel began to induced poor scaling behavior as each node in the cluster opened up a reliable TCP connection to all other nodes in the cluster. The brick processes began to saturate due to a flood of signals from the kernel to the user-level thread scheduler associated with TCP connections with data waiting to be read.

7.1 Java as a Service Platform

We found that Java was an adequate platform from which to build a scalable, high performance subsystem. However, we ran into a number of serious issues with the Java language and runtime. The garbage collector of all JVMs that we experimented with inevitably became the performance bottleneck of the bricks and also a source of throughput and latency variation. Whenever the garbage collector became active, it had a serious impact on all other system activity, and unfortunately, current JVMs do not provide adequate interfaces to allow systems to control garbage collection behavior.

The type safety and array bounds checking features of Java vastly accelerated our software engineering process, and helped us to write stable, clean code. However, these features got in the way of code efficiency, especially when dealing with multiple layers of a system each of which wraps some array of data with layer-specific metadata. We often found ourselves performing copies of regions of byte arrays in order to maintain clean interfaces to data regions, whereas in a C implementation it would be more natural to exploit pointers into `malloc'ed` memory regions to the same effect without needing copies.

Java lacks asynchronous I/O primitives, which necessitated the use of a thread pool at the lowest-layer of the system. This is much more efficient than a thread-per-task system, as the number of threads in our system is equal to the number of outstanding I/O requests rather than the number of tasks. Nonetheless, it introduced performance overhead and scaling problems, since the number of TCP connections per brick increases with the cluster size. We are working on introducing high-throughput asynchronous I/O completion mechanisms into the JVM using the JNI native interface.

7.2 Future Work

We plan on investigating more interesting data-parallel operations on a DDS (such as an iterator, or the Lisp `maplist()` operator). We also plan on building other distributed data structures, including a B-tree and an administrative log. In doing so, we hope to reuse many of the components of the hash table, such as the brick storage layer, the RG map infrastructure, and the two-phase commit code. We would like to explore caching in the DDS libraries (we currently rely on services to build their own application-level caches). We are also exploring adding other single-element operations to the hash table, such as `testandset()`, in order to provide locks and leases to services that may have many service instances competing to write to the same hash table element.

8 Related Work

Litwin et al.'s scalable, distributed data structures (SDDS) such as RP^* [22, 26] helped to motivate our own work. RP^* focuses on algorithmic properties, while we focused on the systems issues of implementing an SDDS that satisfies the concurrency, availability, and incremental scalability needs of Internet services.

Our work has a great deal in common with database research. The problems of partitioning and replicating data across shared-nothing multicomputers has been studied extensively in the distributed and parallel database communities [10, 17, 25]. We use mechanisms such as horizontal partitioning and two-phase commits, but we do not need an SQL parser or a query optimization layer since we have no general-purpose queries in our system.

We also have much in common with distributed and parallel file systems [3, 23, 31, 33]. A DDS presents a higher-level interface than a typical file system, and DDS operations are data-structure specific and atomically affect entire elements. Our research has focused on scalability, availability, and

consistency under high throughput, highly concurrent traffic, which is a different focus than file systems. Our work is most similar to Petal [24], in that a Petal distributed virtual disk can be thought of as a simple hash table with fixed sized elements. Our hash tables have variable sized elements, an additional name space (the set of hash tables), and focus on Internet service workloads and properties as opposed to file system workloads and properties.

The CMU network attached secure disk (NASD) architecture [11] explores variable-sized object interfaces as an abstraction to allow storage subsystems to optimize disk layout. This is similar to our own data structure interface, which is deliberately higher-level than the block or file interfaces of Petal and parallel or distributed file systems.

Distributed object stores [13] attempt to transparently adding persistence to distributed object systems. The persistence of (typed) objects is typically determined by reachability through the transitive closure of object references, and the removal of objects is handled by garbage collection. A DDS has no notion of pointers or object typing, and applications must explicitly use API operations to store and retrieve elements from a DDS. Distributed object stores are often built with the wide-area in mind, and thus do not focus on the scalability, availability, and high throughput requirements of cluster-based Internet services.

Many projects have explored the use of clusters of workstations as a general-purpose platform for building Internet services [1, 4, 15]. To date, these platforms rely on file systems or databases for persistent state management; our DDS's are meant to augment such platforms with a state management platform that is better suited to the needs of Internet services. The Porcupine project [30] includes a storage platform built specifically for the needs of a cluster-based scalable mail server, but they are attempting to generalize their storage platform for arbitrary service construction.

There have been many projects that expolored wide-area replicated, distributed services [9, 27]. Unlike clusters, wide-area systems must deal with heterogeneity, network partitions, untrusted peers, high latency and low throughput networks, and multiple administrative domains. Because of these differences, wide-area distributed systems tend to have relaxed consistency semantics and low update rates. However, if designed correctly, they can scale up enormously.

9 Conclusions

This paper presents a new persistent data management layer that enhances the ability of clusters to support Internet services. This self-managing layer, called a distributed data structure (DDS), fills in an important gap in current cluster platforms by providing a data storage platform specifically tuned for services' workloads and for the cluster environment.

This paper focused on the design and implementation of a distributed hash table DDS, empirically demonstrating that it has many properties necessary for Internet services (incremental scaling of throughput and data capacity, fault tolerance and high availability, high concurrency, and consistency and durability of data). These properties were achieved by carefully designing the partitioning, replication, and recovery techniques in the hash table implementation to exploit features of cluster environments (such as a low-latency network with a lack of network partitions). By doing so, we have "right-sized" the DDS to the problem of persistent data management for Internet services.

The hash table DDS simplifies Internet service construction by decoupling service-specific logic from the complexities of persistent state management, and by allowing services to inherit the necessary service properties from the DDS rather than having to implement the properties themselves.

Acknowledgements

We are very grateful to Eric Anderson, Rob von Behren, Nikita Borisov, Mike Chen, Armando Fox, Jim Gray, Ramki Gummadi, Drew Roselli, Geoff Voelker, the anonymous referees, and our shepherd Bill Weihl for their very helpful suggestions that greatly improved the quality of this paper. We would also like to thank Eric Fraser, Phil Buonadonna, and Brent Chun for their help in giving us access to the Berkeley Millennium cluster for our performance benchmarks.

References

[1] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-Time Multimedia Transcoding. In *Proceedings of ACM SIGCOMM '98*, pages 178–189, Oct 1998.

[2] T. E. Anderson, D. E. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 12(1):54–64, Feb 1995.

[3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec 1995.

[4] D. Andresen, T. Yang, O. Egecioglu, O. H. Ibarra, and T. R. Smith. Scalability Issues for High Performance Digital Libraries on the World Wide Web. In *Proceedings of IEEE ADL '96*, Washington D.C., May 1996.

[5] G. Banga, J. C. Mogul, and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, Jun 1999.

[6] BEA Systems. BEA WebLogic Application Servers. `http://www.bea.com/products/weblogic/`.

[7] T. Brisco. RFC 1764: DNS Support for Load Balancing, Apr 1995.

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, Jan 1996.

[9] A.D. Birrell et al. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):3–23, Feb 1984.

[10] D. DeWitt et al. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), Mar 1990.

[11] G. A. Gibson et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *ASPLOS-VIII*, San Jose, California, 1998.

[12] J. H. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), Feb 1988.

[13] P. Ferreira et al. PerDiS: Design, Implementation, and Use of a PERsistent DIstributed Store. In *Recent Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, chapter 18, pages 427–452. Springer-Verlag, Feb 2000.

[14] V. S. Pai et al. Locality-Aware Request Distribution in Cluster-Based Network Servers. In *ASPLOS-VIII*, San Jose, CA, Oct 1998.

[15] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, Oct 1997.

[16] I. Goldberg, S. D. Gribble, D. Wagner, and E. A. Brewer. The Ninja Jukebox. In *The 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct 1999.

[17] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *ACM SIGMOD Conference on the Management of Data*, Atlantic City, NJ, May 1990.

[18] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of VLDB*, Cannes, France, September 1981.

[19] S. D. Gribble and E. A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS 97)*, Monterey, CA, Dec 1997.

[20] J. C. Hu, I. Pyarali, and D. C. Schmidt. Applying the Proactor Pattern to High-Performance Web Servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, Oct 1998.

[21] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig. High-Performance Web Site Design Techniques. *IEEE Internet Computing*, 4(2), Mar 2000.

[22] J. S. Karlsson, W. Litwin, and T. Risch. LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers. In *Proceedings of the 5th International Conference on Extending Database Technology*, pages 573–591, Avignon, France, Mar 1996.

[23] O. Krieger and M. Stumm. HFS: A Flexible File System for Large-Scale Multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, Jun 1993.

[24] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *ASPLOS-VII*, Cambridge, MA, 1996.

[25] B. G. Lindsay. A Retrospective of R*: A Distributed Database Management System. *Proceedings of the IEEE*, 75(5):668–673, May 1987.

[26] W. Litwin, M. Neimat, and D. A. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 342–353, Santiago, Chile, 1994.

[27] P. V. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *ACM SIGCOMM Computer Communication Review*, 1988.

[28] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, Jun 1999.

[29] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[30] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service. In *Proceedings of the 17th Symposium on Operating System Principles*, Kiawah Island, SC, Dec 1999.

[31] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX 1985 Summer Conference*, El Cerrito, CA, Jun 1985.

[32] J. Song, E. Levy, A. Iyengar, and D. Dias. Design Alternatives for Scalable Web Server Accelerators. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000)*, Austin, TX, Apr 2000.

[33] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, Oct 1997.

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...$5.00.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

a gossip based distributed failure detection and membership protocol. Dynamo is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution.

In the past year, Dynamo has been the underlying storage technology for a number of the core services in Amazon's e-commerce platform. It was able to scale to extreme peak loads efficiently without any downtime during the busy holiday shopping season. For example, the service that maintains shopping cart (Shopping Cart Service) served tens of millions requests that resulted in well over 3 million checkouts in a single day and the service that manages session state handled hundreds of thousands of concurrently active sessions.

The main contribution of this work for the research community is the evaluation of how different techniques can be combined to provide a single highly-available system. It demonstrates that an eventually-consistent storage system can be used in production with demanding applications. It also provides insight into the tuning of these techniques to meet the requirements of production systems with very strict performance demands.

The paper is structured as follows. Section 2 presents the background and Section 3 presents the related work. Section 4 presents the system design and Section 5 describes the implementation. Section 6 details the experiences and insights gained by running Dynamo in production and Section 7 concludes the paper. There are a number of places in this paper where additional information may have been appropriate but where protecting Amazon's business interests require us to reduce some level of detail. For this reason, the intra- and inter-datacenter latencies in section 6, the absolute request rates in section 6.2 and outage lengths and workloads in section 6.3 are provided through aggregate measures instead of absolute details.

2. BACKGROUND

Amazon's e-commerce platform is composed of hundreds of services that work in concert to deliver functionality ranging from recommendations to order fulfillment to fraud detection. Each service is exposed through a well defined interface and is accessible over the network. These services are hosted in an infrastructure that consists of tens of thousands of servers located across many data centers world-wide. Some of these services are stateless (i.e., services which aggregate responses from other services) and some are stateful (i.e., a service that generates its response by executing business logic on its state stored in persistent store).

Traditionally production systems store their state in relational databases. For many of the more common usage patterns of state persistence, however, a relational database is a solution that is far from ideal. Most of these services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS. This excess functionality requires expensive hardware and highly skilled personnel for its operation, making it a very inefficient solution. In addition, the available replication technologies are limited and typically choose consistency over availability. Although many advances have been made in the recent years, it is still not easy to scale-out databases or use smart partitioning schemes for load balancing.

This paper describes Dynamo, a highly available data storage technology that addresses the needs of these important classes of services. Dynamo has a simple key/value interface, is highly available with a clearly defined consistency window, is efficient in its resource usage, and has a simple scale out scheme to address growth in data set size or request rates. Each service that uses Dynamo runs its own Dynamo instances.

2.1 System Assumptions and Requirements

The storage system for this class of services has the following requirements:

Query Model: simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (i.e., blobs) identified by unique keys. No operations span multiple data items and there is no need for relational schema. This requirement is based on the observation that a significant portion of Amazon's services can work with this simple query model and do not need any relational schema. Dynamo targets applications that need to store objects that are relatively small (usually less than 1 MB).

ACID Properties: ACID (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. This has been widely acknowledged by both the industry and academia [5]. Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability. Dynamo does not provide any isolation guarantees and permits only single key updates.

Efficiency: The system needs to function on a commodity hardware infrastructure. In Amazon's platform, services have stringent latency requirements which are in general measured at the 99.9th percentile of the distribution. Given that state access plays a crucial role in service operation the storage system must be capable of meeting such stringent SLAs (see Section 2.2 below). Services must be able to configure Dynamo such that they consistently achieve their latency and throughput requirements. The tradeoffs are in performance, cost efficiency, availability, and durability guarantees.

Other Assumptions: Dynamo is used only by Amazon's internal services. Its operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization. Moreover, since each service uses its distinct instance of Dynamo, its initial design targets a scale of up to hundreds of storage hosts. We will discuss the scalability limitations of Dynamo and possible scalability related extensions in later sections.

2.2 Service Level Agreements (SLA)

To guarantee that the application can deliver its functionality in a bounded time, each and every dependency in the platform needs to deliver its functionality with even tighter bounds. Clients and services engage in a Service Level Agreement (SLA), a formally negotiated contract where a client and a service agree on several system-related characteristics, which most prominently include the client's expected request rate distribution for a particular API and the expected service latency under those conditions. An example of a simple SLA is a service guaranteeing that it will

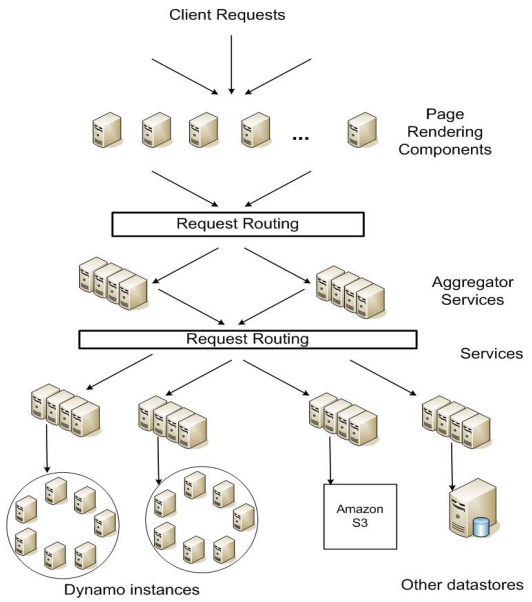


Figure 1: Service-oriented architecture of Amazon's platform

provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.

In Amazon's decentralized service oriented infrastructure, SLAs play an important role. For example a page request to one of the e-commerce sites typically requires the rendering engine to construct its response by sending requests to over 150 services. These services often have multiple dependencies, which frequently are other services, and as such it is not uncommon for the call graph of an application to have more than one level. To ensure that the page rendering engine can maintain a clear bound on page delivery each service within the call chain must obey its performance contract.

Figure 1 shows an abstract view of the architecture of Amazon's platform, where dynamic web content is generated by page rendering components which in turn query many other services. A service can use different data stores to manage its state and these data stores are only accessible within its service boundaries. Some services act as aggregators by using several other services to produce a composite response. Typically, the aggregator services are stateless, although they use extensive caching.

A common approach in the industry for forming a performance oriented SLA is to describe it using average, median and expected variance. At Amazon we have found that these metrics are not good enough if the goal is to build a system where **all** customers have a good experience, rather than just the majority. For example if extensive personalization techniques are used then customers with longer histories require more processing which impacts performance at the high-end of the distribution. An SLA stated in terms of mean or median response times will not address the performance of this important customer segment. To address this issue, at Amazon, SLAs are expressed and measured at the 99.9th percentile of the distribution. The choice for 99.9% over an even higher percentile has been made based on a cost-benefit analysis which demonstrated a significant increase in cost to improve performance that much. Experiences with Amazon's production systems have shown that this approach provides a better overall experience compared to those systems that meet SLAs defined based on the mean or median.

In this paper there are many references to this 99.9th percentile of distributions, which reflects Amazon engineers' relentless focus on performance from the perspective of the customers' experience. Many papers report on averages, so these are included where it makes sense for comparison purposes. Nevertheless, Amazon's engineering and optimization efforts are not focused on averages. Several techniques, such as the load balanced selection of write coordinators, are purely targeted at controlling performance at the 99.9th percentile.

Storage systems often play an important role in establishing a service's SLA, especially if the business logic is relatively lightweight, as is the case for many Amazon services. State management then becomes the main component of a service's SLA. One of the main design considerations for Dynamo is to give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost-effectiveness.

2.3 Design Considerations

Data replication algorithms used in commercial systems traditionally perform synchronous replica coordination in order to provide a strongly consistent data access interface. To achieve this level of consistency, these algorithms are forced to tradeoff the availability of the data under certain failure scenarios. For instance, rather than dealing with the uncertainty of the correctness of an answer, the data is made unavailable until it is absolutely certain that it is correct. From the very early replicated database works, it is well known that when dealing with the possibility of network failures, strong consistency and high data availability cannot be achieved simultaneously [2, 11]. As such systems and applications need to be aware which properties can be achieved under which conditions.

For systems prone to server and network failures, availability can be increased by using optimistic replication techniques, where changes are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated. The challenge with this approach is that it can lead to conflicting changes which must be detected and resolved. This process of conflict resolution introduces two problems: when to resolve them and who resolves them. Dynamo is designed to be an eventually consistent data store; that is all updates reach all replicas eventually.

An important design consideration is to decide *when* to perform the process of resolving update conflicts, i.e., whether conflicts should be resolved during reads or writes. Many traditional data stores execute conflict resolution during writes and keep the read complexity simple [7]. In such systems, writes may be rejected if the data store cannot reach all (or a majority of) the replicas at a given time. On the other hand, Dynamo targets the design space of an "always writeable" data store (i.e., a data store that is highly available for writes). For a number of Amazon services, rejecting customer updates could result in a poor customer experience. For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures. This requirement forces us to push the complexity of conflict resolution to the reads in order to ensure that writes are never rejected.

The next design choice is *who* performs the process of conflict resolution. This can be done by the data store or the application. If conflict resolution is done by the data store, its choices are rather limited. In such cases, the data store can only use simple policies, such as "last write wins" [22], to resolve conflicting updates. On the other hand, since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart. Despite this flexibility, some application developers may not want to write their own conflict resolution mechanisms and choose to push it down to the data store, which in turn chooses a simple policy such as "last write wins".

Other key principles embraced in the design are:

Incremental scalability: Dynamo should be able to scale out one storage host (henceforth, referred to as "*node"*) at a time, with minimal impact on both operators of the system and the system itself.

Symmetry: Every node in Dynamo should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities. In our experience, symmetry simplifies the process of system provisioning and maintenance.

Decentralization: An extension of symmetry, the design should favor decentralized peer-to-peer techniques over centralized control. In the past, centralized control has resulted in outages and the goal is to avoid it as much as possible. This leads to a simpler, more scalable, and more available system.

Heterogeneity: The system needs to be able to exploit heterogeneity in the infrastructure it runs on. e.g. the work distribution must be proportional to the capabilities of the individual servers. This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.

3. RELATED WORK

3.1 Peer to Peer Systems

There are several peer-to-peer (P2P) systems that have looked at the problem of data storage and distribution. The first generation of P2P systems, such as Freenet and Gnutella[1], were predominantly used as file sharing systems. These were examples of unstructured P2P networks where the overlay links between peers were established arbitrarily. In these networks, a search query is usually flooded through the network to find as many peers as possible that share the data. P2P systems evolved to the next generation into what is widely known as structured P2P networks. These networks employ a globally consistent protocol to ensure that any node can efficiently route a search query to some peer that has the desired data. Systems like Pastry [16] and Chord [20] use routing mechanisms to ensure that queries can be answered within a bounded number of hops. To reduce the additional latency introduced by multi-hop routing, some P2P systems (e.g., [14]) employ O(1) routing where each peer maintains enough routing information locally so that it can route requests (to access a data item) to the appropriate peer within a constant number of hops.

[1] http://freenetproject.org/, http://www.gnutella.org

Various storage systems, such as Oceanstore [9] and PAST [17] were built on top of these routing overlays. Oceanstore provides a global, transactional, persistent storage service that supports serialized updates on widely replicated data. To allow for concurrent updates while avoiding many of the problems inherent with wide-area locking, it uses an update model based on conflict resolution. Conflict resolution was introduced in [21] to reduce the number of transaction aborts. Oceanstore resolves conflicts by processing a series of updates, choosing a total order among them, and then applying them atomically in that order. It is built for an environment where the data is replicated on an untrusted infrastructure. By comparison, PAST provides a simple abstraction layer on top of Pastry for persistent and immutable objects. It assumes that the application can build the necessary storage semantics (such as mutable files) on top of it.

3.2 Distributed File Systems and Databases

Distributing data for performance, availability and durability has been widely studied in the file system and database systems community. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus [15] and Coda [19] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. The Farsite system [1] is a distributed file system that does not use any centralized server like NFS. Farsite achieves high availability and scalability using replication. The Google File System [6] is another distributed file system built for hosting the state of Google's internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunkservers. Bayou is a distributed relational database system that allows disconnected operations and provides eventual data consistency [21].

Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level resolution. All of them, however, guarantee eventual consistency. Similar to these systems, Dynamo allows read and write operations to continue even during network partitions and resolves updated conflicts using different conflict resolution mechanisms. Distributed block storage systems like FAB [18] split large size objects into smaller blocks and stores each block in a highly available manner. In comparison to these systems, a key-value store is more suitable in this case because: (a) it is intended to store relatively small objects (size < 1M) and (b) key-value stores are easier to configure on a per-application basis. Antiquity is a wide-area distributed storage system designed to handle multiple server failures [23]. It uses a secure log to preserve data integrity, replicates each log on multiple servers for durability, and uses Byzantine fault tolerance protocols to ensure data consistency. In contrast to Antiquity, Dynamo does not focus on the problem of data integrity and security and is built for a trusted environment. Bigtable is a distributed storage system for managing structured data. It maintains a sparse, multi-dimensional sorted map and allows applications to access their data using multiple attributes [2]. Compared to Bigtable, Dynamo targets applications that require only key/value access with primary focus on high availability where updates are not rejected even in the wake of network partitions or server failures.

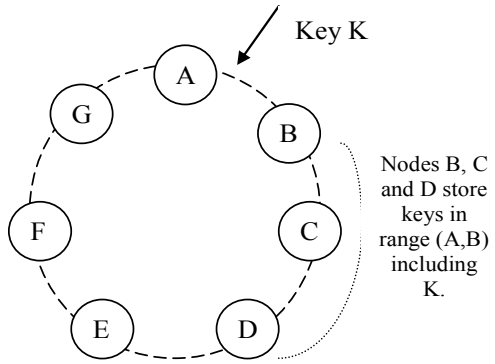


Figure 2: Partitioning and replication of keys in Dynamo ring.

Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency to replicated data. Although strong consistency provides the application writer a convenient programming model, these systems are limited in scalability and availability [7]. These systems are not capable of handling network partitions because they typically provide strong consistency guarantees.

3.3 Discussion

Dynamo differs from the aforementioned decentralized storage systems in terms of its target requirements. First, Dynamo is targeted mainly at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes. This is a crucial requirement for many Amazon applications. Second, as noted earlier, Dynamo is built for an infrastructure within a single administrative domain where all nodes are assumed to be trusted. Third, applications that use Dynamo do not require support for hierarchical namespaces (a norm in many file systems) or complex relational schema (supported by traditional databases). Fourth, Dynamo is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds. To meet these stringent latency requirements, it was imperative for us to avoid routing requests through multiple nodes (which is the typical design adopted by several distributed hash table systems such as Chord and Pastry). This is because multi-hop routing increases variability in response times, thereby increasing the latency at higher percentiles. Dynamo can be characterized as a zero-hop DHT, where each node maintains enough routing information locally to route a request to the appropriate node directly.

4. SYSTEM ARCHITECTURE

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is not possible, so this paper focuses on the core distributed systems techniques used in Dynamo: partitioning, replication, versioning, membership, failure handling and scaling.

Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

4.1 System Interface

Dynamo stores objects associated with a key through a simple interface; it exposes two operations: get() and put(). The get(*key*) operation locates the object replicas associated with the *key* in the storage system and returns a single object or a list of objects with conflicting versions along with a *context*. The put(*key, context, object*) operation determines where the replicas of the *object* should be placed based on the associated *key*, and writes the replicas to disk. The *context* encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object. The context information is stored along with the object so that the system can verify the validity of the context object supplied in the put request.

Dynamo treats both the key and the object supplied by the caller as an opaque array of bytes. It applies a MD5 hash on the key to generate a 128-bit identifier, which is used to determine the storage nodes that are responsible for serving the key.

4.2 Partitioning Algorithm

One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes (i.e., storage hosts) in the system. Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing [10], the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position.

Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Dynamo uses a variant of consistent hashing (similar to the one used in [10, 20]): instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of "virtual nodes". A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions (henceforth, "tokens") in the ring. The process of fine-tuning Dynamo's partitioning scheme is discussed in Section 6.

Using virtual nodes has the following advantages:

- If a node becomes unavailable (due to failures or routine maintenance), the load handled by this node is evenly dispersed across the remaining available nodes.
- When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
- The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

4.3 Replication

To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured *"per-instance"*. Each key, *k*, is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its Nth predecessor. In Figure 2, node B replicates the key *k* at nodes C and D in addition to storing it locally. Node D will store the keys that fall in the ranges (A, B], (B, C], and (C, D].

The list of nodes that is responsible for storing a particular key is called the *preference list*. The system is designed, as will be explained in Section 4.8, so that every node in the system can determine which nodes should be in this list for any particular key. To account for node failures, preference list contains more than N nodes. Note that with the use of virtual nodes, it is possible that the first N successor positions for a particular key may be owned by less than N distinct physical nodes (i.e. a node may hold more than one of the first N positions). To address this, the preference list for a key is constructed by skipping positions in the ring to ensure that the list contains only distinct physical nodes.

4.4 Data Versioning

Dynamo provides eventual consistency, which allows for updates to be propagated to all replicas asynchronously. A put() call may return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent get() operation may return an object that does not have the latest updates.. If there are no failures then there is a bound on the update propagation times. However, under certain failure scenarios (e.g., server outages or network partitions), updates may not arrive at all replicas for an extended period of time.

There is a category of applications in Amazon's platform that can tolerate such inconsistencies and can be constructed to operate under these conditions. For example, the shopping cart application requires that an "*Add to Cart"* operation can never be forgotten or rejected. If the most recent state of the cart is unavailable, and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved. But at the same time it shouldn't supersede the currently unavailable state of the cart, which itself may contain changes that should be preserved. Note that both "*add to cart*" and "*delete item from cart*" operations are translated into put requests to Dynamo. When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later.

In order to provide this kind of guarantee, Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version (syntactic reconciliation). However, version branching may happen, in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to *collapse* multiple branches of data evolution back into one (semantic reconciliation). A typical example of a collapse operation is "merging" different versions of a customer's shopping cart. Using this reconciliation mechanism, an "add to cart" operation is never lost. However, deleted items can resurface.

It is important to understand that certain failure modes can potentially result in the system having not just two but several versions of the same data. Updates in the presence of network partitions and node failures can potentially result in an object having distinct version sub-histories, which the system will need to reconcile in the future. This requires us to design applications that explicitly acknowledge the possibility of multiple versions of the same data (in order to never lose any updates).

Dynamo uses vector clocks [12] in order to capture causality between different versions of the same object. A vector clock is effectively a list of (node, counter) pairs. One vector clock is associated with every version of every object. One can determine whether two versions of an object are on parallel branches or have a causal ordering, by examine their vector clocks. If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation.

In Dynamo, when a client wishes to update an object, it must specify which version it is updating. This is done by passing the context it obtained from an earlier read operation, which contains the vector clock information. Upon processing a read request, if

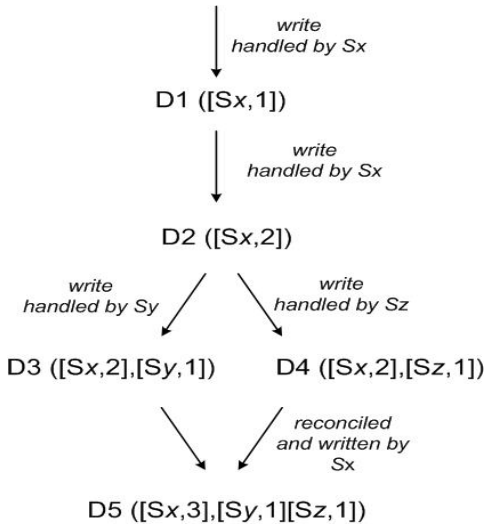


Figure 3: Version evolution of an object over time.

Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context. An update using this context is considered to have reconciled the divergent versions and the branches are collapsed into a single new version.

To illustrate the use of vector clocks, let us consider the example shown in Figure 3. A client writes a new object. The node (say Sx) that handles the write for this key increases its sequence number and uses it to create the data's vector clock. The system now has the object D1 and its associated clock [(Sx, 1)]. The client updates the object. Assume the same node handles this request as well. The system now also has object D2 and its associated clock [(Sx, 2)]. D2 *descends* from D1 and therefore over-writes D1, however there may be replicas of D1 lingering at nodes that have not yet seen D2. Let us assume that the same client updates the object again and a different server (say Sy) handles the request. The system now has data D3 and its associated clock [(Sx, 2), (Sy, 1)].

Next assume a different client reads D2 and then tries to update it, and another node (say Sz) does the write. The system now has D4 (descendant of D2) whose version clock is [(Sx, 2), (Sz, 1)]. A node that is aware of D1 or D2 could determine, upon receiving D4 and its clock, that D1 and D2 are overwritten by the new data and can be garbage collected. A node that is aware of D3 and receives D4 will find that there is no causal relation between them. In other words, there are changes in D3 and D4 that are not reflected in each other. Both versions of the data must be kept and presented to a client (upon a read) for semantic reconciliation.

Now assume some client reads both D3 and D4 (the context will reflect that both values were found by the read). The read's context is a summary of the clocks of D3 and D4, namely [(Sx, 2), (Sy, 1), (Sz, 1)]. If the client performs the reconciliation and node Sx coordinates the write, Sx will update its sequence number in the clock. The new data D5 will have the following clock: [(Sx, 3), (Sy, 1), (Sz, 1)].

A possible issue with vector clocks is that the size of vector clocks may grow if many servers coordinate the writes to an object. In practice, this is not likely because the writes are usually handled by one of the top N nodes in the preference list. In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow. In these scenarios, it is desirable to limit the size of vector clock. To this end, Dynamo employs the following clock truncation scheme: Along with each (node, counter) pair, Dynamo stores a timestamp that indicates the last time the node updated the data item. When the number of (node, counter) pairs in the vector clock reaches a threshold (say 10), the oldest pair is removed from the clock. Clearly, this truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. However, this problem has not surfaced in production and therefore this issue has not been thoroughly investigated.

4.5 Execution of get () and put () operations

Any storage node in Dynamo is eligible to receive client get and put operations for any key. In this section, for sake of simplicity, we describe how these operations are performed in a failure-free environment and in the subsequent section we describe how read and write operations are executed during failures.

Both get and put operations are invoked using Amazon's infrastructure-specific request processing framework over HTTP. There are two strategies that a client can use to select a node: (1) route its request through a generic load balancer that will select a node based on load information, or (2) use a partition-aware client library that routes requests directly to the appropriate coordinator nodes. The advantage of the first approach is that the client does not have to link any code specific to Dynamo in its application, whereas the second strategy can achieve lower latency because it skips a potential forwarding step.

A node handling a read or write operation is known as the *coordinator*. Typically, this is the first among the top N nodes in the preference list. If the requests are received through a load balancer, requests to access a key may be routed to any random node in the ring. In this scenario, the node that receives the request will not coordinate it if the node is not in the top N of the requested key's preference list. Instead, that node will forward the request to the first among the top N nodes in the preference list.

Read and write operations involve the first N healthy nodes in the preference list, skipping over those that are down or inaccessible. When all nodes are healthy, the top N nodes in a key's preference list are accessed. When there are node failures or network partitions, nodes that are lower ranked in the preference list are accessed.

To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems. This protocol has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that R + W > N yields a quorum-like system. In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Upon receiving a put() request for a key, the coordinator generates the vector clock for the new version and writes the new version locally. The coordinator then sends the new version (along with

the new vector clock) to the N highest-ranked reachable nodes. If at least W-1 nodes respond then the write is considered successful.

Similarly, for a get() request, the coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key, and then waits for R responses before returning the result to the client. If the coordinator ends up gathering multiple versions of the data, it returns all the versions it deems to be causally unrelated. The divergent versions are then reconciled and the reconciled version superseding the current versions is written back.

4.6 Handling Failures: Hinted Handoff

If Dynamo used a traditional quorum approach it would be unavailable during server failures and network partitions, and would have reduced durability even under the simplest of failure conditions. To remedy this it does not enforce strict quorum membership and instead it uses a "sloppy quorum"; all read and write operations are performed on the first N *healthy* nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring.

Consider the example of Dynamo configuration given in Figure 2 with N=3. In this example, if node A is temporarily down or unreachable during a write operation then a replica that would normally have lived on A will now be sent to node D. This is done to maintain the desired availability and durability guarantees. The replica sent to D will have a hint in its metadata that suggests which node was the intended recipient of the replica (in this case A). Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically. Upon detecting that A has recovered, D will attempt to deliver the replica to A. Once the transfer succeeds, D may delete the object from its local store without decreasing the total number of replicas in the system.

Using hinted handoff, Dynamo ensures that the read and write operations are not failed due to temporary node or network failures. Applications that need the highest level of availability can set W to 1, which ensures that a write is accepted as long as a single node in the system has durably written the key it to its local store. Thus, the write request is only rejected if all nodes in the system are unavailable. However, in practice, most Amazon services in production set a higher W to meet the desired level of durability. A more detailed discussion of configuring N, R and W follows in section 6.

It is imperative that a highly available storage system be capable of handling the failure of an entire data center(s). Data center failures happen due to power outages, cooling failures, network failures, and natural disasters. Dynamo is configured such that each object is replicated across multiple data centers. In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers. These datacenters are connected through high speed network links. This scheme of replicating across multiple datacenters allows us to handle entire data center failures without a data outage.

4.7 Handling permanent failures: Replica synchronization

Hinted handoff works best if the system membership churn is low and node failures are transient. There are scenarios under which hinted replicas become unavailable before they can be returned to the original replica node. To handle this and other threats to durability, Dynamo implements an anti-entropy (replica synchronization) protocol to keep the replicas synchronized.

To detect the inconsistencies between replicas faster and to minimize the amount of transferred data, Dynamo uses Merkle trees [13]. A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children. The principal advantage of Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the entire data set. Moreover, Merkle trees help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas. For instance, if the hash values of the root of two trees are equal, then the values of the leaf nodes in the tree are equal and the nodes require no synchronization. If not, it implies that the values of some replicas are different. In such cases, the nodes may exchange the hash values of children and the process continues until it reaches the leaves of the trees, at which point the hosts can identify the keys that are "out of sync". Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during the anti-entropy process.

Dynamo uses Merkle trees for anti-entropy as follows: Each node maintains a separate Merkle tree for each key range (the set of keys covered by a virtual node) it hosts. This allows nodes to compare whether the keys within a key range are up-to-date. In this scheme, two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common. Subsequently, using the tree traversal scheme described above the nodes determine if they have any differences and perform the appropriate synchronization action. The disadvantage with this scheme is that many key ranges change when a node joins or leaves the system thereby requiring the tree(s) to be recalculated. This issue is addressed, however, by the refined partitioning scheme described in Section 6.2.

4.8 Membership and Failure Detection

4.8.1 Ring Membership

In Amazon's environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. A node outage rarely signifies a permanent departure and therefore should not result in rebalancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Dynamo nodes. For these reasons, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring. An administrator uses a command line tool or a browser to connect to a Dynamo node and issue a membership change to join a node to a ring or remove a node from a ring. The node that serves the request writes the membership change and its time of issue to persistent store. The membership changes form a history because nodes can be removed and added back multiple times. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership. Each node contacts a peer chosen at random every second and the two nodes efficiently reconcile their persisted membership change histories.

When a node starts for the first time, it chooses its set of tokens (virtual nodes in the consistent hash space) and maps nodes to their respective token sets. The mapping is persisted on disk and

initially contains only the local node and token set. The mappings stored at different Dynamo nodes are reconciled during the same communication exchange that reconciles the membership change histories. Therefore, partitioning and placement information also propagates via the gossip-based protocol and each storage node is aware of the token ranges handled by its peers. This allows each node to forward a key's read/write operations to the right set of nodes directly.

4.8.2 External Discovery

The mechanism described above could temporarily result in a logically partitioned Dynamo ring. For example, the administrator could contact node A to join A to the ring, then contact node B to join B to the ring. In this scenario, nodes A and B would each consider itself a member of the ring, yet neither would be immediately aware of the other. To prevent logical partitions, some Dynamo nodes play the role of seeds. Seeds are nodes that are discovered via an external mechanism and are known to all nodes. Because all nodes eventually reconcile their membership with a seed, logical partitions are highly unlikely. Seeds can be obtained either from static configuration or from a configuration service. Typically seeds are fully functional nodes in the Dynamo ring.

4.8.3 Failure Detection

Failure detection in Dynamo is used to avoid attempts to communicate with unreachable peers during get() and put() operations and when transferring partitions and hinted replicas. For the purpose of avoiding failed attempts at communication, a purely local notion of failure detection is entirely sufficient: node A may consider node B failed if node B does not respond to node A's messages (even if B is responsive to node C's messages). In the presence of a steady rate of client requests generating inter-node communication in the Dynamo ring, a node A quickly discovers that a node B is unresponsive when B fails to respond to a message; Node A then uses alternate nodes to service requests that map to B's partitions; A periodically retries B to check for the latter's recovery. In the absence of client requests to drive traffic between two nodes, neither node really needs to know whether the other is reachable and responsive.

Decentralized failure detection protocols use a simple gossip-style protocol that enable each node in the system to learn about the arrival (or departure) of other nodes. For detailed information on decentralized failure detectors and the parameters affecting their accuracy, the interested reader is referred to [8]. Early designs of Dynamo used a decentralized failure detector to maintain a globally consistent view of failure state. Later it was determined that the explicit node join and leave methods obviates the need for a global view of failure state. This is because nodes are notified of permanent node additions and removals by the explicit node join and leave methods and temporary node failures are detected by the individual nodes when they fail to communicate with others (while forwarding requests).

4.9 Adding/Removing Storage Nodes

When a new node (say X) is added into the system, it gets assigned a number of tokens that are randomly scattered on the ring. For every key range that is assigned to node X, there may be a number of nodes (less than or equal to N) that are currently in charge of handling keys that fall within its token range. Due to the allocation of key ranges to X, some existing nodes no longer have to some of their keys and these nodes transfer those keys to X. Let us consider a simple bootstrapping scenario where node X is added to the ring shown in Figure 2 between A and B. When X is added to the system, it is in charge of storing keys in the ranges (F, G], (G, A] and (A, X]. As a consequence, nodes B, C and D no longer have to store the keys in these respective ranges. Therefore, nodes B, C, and D will offer to and upon confirmation from X transfer the appropriate set of keys. When a node is removed from the system, the reallocation of keys happens in a reverse process.

Operational experience has shown that this approach distributes the load of key distribution uniformly across the storage nodes, which is important to meet the latency requirements and to ensure fast bootstrapping. Finally, by adding a confirmation round between the source and the destination, it is made sure that the destination node does not receive any duplicate transfers for a given key range.

5. IMPLEMENTATION

In Dynamo, each storage node has three main software components: request coordination, membership and failure detection, and a local persistence engine. All these components are implemented in Java.

Dynamo's local persistence component allows for different storage engines to be plugged in. Engines that are in use are Berkeley Database (BDB) Transactional Data Store[2], BDB Java Edition, MySQL, and an in-memory buffer with persistent backing store. The main reason for designing a pluggable persistence component is to choose the storage engine best suited for an application's access patterns. For instance, BDB can handle objects typically in the order of tens of kilobytes whereas MySQL can handle objects of larger sizes. Applications choose Dynamo's local persistence engine based on their object size distribution. The majority of Dynamo's production instances use BDB Transactional Data Store.

The request coordination component is built on top of an event-driven messaging substrate where the message processing pipeline is split into multiple stages similar to the SEDA architecture [24]. All communications are implemented using Java NIO channels. The coordinator executes the read and write requests on behalf of clients by collecting data from one or more nodes (in the case of reads) or storing data at one or more nodes (for writes). Each client request results in the creation of a state machine on the node that received the client request. The state machine contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies and packaging the response to the client. Each state machine instance handles exactly one client request. For instance, a read operation implements the following state machine: (i) send read requests to the nodes, (ii) wait for minimum number of required responses, (iii) if too few replies were received within a given time bound, fail the request, (iv) otherwise gather all the data versions and determine the ones to be returned and (v) if versioning is enabled, perform syntactic reconciliation and generate an opaque write context that contains the vector clock that subsumes all the remaining versions. For the sake of brevity the failure handling and retry states are left out.

After the read response has been returned to the caller the state

[2] http://www.oracle.com/database/berkeley-db.html

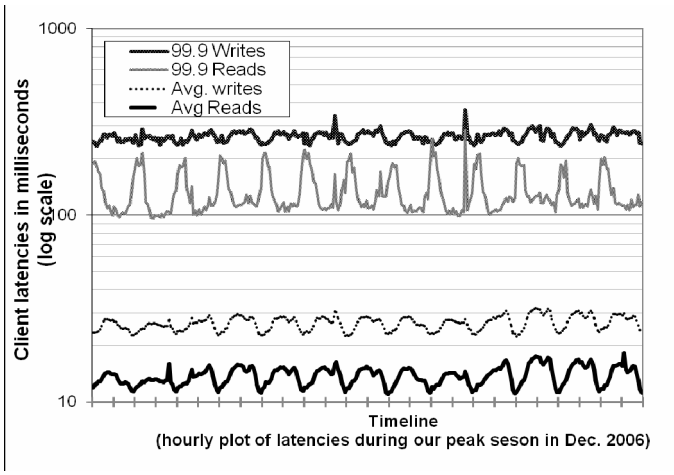


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

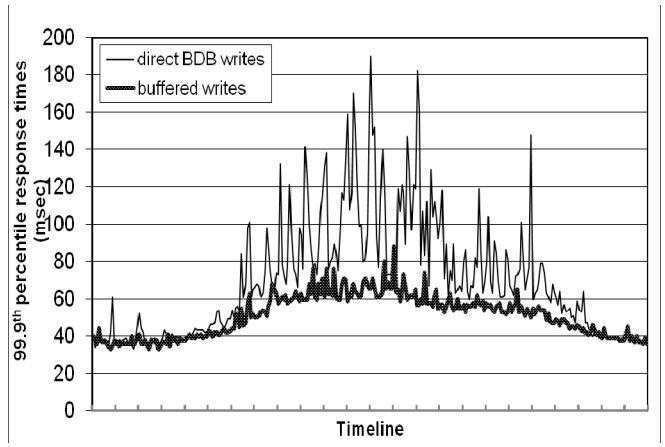


Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

machine waits for a small period of time to receive any outstanding responses. If stale versions were returned in any of the responses, the coordinator updates those nodes with the latest version. This process is called *read repair* because it repairs replicas that have missed a recent update at an opportunistic time and relieves the anti-entropy protocol from having to do it.

As noted earlier, write requests are coordinated by one of the top N nodes in the preference list. Although it is desirable always to have the first node among the top N to coordinate the writes thereby serializing all writes at a single location, this approach has led to uneven load distribution resulting in SLA violations. This is because the request load is not uniformly distributed across objects. To counter this, any of the top N nodes in the preference list is allowed to coordinate the writes. In particular, since each write usually follows a read operation, the coordinator for a write is chosen to be the node that replied fastest to the previous read operation which is stored in the context information of the request. This optimization enables us to pick the node that has the data that was read by the preceding read operation thereby increasing the chances of getting "read-your-writes" consistency. It also reduces variability in the performance of the request handling which improves the performance at the 99.9 percentile.

6. EXPERIENCES & LESSONS LEARNED

Dynamo is used by several services with different configurations. These instances differ by their version reconciliation logic, and read/write quorum characteristics. The following are the main patterns in which Dynamo is used:

- *Business logic specific reconciliation:* This is a popular use case for Dynamo. Each data object is replicated across multiple nodes. In case of divergent versions, the client application performs its own reconciliation logic. The shopping cart service discussed earlier is a prime example of this category. Its business logic reconciles objects by merging different versions of a customer's shopping cart.
- *Timestamp based reconciliation:* This case differs from the previous one only in the reconciliation mechanism. In case of divergent versions, Dynamo performs simple timestamp based reconciliation logic of "last write wins"; i.e., the object with the largest physical timestamp value is chosen as the correct version. The service that maintains customer's session information is a good example of a service that uses this mode.
- *High performance read engine:* While Dynamo is built to be an "always writeable" data store, a few services are tuning its quorum characteristics and using it as a high performance read engine. Typically, these services have a high read request rate and only a small number of updates. In this configuration, typically R is set to be 1 and W to be N. For these services, Dynamo provides the ability to partition and replicate their data across multiple nodes thereby offering incremental scalability. Some of these instances function as the authoritative persistence cache for data stored in more heavy weight backing stores. Services that maintain product catalog and promotional items fit in this category.

The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability. For instance, the value of N determines the durability of each object. A typical value of N used by Dynamo's users is 3.

The values of W and R impact object availability, durability and consistency. For instance, if W is set to 1, then the system will never reject a write request as long as there is at least one node in the system that can successfully process a write request. However, low values of W and R can increase the risk of inconsistency as write requests are deemed successful and returned to the clients even if they are not processed by a majority of the replicas. This also introduces a vulnerability window for durability when a write request is successfully returned to the client even though it has been persisted at only a small number of nodes.

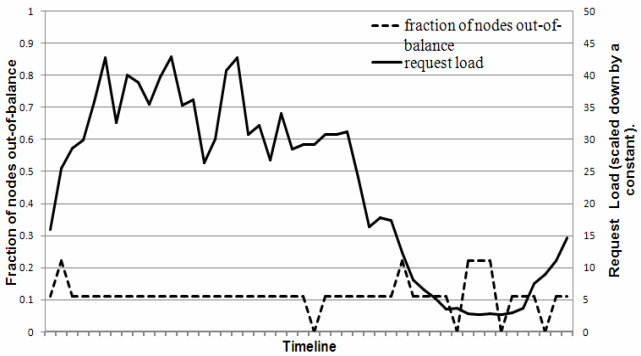


Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

Traditional wisdom holds that durability and availability go hand-in-hand. However, this is not necessarily true here. For instance, the vulnerability window for durability can be decreased by increasing W. This may increase the probability of rejecting requests (thereby decreasing availability) because more storage hosts need to be alive to process a write request.

The common (N,R,W) configuration used by several instances of Dynamo is (3,2,2). These values are chosen to meet the necessary levels of performance, durability, consistency, and availability SLAs.

All the measurements presented in this section were taken on a live system operating with a configuration of (3,2,2) and running a couple hundred nodes with homogenous hardware configurations. As mentioned earlier, each instance of Dynamo contains nodes that are located in multiple datacenters. These datacenters are typically connected through high speed network links. Recall that to generate a successful get (or put) response R (or W) nodes need to respond to the coordinator. Clearly, the network latencies between datacenters affect the response time and the nodes (and their datacenter locations) are chosen such that the applications target SLAs are met.

6.1 Balancing Performance and Durability

While Dynamo's principle design goal is to build a highly available data store, performance is an equally important criterion in Amazon's platform. As noted earlier, to provide a consistent customer experience, Amazon's services set their performance targets at higher percentiles (such as the 99.9th or 99.99th percentiles). A typical SLA required of services that use Dynamo is that 99.9% of the read and write requests execute within 300ms.

Since Dynamo is run on standard commodity hardware components that have far less I/O throughput than high-end enterprise servers, providing consistently high performance for read and write operations is a non-trivial task. The involvement of multiple storage nodes in read and write operations makes it even more challenging, since the performance of these operations is limited by the slowest of the R or W replicas. Figure 4 shows the average and 99.9th percentile latencies of Dynamo's read and write operations during a period of 30 days. As seen in the figure, the latencies exhibit a clear diurnal pattern which is a result of the diurnal pattern in the incoming request rate (i.e., there is a significant difference in request rate between the daytime and night). Moreover, the write latencies are higher than read latencies obviously because write operations always results in disk access. Also, the 99.9th percentile latencies are around 200 ms and are an order of magnitude higher than the averages. This is because the 99.9th percentile latencies are affected by several factors such as variability in request load, object sizes, and locality patterns.

While this level of performance is acceptable for a number of services, a few customer-facing services required higher levels of performance. For these services, Dynamo provides the ability to trade-off durability guarantees for performance. In the optimization each storage node maintains an object buffer in its main memory. Each write operation is stored in the buffer and gets periodically written to storage by a *writer thread*. In this scheme, read operations first check if the requested key is present in the buffer. If so, the object is read from the buffer instead of the storage engine.

This optimization has resulted in lowering the 99.9th percentile latency by a factor of 5 during peak traffic even for a very small buffer of a thousand objects (see Figure 5). Also, as seen in the figure, write buffering smoothes out higher percentile latencies. Obviously, this scheme trades durability for performance. In this scheme, a server crash can result in missing writes that were queued up in the buffer. To reduce the durability risk, the write operation is refined to have the coordinator choose one out of the N replicas to perform a "durable write". Since the coordinator waits only for W responses, the performance of the write operation is not affected by the performance of the durable write operation performed by a single replica.

6.2 Ensuring Uniform Load distribution

Dynamo uses consistent hashing to partition its key space across its replicas and to ensure uniform load distribution. A uniform key distribution can help us achieve uniform load distribution assuming the access distribution of keys is not highly skewed. In particular, Dynamo's design assumes that even where there is a significant skew in the access distribution there are enough keys in the popular end of the distribution so that the load of handling popular keys can be spread across the nodes uniformly through partitioning. This section discusses the load imbalance seen in Dynamo and the impact of different partitioning strategies on load distribution.

To study the load imbalance and its correlation with request load, the total number of requests received by each node was measured for a period of 24 hours - broken down into intervals of 30 minutes. In a given time window, a node is considered to be "in-balance", if the node's request load deviates from the average load by a value a less than a certain threshold (here 15%). Otherwise the node was deemed "out-of-balance". Figure 6 presents the fraction of nodes that are "out-of-balance" (henceforth, "imbalance ratio") during this time period. For reference, the corresponding request load received by the entire system during this time period is also plotted. As seen in the figure, the imbalance ratio decreases with increasing load. For instance, during low loads the imbalance ratio is as high as 20% and during high loads it is close to 10%. Intuitively, this can be explained by the fact that under high loads, a large number of popular keys are accessed and due to uniform distribution of keys the load is evenly distributed. However, during low loads (where load is 1/8th

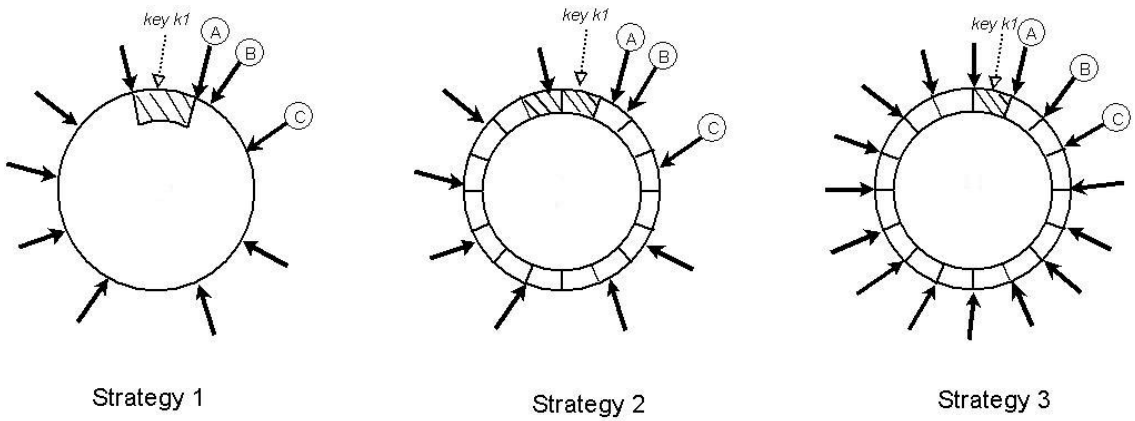


Figure 7: Partitioning and placement of keys in the three strategies. A, B, and C depict the three unique nodes that form the preference list for the key k1 on the consistent hashing ring (N=3). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows indicate the token locations for various nodes.

of the measured peak load), fewer popular keys are accessed, resulting in a higher load imbalance.

This section discusses how Dynamo's partitioning scheme has evolved over time and its implications on load distribution.

Strategy 1: T random tokens per node and partition by token value: This was the initial strategy deployed in production (and described in Section 4.2). In this scheme, each node is assigned T tokens (chosen uniformly at random from the hash space). The tokens of all nodes are ordered according to their values in the hash space. Every two consecutive tokens define a range. The last token and the first token form a range that "wraps" around from the highest value to the lowest value in the hash space. Because the tokens are chosen randomly, the ranges vary in size. As nodes join and leave the system, the token set changes and consequently the ranges change. Note that the space needed to maintain the membership at each node increases linearly with the number of nodes in the system.

While using this strategy, the following problems were encountered. First, when a new node joins the system, it needs to "steal" its key ranges from other nodes. However, the nodes handing the key ranges off to the new node have to scan their local persistence store to retrieve the appropriate set of data items. Note that performing such a scan operation on a production node is tricky as scans are highly resource intensive operations and they need to be executed in the background without affecting the customer performance. This requires us to run the bootstrapping task at the lowest priority. However, this significantly slows the bootstrapping process and during busy shopping season, when the nodes are handling millions of requests a day, the bootstrapping has taken almost a day to complete. Second, when a node joins/leaves the system, the key ranges handled by many nodes change and the Merkle trees for the new ranges need to be recalculated, which is a non-trivial operation to perform on a production system. Finally, there was no easy way to take a snapshot of the entire key space due to the randomness in key ranges, and this made the process of archival complicated. In this scheme, archiving the entire key space requires us to retrieve the keys from each node separately, which is highly inefficient.

The fundamental issue with this strategy is that the schemes for data partitioning and data placement are intertwined. For instance, in some cases, it is preferred to add more nodes to the system in order to handle an increase in request load. However, in this scenario, it is not possible to add nodes without affecting data partitioning. Ideally, it is desirable to use independent schemes for partitioning and placement. To this end, following strategies were evaluated:

Strategy 2: T random tokens per node and equal sized partitions: In this strategy, the hash space is divided into Q equally sized partitions/ranges and each node is assigned T random tokens. Q is usually set such that Q >> N and Q >> S*T, where S is the number of nodes in the system. In this strategy, the tokens are only used to build the function that maps values in the hash space to the ordered lists of nodes and not to decide the partitioning. A partition is placed on the first N unique nodes that are encountered while walking the consistent hashing ring clockwise from the end of the partition. Figure 7 illustrates this strategy for N=3. In this example, nodes A, B, C are encountered while walking the ring from the end of the partition that contains key k1. The primary advantages of this strategy are: (i) decoupling of partitioning and partition placement, and (ii) enabling the possibility of changing the placement scheme at runtime.

Strategy 3: Q/S tokens per node, equal-sized partitions: Similar to strategy 2, this strategy divides the hash space into Q equally sized partitions and the placement of partition is decoupled from the partitioning scheme. Moreover, each node is assigned Q/S tokens where S is the number of nodes in the system. When a node leaves the system, its tokens are randomly distributed to the remaining nodes such that these properties are preserved. Similarly, when a node joins the system it "steals" tokens from nodes in the system in a way that preserves these properties.

The efficiency of these three strategies is evaluated for a system with S=30 and N=3. However, comparing these different strategies in a fair manner is hard as different strategies have different configurations to tune their efficiency. For instance, the load distribution property of strategy 1 depends on the number of tokens (i.e., T) while strategy 3 depends on the number of partitions (i.e., Q). One fair way to compare these strategies is to

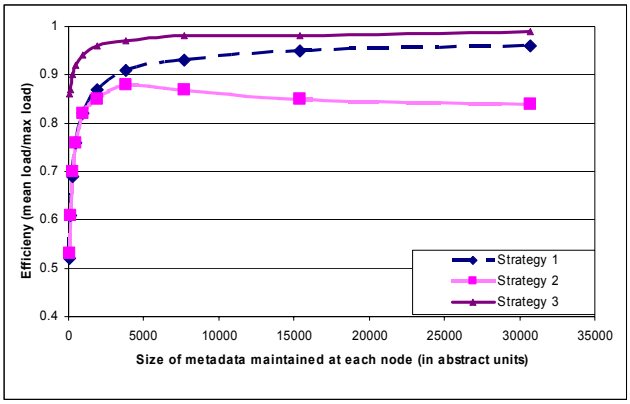


Figure 8: Comparison of the load distribution efficiency of different strategies for system with 30 nodes and N=3 with equal amount of metadata maintained at each node. The values of the system size and number of replicas are based on the typical configuration deployed for majority of our services.

evaluate the skew in their load distribution while all strategies use the same amount of space to maintain their membership information. For instance, in strategy 1 each node needs to maintain the token positions of all the nodes in the ring and in strategy 3 each node needs to maintain the information regarding the partitions assigned to each node.

In our next experiment, these strategies were evaluated by varying the relevant parameters (T and Q). The load balancing efficiency of each strategy was measured for different sizes of membership information that needs to be maintained at each node, where *Load balancing efficiency* is defined as the ratio of average number of requests served by each node to the maximum number of requests served by the hottest node.

The results are given in Figure 8. As seen in the figure, strategy 3 achieves the best load balancing efficiency and strategy 2 has the worst load balancing efficiency. For a brief time, Strategy 2 served as an interim setup during the process of migrating Dynamo instances from using Strategy 1 to Strategy 3. Compared to Strategy 1, Strategy 3 achieves better efficiency and reduces the size of membership information maintained at each node by three orders of magnitude. While storage is not a major issue the nodes gossip the membership information periodically and as such it is desirable to keep this information as compact as possible. In addition to this, strategy 3 is advantageous and simpler to deploy for the following reasons: (i) *Faster bootstrapping/recovery:* Since partition ranges are fixed, they can be stored in separate files, meaning a partition can be relocated as a unit by simply transferring the file (avoiding random accesses needed to locate specific items). This simplifies the process of bootstrapping and recovery. (ii) *Ease of archival*: Periodical archiving of the dataset is a mandatory requirement for most of Amazon storage services. Archiving the entire dataset stored by Dynamo is simpler in strategy 3 because the partition files can be archived separately. By contrast, in Strategy 1, the tokens are chosen randomly and, archiving the data stored in Dynamo requires retrieving the keys from individual nodes separately and is usually inefficient and slow. The disadvantage of strategy 3 is that changing the node membership requires coordination in order to preserve the properties required of the assignment.

6.3 Divergent Versions: When and How Many?

As noted earlier, Dynamo is designed to tradeoff consistency for availability. To understand the precise impact of different failures on consistency, detailed data is required on multiple factors: outage length, type of failure, component reliability, workload etc. Presenting these numbers in detail is outside of the scope of this paper. However, this section discusses a good summary metric: the number of divergent versions seen by the application in a live production environment.

Divergent versions of a data item arise in two scenarios. The first is when the system is facing failure scenarios such as node failures, data center failures, and network partitions. The second is when the system is handling a large number of concurrent writers to a single data item and multiple nodes end up coordinating the updates concurrently. From both a usability and efficiency perspective, it is preferred to keep the number of divergent versions at any given time as low as possible. If the versions cannot be syntactically reconciled based on vector clocks alone, they have to be passed to the business logic for semantic reconciliation. Semantic reconciliation introduces additional load on services, so it is desirable to minimize the need for it.

In our next experiment, the number of versions returned to the shopping cart service was profiled for a period of 24 hours. During this period, 99.94% of requests saw exactly one version; 0.00057% of requests saw 2 versions; 0.00047% of requests saw 3 versions and 0.00009% of requests saw 4 versions. This shows that divergent versions are created rarely.

Experience shows that the increase in the number of divergent versions is contributed not by failures but due to the increase in number of concurrent writers. The increase in the number of concurrent writes is usually triggered by busy robots (automated client programs) and rarely by humans. This issue is not discussed in detail due to the sensitive nature of the story.

6.4 Client-driven or Server-driven Coordination

As mentioned in Section 5, Dynamo has a request coordination component that uses a state machine to handle incoming requests. Client requests are uniformly assigned to nodes in the ring by a load balancer. Any Dynamo node can act as a coordinator for a read request. Write requests on the other hand will be coordinated by a node in the key's current preference list. This restriction is due to the fact that these preferred nodes have the added responsibility of creating a new version stamp that causally subsumes the version that has been updated by the write request. Note that if Dynamo's versioning scheme is based on physical timestamps, any node can coordinate a write request.

An alternative approach to request coordination is to move the state machine to the client nodes. In this scheme client applications use a library to perform request coordination locally. A client periodically picks a random Dynamo node and downloads its current view of Dynamo membership state. Using this information the client can determine which set of nodes form the preference list for any given key. Read requests can be coordinated at the client node thereby avoiding the extra network hop that is incurred if the request were assigned to a random Dynamo node by the load balancer. Writes will either be forwarded to a node in the key's preference list or can be

Table 2: Performance of client-driven and server-driven coordination approaches.

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

coordinated locally if Dynamo is using timestamps based versioning.

An important advantage of the client-driven coordination approach is that a load balancer is no longer required to uniformly distribute client load. Fair load distribution is implicitly guaranteed by the near uniform assignment of keys to the storage nodes. Obviously, the efficiency of this scheme is dependent on how fresh the membership information is at the client. Currently clients poll a random Dynamo node every 10 seconds for membership updates. A pull based approach was chosen over a push based one as the former scales better with large number of clients and requires very little state to be maintained at servers regarding clients. However, in the worst case the client can be exposed to stale membership for duration of 10 seconds. In case, if the client detects its membership table is stale (for instance, when some members are unreachable), it will immediately refresh its membership information.

Table 2 shows the latency improvements at the 99.9th percentile and averages that were observed for a period of 24 hours using client-driven coordination compared to the server-driven approach. As seen in the table, the client-driven coordination approach reduces the latencies by at least 30 milliseconds for 99.9th percentile latencies and decreases the average by 3 to 4 milliseconds. The latency improvement is because the client-driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node. As seen in the table, average latencies tend to be significantly lower than latencies at the 99.9th percentile. This is because Dynamo's storage engine caches and write buffer have good hit ratios. Moreover, since the load balancers and network introduce additional variability to the response time, the gain in response time is higher for the 99.9th percentile than the average.

6.5 Balancing background vs. foreground tasks

Each node performs different kinds of background tasks for replica synchronization and data handoff (either due to hinting or adding/removing nodes) in addition to its normal foreground put/get operations. In early production settings, these background tasks triggered the problem of resource contention and affected the performance of the regular put and get operations. Hence, it became necessary to ensure that background tasks ran only when the regular critical operations are not affected significantly. To this end, the background tasks were integrated with an admission control mechanism. Each of the background tasks uses this controller to reserve runtime slices of the resource (e.g. database), shared across all background tasks. A feedback mechanism based on the monitored performance of the foreground tasks is employed to change the number of slices that are available to the background tasks.

The admission controller constantly monitors the behavior of resource accesses while executing a "foreground" put/get operation. Monitored aspects include latencies for disk operations, failed database accesses due to lock-contention and transaction timeouts, and request queue wait times. This information is used to check whether the percentiles of latencies (or failures) in a given trailing time window are close to a desired threshold. For example, the background controller checks to see how close the 99th percentile database read latency (over the last 60 seconds) is to a preset threshold (say 50ms). The controller uses such comparisons to assess the resource availability for the foreground operations. Subsequently, it decides on how many time slices will be available to background tasks, thereby using the feedback loop to limit the intrusiveness of the background activities. Note that a similar problem of managing background tasks has been studied in [4].

6.6 Discussion

This section summarizes some of the experiences gained during the process of implementation and maintenance of Dynamo. Many Amazon internal services have used Dynamo for the past two years and it has provided significant levels of availability to its applications. In particular, applications have received successful responses (without timing out) for 99.9995% of its requests and no data loss event has occurred to date.

Moreover, the primary advantage of Dynamo is that it provides the necessary knobs using the three parameters of (N,R,W) to tune their instance based on their needs.. Unlike popular commercial data stores, Dynamo exposes data consistency and reconciliation logic issues to the developers. At the outset, one may expect the application logic to become more complex. However, historically, Amazon's platform is built for high availability and many applications are designed to handle different failure modes and inconsistencies that may arise. Hence, porting such applications to use Dynamo was a relatively simple task. For new applications that want to use Dynamo, some analysis is required during the initial stages of the development to pick the right conflict resolution mechanisms that meet the business case appropriately. Finally, Dynamo adopts a full membership model where each node is aware of the data hosted by its peers. To do this, each node actively gossips the full routing table with other nodes in the system. This model works well for a system that contains couple of hundreds of nodes. However, scaling such a design to run with tens of thousands of nodes is not trivial because the overhead in maintaining the routing table increases with the system size. This limitation might be overcome by introducing hierarchical extensions to Dynamo. Also, note that this problem is actively addressed by O(1) DHT systems(e.g., [14]).

7. CONCLUSIONS

This paper described Dynamo, a highly available and scalable data store, used for storing state of a number of core services of Amazon.com's e-commerce platform. Dynamo has provided the desired levels of availability and performance and has been successful in handling server failures, data center failures and network partitions. Dynamo is incrementally scalable and allows service owners to scale up and down based on their current

request load. Dynamo allows service owners to customize their storage system to meet their desired performance, durability and consistency SLAs by allowing them to tune the parameters N, R, and W.

The production use of Dynamo for the past year demonstrates that decentralized techniques can be combined to provide a single highly-available system. Its success in one of the most challenging application environments shows that an eventual-consistent storage system can be a building block for highly-available applications.

ACKNOWLEDGEMENTS

The authors would like to thank Pat Helland for his contribution to the initial design of Dynamo. We would also like to thank Marvin Theimer and Robert van Renesse for their comments. Finally, we would like to thank our shepherd, Jeff Mogul, for his detailed comments and inputs while preparing the camera ready version that vastly improved the quality of the paper.

REFERENCES

[1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 1-14.

[2] Bernstein, P.A., and Goodman, N. An algorithm for concurrency control and recovery in replicated distributed databases. ACM Trans. on Database Systems, 9(4):596-615, December 1984

[3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA, November 06 - 08, 2006). USENIX Association, Berkeley, CA, 15-15.

[4] Douceur, J. R. and Bolosky, W. J. 2000. Process-based regulation of low-importance processes. *SIGOPS Oper. Syst. Rev.* 34, 2 (Apr. 2000), 26-27.

[5] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France, October 05 - 08, 1997). W. M. Waite, Ed. SOSP '97. ACM Press, New York, NY, 78-91.

[6] Ghemawat, S., Gobioff, H., and Leung, S. 2003. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM Press, New York, NY, 29-43.

[7] Gray, J., Helland, P., O'Neil, P., and Shasha, D. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data* (Montreal, Quebec, Canada, June 04 - 06, 1996). J. Widom, Ed. SIGMOD '96. ACM Press, New York, NY, 173-182.

[8] Gupta, I., Chandra, T. D., and Goldszmidt, G. S. 2001. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, Rhode Island, United States). PODC '01. ACM Press, New York, NY, 170-179.

[9] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. 2000. OceanStore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News* 28, 5 (Dec. 2000), 190-201.

[10] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing* (El Paso, Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663.

[11] Lindsay, B.G., et. al., "Notes on Distributed Databases", Research Report RJ2571(33471), IBM Research, July 1979

[12] Lamport, L. Time, clocks, and the ordering of events in a distributed system. ACM Communications, 21(7), pp. 558-565, 1978.

[13] Merkle, R. A digital signature based on a conventional encryption function. Proceedings of CRYPTO, pages 369–378. Springer-Verlag, 1988.

[14] Ramasubramanian, V., and Sirer, E. G. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation,* San Francisco, CA, March 29 - 31, 2004.

[15] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. 1994. Resolving file conflicts in the Ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (Boston, Massachusetts, June 06 - 10, 1994). USENIX Association, Berkeley, CA, 12-12..

[16] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Proceedings of Middleware, pages 329-350, November, 2001.

[17] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. Proceedings of Symposium on Operating Systems Principles, October 2001.

[18] Saito, Y., Frølund, S., Veitch, A., Merchant, A., and Spence, S. 2004. FAB: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.* 38, 5 (Dec. 2004), 48-58.

[19] Satyanarayanan, M., Kistler, J.J., Siegel, E.H. Coda: A Resilient Distributed File System. IEEE Workshop on Workstation Operating Systems, Nov. 1987.

[20] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California, United States). SIGCOMM '01. ACM Press, New York, NY, 149-160.

[21] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, United States, December 03 - 06, 1995). M. B. Jones, Ed. SOSP '95. ACM Press, New York, NY, 172-182.

[22] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems 4 (2): 180-209, 1979.

[23] Weatherspoon, H., Eaton, P., Chun, B., and Kubiatowicz, J. 2007. Antiquity: exploiting a secure log for wide-area distributed storage. *SIGOPS Oper. Syst. Rev.* 41, 3 (Jun. 2007), 371-384.

[24] Welsh, M., Culler, D., and Brewer, E. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada, October 21 - 24, 2001). SOSP '01. ACM Press, New York, NY, 230-243.

Paxos (computer science)

From Wikipedia, the free encyclopedia

Paxos is a family of protocols for solving consensus in a network of unreliable processors. Consensus is the process of agreeing on one result among a group of participants. This problem becomes difficult when the participants or their communication medium may experience failures.[1]

Consensus protocols are the basis for the state machine replication approach to distributed computing, as suggested by Leslie Lamport[2] and surveyed by Fred B. Schneider.[3] State machine replication is a technique for converting an algorithm into a fault-tolerant, distributed implementation. Ad-hoc techniques may leave important cases of failures unresolved. The principled approach proposed by Lamport et al. ensures all cases are handled safely.

The Paxos protocol was first published in 1989 and named after a fictional legislative consensus system used on the Paxos island in Greece.[4] It was later published as a journal article in 1998.[5]

The Paxos family of protocols includes a spectrum of trade-offs between the number of processors, number of message delays before learning the agreed value, the activity level of individual participants, number of messages sent, and types of failures. Although no deterministic fault-tolerant consensus protocol can guarantee progress in an asynchronous network (a result proven in a paper by Fischer, Lynch and Paterson),[6] Paxos guarantees safety (consistency), and the conditions that could prevent it from making progress are difficult to provoke.[5][7][8][9][10]

Paxos is usually used where durability is required (for example, to replicate a file or a database), in which the amount of durable state could be large. The protocol attempts to make progress even during periods when some bounded number of replicas are unresponsive. There is also a mechanism to drop a permanently failed replica or to add a new replica.

Contents

- 1 History
- 2 Assumptions
  - 2.1 Processors
  - 2.2 Network
  - 2.3 Number of processors
- 3 Roles
  - 3.1 Quorums
  - 3.2 Proposal Number & Agreed Value
- 4 Safety and liveness properties
- 5 Typical deployment
- 6 Basic Paxos
  - 6.1 Phase 1a: *Prepare*
  - 6.2 Phase 1b: *Promise*
  - 6.3 Phase 2a: *Accept Request*

- 6.4 Phase 2b: *Accepted*
- 6.5 Message flow: Basic Paxos
- 6.6 Error cases in Basic Paxos
- 6.7 Message flow: Basic Paxos, failure of Acceptor
- 6.8 Message flow: Basic Paxos, failure of redundant Learner
- 6.9 Message flow: Basic Paxos, failure of Proposer
- 6.10 Message flow: Basic Paxos, dueling Proposers
- 7 Multi-Paxos
  - 7.1 Message flow: Multi-Paxos, start
  - 7.2 Message flow: Multi-Paxos, steady-state
  - 7.3 Typical Multi-Paxos Collapsed Roles deployment
  - 7.4 Message flow: Multi-Paxos Collapsed Roles, start
  - 7.5 Message flow: Multi-Paxos Collapsed Roles, steady state
- 8 Optimizations
- 9 Cheap Paxos
  - 9.1 Message flow: Cheap Multi-Paxos
- 10 Fast Paxos
  - 10.1 Message flow: Fast Paxos, non-conflicting
  - 10.2 Message flow: Fast Paxos, conflicting proposals
  - 10.3 Message flow: Fast Paxos, collapsed roles
- 11 Generalized Paxos
  - 11.1 Example
  - 11.2 Message flow: Generalized Paxos (example)
  - 11.3 Performance
- 12 Byzantine Paxos
  - 12.1 Message flow: Byzantine Multi-Paxos, steady state
  - 12.2 Message flow: Fast Byzantine Multi-Paxos, steady state
  - 12.3 Message flow: Fast Byzantine Multi-Paxos, failure
- 13 Production use of Paxos
- 14 See also
- 15 References
- 16 External links

History

The topic predates the protocol. In 1988, Lynch, Dwork and Stockmeyer had demonstrated [11] the solvability of consensus in a broad family of "partially synchronous" systems. Paxos has strong similarities to a protocol used for agreement in viewstamped replication, first published by Oki and Liskov in 1988, in the context of distributed transactions.[12] Not withstanding this prior work, Paxos offered a particularly elegant formalism, and included one of the earliest proofs of safety for a fault-tolerant distributed consensus protocol.

Reconfigurable state machines have strong ties to prior work on reliable group multicast protocols that support dynamic group membership, for example Birman's work in 1985 and 1987 on the virtually synchronous gbcast[13] protocol. However, it should be noted that gbcast is unusual in supporting durability and addressing partitioning failures. Most reliable

multicast protocols lack these properties, which are required for implementations of the state machine replication model. This point is elaborated in a paper by Lamport, Malkhi and Zhou.[14]

Assumptions

In order to simplify the presentation of Paxos, the following assumptions and definitions are made explicit. Techniques to broaden the applicability are known in the literature, and are not covered in this article.

Processors

- Processors operate at arbitrary speed.
- Processors may experience failures.
- Processors with stable storage may re-join the protocol after failures (following a crash-recovery failure model).
- Processors do not collude, lie, or otherwise attempt to subvert the protocol. (That is, Byzantine failures don't occur. See Byzantine Paxos for a solution that tolerates failures that arise from arbitrary/malicious behavior of the processes.)

Network

- Processors can send messages to any other processor.
- Messages are sent asynchronously and may take arbitrarily long to deliver.
- Messages may be lost, reordered, or duplicated.
- Messages are delivered without corruption. (That is, Byzantine failures don't occur. See Byzantine Paxos for a solution which tolerates corrupted messages that arise from arbitrary/malicious behavior of the messaging channels.)

Number of processors

In general, a consensus algorithm can make progress using 2F+1 processors despite the simultaneous failure of any F processors.[15] However, using reconfiguration, a protocol may be employed which survives any number of total failures as long as no more than F fail simultaneously.

Roles

Paxos describes the actions of the processes by their roles in the protocol: client, acceptor, proposer, learner, and leader. In typical implementations, a single processor may play one or more roles at the same time. This does not affect the correctness of the protocol—it is usual to coalesce roles to improve the latency and/or number of messages in the protocol.

Client
: The Client issues a *request* to the distributed system, and waits for a *response*. For instance, a write request on a file in a distributed file server.

Acceptor (Voters)
: The Acceptors act as the fault-tolerant "memory" of the protocol. Acceptors can form Quorums (cf. the definition of Quorum below). Any message sent to an Acceptor must be sent to a Quorum of Acceptors. Any message received from an Acceptor is ignored

unless a copy is received from each Acceptor in a Quorum (e.g., Promises sent to a Proposer, or Accepted messages sent to a Learner).

Proposer
: A Proposer advocates a client request, attempting to convince the Acceptors to agree on it, and acting as a coordinator to move the protocol forward when conflicts occur.

Learner
: Learners act as the replication factor for the protocol. Once a Client request has been agreed on by the Acceptors, the Learner may take action (i.e.: execute the request and send a response to the client). To improve availability of processing, additional Learners can be added.

Leader
: Paxos requires a distinguished Proposer (called the leader) to make progress. Many processes may believe they are leaders, but the protocol only guarantees progress if one of them is eventually chosen. If two processes believe they are leaders, they may stall the protocol by continuously proposing conflicting updates. However, the safety properties are still preserved in that case.

Quorums

Quorums express the safety properties of Paxos by ensuring at least some surviving processor retains knowledge of the results.

Quorums are defined as subsets of the set of Acceptors such that any two subsets (that is, any two Quorums) share at least one member. Typically, a Quorum is any majority of participating Acceptors. For example, given the set of Acceptors {A,B,C,D}, a majority Quorum would be any three Acceptors: {A,B,C}, {A,C,D}, {A,B,D}, {B,C,D}. More generally, arbitrary positive weights can be assigned to Acceptors and a Quorum defined as any subset of Acceptors with the summary weight greater than half of the total weight of all Acceptors.

Proposal Number & Agreed Value

Each attempt to define an agreed value v is performed with proposals which may or may not be accepted by Acceptors. Each proposal is uniquely numbered for a given Proposer. The value corresponding to a numbered proposal can be computed as part of running the Paxos protocol, but need not be.

Safety and liveness properties

In order to guarantee safety, Paxos defines three safety properties and ensures they are always held, regardless of the pattern of failures:

Non-triviality
: Only proposed values can be learned.[8]

Safety
: At most one value can be learned (i.e., two different learners cannot learn different values).[8][9]

Liveness(C;L)
: If value C has been proposed, then eventually learner L will learn some value (if sufficient processes remain non-faulty).[9]

Typical deployment

In most deployments of Paxos, each participating process acts in three roles; Proposer, Acceptor and Learner.[16] This reduces the message complexity significantly, without sacrificing correctness:

> *In Paxos, clients send commands to a leader. During normal operation, the leader receives a client's command, assigns it a new command number i, and then begins the ith instance of the consensus algorithm by sending messages to a set of acceptor processes.*[9]

By merging roles, the protocol "collapses" into an efficient client-master-replica style deployment, typical of the database community. The benefit of the Paxos protocols (including implementations with merged roles) is the guarantee of its safety properties.

A typical implementation's message flow is covered in the section Multi-Paxos.

Basic Paxos

This protocol is the most basic of the Paxos family. Each instance of the Basic Paxos protocol decides on a single output value. The protocol proceeds over several rounds. A successful round has two phases. A Proposer should not initiate Paxos if it cannot communicate with at least a Quorum of Acceptors:

Phase 1a: *Prepare*

A Proposer (the leader) creates a proposal identified with a number N. This number must be greater than any previous proposal number used by this Proposer. Then, it sends a *Prepare* message containing this proposal to a Quorum of Acceptors. The Proposer decides who is in the Quorum.

Phase 1b: *Promise*

If the proposal's number N is higher than any previous proposal number received from any Proposer by the Acceptor, then the Acceptor must return a promise to ignore all future proposals having a number less than N. If the Acceptor accepted a proposal at some point in the past, it must include the previous proposal number and previous value in its response to the Proposer.

Otherwise, the Acceptor can ignore the received proposal. It does not have to answer in this case for Paxos to work. However, for the sake of optimization, sending a denial (*Nack*) response would tell the Proposer that it can stop its attempt to create consensus with proposal N.

Phase 2a: *Accept Request*

If a Proposer receives enough promises from a Quorum of Acceptors, it needs to set a value to its proposal. If any Acceptors had previously accepted any proposal, then they'll have sent their values to the Proposer, who now must set the value of its proposal to the value associated with the highest proposal number reported by the Acceptors. If

none of the Acceptors had accepted a proposal up to this point, then the Proposer may choose any value for its proposal.[17]

The Proposer sends an *Accept Request* message to a Quorum of Acceptors with the chosen value for its proposal.

Phase 2b: *Accepted*

If an Acceptor receives an Accept Request message for a proposal N, it must accept it if and only if it has not already promised to any prepare proposals having an identifier greater than N. In this case, it should register the corresponding value v and send an *Accepted* message to the Proposer and every Learner. Else, it can ignore the Accept Request.

Note that an Acceptor can accept multiple proposals. These proposals may even have different values in the presence of certain failures. However, the Paxos protocol will guarantee that the Acceptors will ultimately agree on a single value.

Rounds fail when multiple Proposers send conflicting *Prepare* messages, or when the Proposer does not receive a Quorum of responses (*Promise* or *Accepted*). In these cases, another round must be started with a higher proposal number.

Notice that when Acceptors accept a request, they also acknowledge the leadership of the Proposer. Hence, Paxos can be used to select a leader in a cluster of nodes.

Here is a graphic representation of the Basic Paxos protocol. Note that the values returned in the *Promise* message are null the first time a proposal is made, since no Acceptor has accepted a value before in this round.

A Learner learns a value only if it receives a Quorum of *Accepted* messages with the same value.

Message flow: Basic Paxos

(first round is successful)

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(1)
   |         |<---------X--X--X       |  |  Promise(1,{Va,Vb,Vc})
   |         X--------->|->|->|       |  |  Accept!(1,Vn)
   |         |<---------X--X--X------>|->|  Accepted(1,Vn)
   |<----------------------------------X--X  Response
   |         |          |  |  |       |  |
```

Vn = highest of (Va,Vb,Vc)

Error cases in Basic Paxos

The simplest error cases are the failure of a redundant Learner, or failure of an Acceptor when a Quorum of Acceptors remains live. In these cases, the protocol requires no recovery. No additional rounds or messages are required, as shown below:

Message flow: Basic Paxos, failure of Acceptor

(Quorum size = 2 Acceptors)

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(1)
   |         |          |  |  !       |  |  !! FAIL !!
   |         |<---------X--X          |  |  Promise(1,{null,null})
   |         X--------->|->|          |  |  Accept!(1,V)
   |         |<---------X--X--------->|->|  Accepted(1,V)
   |<--------------------------------X--X  Response
   |         |          |  |          |  |
```

Message flow: Basic Paxos, failure of redundant Learner

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(1)
   |         |<---------X--X--X       |  |  Promise(1,{null,null,null})
   |         X--------->|->|->|       |  |  Accept!(1,V)
   |         |<---------X--X--X------>|->|  Accepted(1,V)
   |         |          |  |  |       |  !  !! FAIL !!
   |<--------------------------------X     Response
   |         |          |  |  |       |
```

The next failure case is when a Proposer fails after proposing a value, but before agreement is reached. Ignoring Leader election, an example message flow is as follows:

Message flow: Basic Paxos, failure of Proposer

(re-election not shown, one instance, two rounds)

```
Client  Proposer        Acceptor     Learner
   |      |             |  |  |       |  |
   X----->|             |  |  |       |  |  Request
   |      X------------>|->|->|       |  |  Prepare(1)
   |      |<------------X--X--X       |  |  Promise(1,{null, null, null})
   |      |             |  |  |       |  |
   |      |             |  |  |       |  |  !! Leader fails during broadcast !!
   |      X------------>|  |  |       |  |  Accept!(1,Va)
   |      !             |  |  |       |  |
   |         |          |  |  |       |  |  !! NEW LEADER !!
   |         X--------->|->|->|       |  |  Prepare(2)
   |         |<---------X--X--X       |  |  Promise(2,{null, null, null})
   |         X--------->|->|->|       |  |  Accept!(2,V)
   |         |<---------X--X--X------>|->|  Accepted(2,V)
   |<--------------------------------X--X  Response
   |         |          |  |  |       |  |
```

The most complex case is when multiple Proposers believe themselves to be Leaders. For instance the current leader may fail and later recover, but the other Proposers have already re-elected a new leader. The recovered leader has not learned this yet and attempts to begin a round in conflict with the current leader.

Message flow: Basic Paxos, dueling Proposers

(one instance, four unsuccessful rounds)

```
Client   Proposer        Acceptor     Learner
   |       |             |  |  |       |  |
   X----->|             |  |  |       |  |  Request
   |       X------------>|->|->|       |  |  Prepare(1)
   |       |<------------X--X--X       |  |  Promise(1,{null,null,null})
   |       !             |  |  |       |  |  !! LEADER FAILS
   |          |          |  |  |       |  |  !! NEW LEADER (knows last number was 1)
   |          X--------->|->|->|       |  |  Prepare(2)
   |          |<---------X--X--X       |  |  Promise(2,{null,null,null})
   |       |  |          |  |  |       |  |  !! OLD LEADER recovers
   |       |  |          |  |  |       |  |  !! OLD LEADER tries 2, denied
   |       X------------>|->|->|       |  |  Prepare(2)
   |       |<------------X--X--X       |  |  Nack(2)
   |       |  |          |  |  |       |  |  !! OLD LEADER tries 3
   |       X------------>|->|->|       |  |  Prepare(3)
   |       |<------------X--X--X       |  |  Promise(3,{null,null,null})
   |       |  |          |  |  |       |  |  !! NEW LEADER proposes, denied
   |       |  X--------->|->|->|       |  |  Accept!(2,Va)
   |       |  |<---------X--X--X       |  |  Nack(3)
   |       |  |          |  |  |       |  |  !! NEW LEADER tries 4
   |       |  X--------->|->|->|       |  |  Prepare(4)
   |       |  |<---------X--X--X       |  |  Promise(4,{null,null,null})
   |       |  |          |  |  |       |  |  !! OLD LEADER proposes, denied
   |       X------------>|->|->|       |  |  Accept!(3,Vb)
   |       |<------------X--X--X       |  |  Nack(4)
   |       |  |          |  |  |       |  |  ... and so on ...
```

Note that Basic Paxos is a stripped down model of the core of every version of Paxos used to prove the correctness of family of Paxos algorithms, as shown in the paper Paxos Made Simple.[17]

Multi-Paxos

If each command is the result of a single instance of the Basic Paxos protocol a significant amount of overhead would result. The paper Paxos Made Simple[17] defines Paxos to be what is commonly called "Multi-Paxos" which in steady state uses a distinguished leader to coordinate an infinite stream of commands. A typical deployment of Paxos uses a continuous stream of agreed values acting as commands to update a distributed state machine.

If the leader is relatively stable, phase 1 becomes unnecessary. Thus, it is possible to skip phase 1 for future instances of the protocol with the same leader.

To achieve this, the instance number `I` is included along with each value. Multi-Paxos reduces the failure-free message delay (proposal to learning) from 4 delays to 2 delays.

Message flow: Multi-Paxos, start

(first instance with new leader)

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  | --- First Request ---
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(N)
   |         |<---------X--X--X       |  |  Promise(N,I,{Va,Vb,Vc})
   |         X--------->|->|->|       |  |  Accept!(N,I,Vm)
   |         |<---------X--X--X------>|->|  Accepted(N,I,Vm)
   |<----------------------------------X--X  Response
   |         |          |  |  |       |  |
```

Vm = highest of (Va, Vb, Vc)

Message flow: Multi-Paxos, steady-state

(subsequent instances with same leader)

```
Client   Proposer       Acceptor     Learner
   |         |          |  |  |       |  |  --- Following Requests ---
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Accept!(N,I+1,W)
   |         |<---------X--X--X------>|->|  Accepted(N,I+1,W)
   |<----------------------------------X--X  Response
   |         |          |  |  |       |  |
```

Typical Multi-Paxos Collapsed Roles deployment

The most common deployment of the Paxos family is Multi-Paxos,[16] specialized for participating processors to each be Proposers, Acceptors and Learners. The message flow with roles collapsed may be optimized as depicted here:

Message flow: Multi-Paxos Collapsed Roles, start

(first instance with new leader)

```
Client      Servers
   |         |  |  | --- First Request ---
   X-------->|  |  |  Request
   |         X->|->|  Prepare(N)
   |         |<-X--X  Promise(N,I,{Va,Vb})
   |         X->|->|  Accept!(N,I,Vn)
   |         |<-X--X  Accepted(N,I)
   |<--------X  |  |  Response
   |         |  |  |
```

Message flow: Multi-Paxos Collapsed Roles, steady state

(subsequent instances with same leader)

```
Client      Servers
   X-------->|  |  |  Request
   |         X->|->|  Accept!(N,I+1,W)
   |         |<-X--X  Accepted(N,I+1)
   |<--------X  |  |  Response
   |         |  |  |
```

Optimizations

A number of optimizations reduce message complexity and size. These optimizations are summarized below:

> "We can save messages at the cost of an extra message delay by having a single distinguished learner that informs the other learners when it finds out that a value has been chosen. Acceptors then send *Accepted* messages only to the distinguished learner. In most applications, the roles of leader and distinguished learner are performed by the same processor.

"A leader can send its *Prepare* and *Accept!* messages just to a quorum of acceptors. As long as all acceptors in that quorum are working and can communicate with the leader and the learners, there is no need for acceptors not in the quorum to do anything.

"Acceptors do not care what value is chosen. They simply respond to *Prepare* and *Accept!* messages to ensure that, despite failures, only a single value can be chosen. However, if an acceptor does learn what value has been chosen, it can store the value in stable storage and erase any other information it has saved there. If the acceptor later receives a *Prepare* or *Accept!* message, instead of performing its Phase1b or Phase2b action, it can simply inform the leader of the chosen value.

"Instead of sending the value v, the leader can send a hash of v to some acceptors in its *Accept!* messages. A learner will learn that v is chosen if it receives *Accepted* messages for either v or its hash from a quorum of acceptors, and at least one of those messages contains v rather than its hash. However, a leader could receive *Promise* messages that tell it the hash of a value v that it must use in its Phase2a action without telling it the actual value of v. If that happens, the leader cannot execute its Phase2a action until it communicates with some process that knows v."[7]

"A proposer can send its proposal only to the leader rather than to all coordinators. However, this requires that the result of the leader-selection algorithm be broadcast to the proposers, which might be expensive. So, it might be better to let the proposer send its proposal to all coordinators. (In that case, only the coordinators themselves need to know who the leader is.)

"Instead of each acceptor sending *Accepted* messages to each learner, acceptors can send their *Accepted* messages to the leader and the leader can inform the learners when a value has been chosen. However, this adds an extra message delay.

"Finally, observe that phase 1 is unnecessary for round 1 .. The leader of round 1 can begin the round by sending an *Accept!* message with any proposed value."[8]

Cheap Paxos

Cheap Paxos extends Basic Paxos to tolerate F failures with F+1 main processors and F auxiliary processors by dynamically reconfiguring after each failure.

This reduction in processor requirements comes at the expense of liveness; if too many main processors fail in a short time, the system must halt until the auxiliary processors can reconfigure the system. During stable periods, the auxiliary processors take no part in the protocol.

> "With only two processors p and q, one processor cannot distinguish failure of the other processor from failure of the communication medium. A third processor is needed. However, that third processor does not have to participate in choosing the sequence of commands. It must take action only in case p or q fails, after which it does nothing while either p or q continues to operate the system by itself. The third processor can therefore be a small/slow/cheap one, or a processor primarily devoted to other tasks."[7]

Message flow: Cheap Multi-Paxos

3 main Acceptors, 1 Auxiliary Acceptor, Quorum size = 3, showing failure of one main processor and subsequent reconfiguration

```
{  Acceptors   }
Proposer     Main       Aux    Learner
|            |  |  |     |       |  -- Phase 2 --
X----------->|->|->|     |       |  Accept!(N,I,V)
|            |  |  !     |       |  --- FAIL! ---
|<-----------X--X--------------->|  Accepted(N,I,V)
|            |  |        |       |  -- Failure detected (only 2 accepted) --
X----------->|->|------->|       |  Accept!(N,I,V)  (re-transmit, include Aux)
|<-----------X--X--------X------>|  Accepted(N,I,V)
|            |  |        |       |  -- Reconfigure : Quorum = 2 --
X----------->|->|        |       |  Accept!(N,I+1,W) (Aux not participating)
|<-----------X--X--------------->|  Accepted(N,I+1,W)
|            |  |        |       |
```

Fast Paxos

Fast Paxos generalizes Basic Paxos to reduce end-to-end message delays. In Basic Paxos, the message delay from client request to learning is 3 message delays. Fast Paxos allows 2 message delays, but requires the Client to send its request to multiple destinations.

Intuitively, if the leader has no value to propose, then a client could send an *Accept!* message to the Acceptors directly. The Acceptors would respond as in Basic Paxos, sending *Accepted* messages to the leader and every Learner achieving two message delays from Client to Learner.

If the leader detects a collision, it resolves the collision by sending *Accept!* messages for a new round which are *Accepted* as usual. This coordinated recovery technique requires four message delays from Client to Learner.

The final optimization occurs when the leader specifies a recovery technique in advance, allowing the Acceptors to perform the collision recovery themselves. Thus, uncoordinated collision recovery can occur in three message delays (and only two message delays if all Learners are also Acceptors).

Message flow: Fast Paxos, non-conflicting

```
Client    Leader         Acceptor      Learner
   |         |          |  |  |  |       |  |
   |         X--------->|->|->|->|       |  |  Any(N,I,Recovery)
   |         |          |  |  |  |       |  |
   X------------------->|->|->|->|       |  |  Accept!(N,I,W)
   |         |<---------X--X--X--X------>|->|  Accepted(N,I,W)
   |<-----------------------------------X--X  Response(W)
   |         |          |  |  |  |       |  |
```

Message flow: Fast Paxos, conflicting proposals

Conflicting proposals with uncoordinated recovery. Note: the protocol does not specify how to handle the dropped client request.

```
Client   Leader      Acceptor     Learner
 |  |      |         |  |  |  |      |  |
 |  |      X------->|->|->|->|      |  |  Any(N,I,Recovery)
 |  |      |         |  |  |  |      |  |
 |  |      |         |  |  |  |      |  |  !! Concurrent conflicting proposals
 |  |      |         |  |  |  |      |  |  !!   received in different order
 |  |      |         |  |  |  |      |  |  !!   by the Acceptors
 |  X--------------?|-?|-?|-?|      |  |  Accept!(N,I,V)
```

```
X-----------------?|-?|-?|-?|      |   |  Accept!(N,I,W)
|  |      |         |  |  |  |      |   |
|  |      |         |  |  |  |      |   |  !! Acceptors disagree on value
|  |      |<-------X--X->|->|----->|->|  Accepted(N,I,V)
|  |      |<-------|<-|<-X--X----->|->|  Accepted(N,I,W)
|  |      |         |  |  |  |      |   |
|  |      |         |  |  |  |      |   |  !! Detect collision & recover
|  |      |<-------X--X--X--X----->|->|  Accepted(N+1,I,W)
|<--------------------------------X--X  Response(W)
|  |      |         |  |  |  |      |   |
```

Message flow: Fast Paxos, collapsed roles

(merged Acceptor/Learner roles)

```
Client         Servers
|  |          |  |  |  |
|  |         X->|->|->|  Any(N,I,Recovery)
|  |          |  |  |  |
|  |          |  |  |  |  !! Concurrent conflicting proposals
|  |          |  |  |  |  !!   received in different order
|  |          |  |  |  |  !!   by the Servers
|  X--------?|-?|-?|-?|  Accept!(N,I,V)
X-----------?|-?|-?|-?|  Accept!(N,I,W)
|  |          |  |  |  |
|  |          |  |  |  |  !! Servers disagree on value
|  |         X--X->|->|  Accepted(N,I,V)
|  |          |<-|<-X--X  Accepted(N,I,W)
|  |          |  |  |  |
|  |          |  |  |  |  !! Detect collision & recover
|<-----------X--X--X--X  Response(W)
|  |          |  |  |  |
```

Generalized Paxos

Generalized consensus explores the relationship between the operations of the replicated state machine and the consensus protocol that implements it. The main discovery involves optimizations of Paxos when conflicting proposals could be applied in any order. i.e., when the proposed operations are commutative operations for the state machine. In such cases, the conflicting operations can both be accepted, avoiding the delays required for resolving conflicts and re-proposing the rejected operations.

This concept is further generalized into ever-growing sequences of commutative operations, some of which are known to be stable (and thus may be executed). The protocol tracks these sequences ensuring that all proposed operations of one sequence are stabilized before allowing any operation non-commuting with them to become stable.

Example

In order to illustrate Generalized Paxos, the example below shows a message flow between two concurrently executing clients and a replicated state machine implementing read/write operations over two distinct registers A and B.

Commutativity Table

	Read(A)	Write(A)	Read(B)	Write(B)
Read(A)		X		
Write(A)	X	X		
Read(B)				X
Write(B)			X	X

Note that X in this table indicates operations which are non-commutative.

A possible sequence of operations :

```
<1:Read(A), 2:Read(B), 3:Write(B), 4:Read(B), 5:Read(A), 6:Write(A)>
```

Since `5:Read(A)` commutes with both `3:Write(B)` and `4:Read(B)`, one possible permutation equivalent to the previous order is the following:

```
<1:Read(A), 2:Read(B), 5:Read(A), 3:Write(B), 4:Read(B), 6:Write(A)>
```

In practice, a commute occurs only when operations are proposed concurrently.

Message flow: Generalized Paxos (example)

Responses not shown. Note: message abbreviations differ from previous message flows due to specifics of the protocol, see [9] for a full discussion.

```
{    Acceptors   }
Client      Leader  Acceptor       Learner
 |  |         |      |  |  |         |  |  !! New Leader Begins Round
 |  |         X----->|->|->|         |  |  Prepare(N)
 |  |         |<-----X--X--X         |  |  Promise(N,null)
 |  |         X----->|->|->|         |  |  Phase2Start(N,null)
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  |  !! Concurrent commuting proposals
 |  X--------?|-----?|-?|-?|         |  |  Propose(ReadA)
 X-----------?|-----?|-?|-?|         |  |  Propose(ReadB)
 |  |         |<-----X-->-->-------->|->|  Accepted(N,<ReadA,ReadB>)
 |  |         |<-----<--X--X-------->|->|  Accepted(N,<ReadB,ReadA>)
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  |  !! No Conflict, both stable
 |  |         |      |  |  |         |  |  V = <ReadA, ReadB>
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  |  !! Concurrent conflicting proposals
 X-----------?|-----?|-?|-?|         |  |  Propose(WriteB)
 |  X--------?|-----?|-?|-?|         |  |  Propose(ReadB)
 |  |         |      |  |  |         |  |
 |  |         |<-----X------         |  |  Accepted(N,V.<WriteB,ReadB>)
 |  |         |<--------X---         |  |  Accepted(N,V.<ReadB,WriteB>)
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  |  !! Conflict detected at the leader.
 |  |         |      |  |  |         |  |
 |  |         X----->|->|->|         |  |  Prepare(N+1)
 |  |         |<-----X------         |  |  Promise(N+1, N, V.<WriteB,ReadB>)
 |  |         |<--------X---         |  |  Promise(N+1, N, V.<ReadB, WriteB>)
 |  |         |<-----------X         |  |  Promise(N+1, N, V)
 |  |         |      |  |  |         |  |
 |  |         |      |  |  |         |  |
 |  |         X----->|->|->|         |  |  Phase2Start(N+1,V.<WriteB,ReadB>)
 |  |         |<-----X--X--X-------->|->|  Accepted(N+1,V.<WriteB,ReadB>)
```

```
|  |         |      |  |  |         |  |
|  |         |      |  |  |         |  |  !! New stable sequence
|  |         |      |  |  |         |  |  U = <ReadA, ReadB>, WriteB, ReadB>
|  |         |      |  |  |         |  |
|  |         |      |  |  |         |  |  !! More conflicting proposals
X----------?|-----?|-?|-?|         |  |  Propose(WriteA)
|  X--------?|-----?|-?|-?|         |  |  Propose(ReadA)
|  |         |      |  |  |         |  |
|  |         |      |  |  |         |  |  !! This time spontaneously ordered by the network
|  |         [<-----X--X----------->|->|  Accepted(N+1,U.<WriteA, ReadA>)
|  |         |      |  |  |         |  |
```

Performance

The above message flow shows us that Generalized Paxos can leverage operation semantics to avoid collisions when the spontaneous ordering of the network fails. This allows the protocol to be in practice quicker than Fast Paxos. However, when a collision occurs, Generalized Paxos needs two additional round trips to recover. This situation is illustrated with operations WriteB and ReadB in the above schema.

In the general case, such round trips are unavoidable and comes from the fact that multiple commands might be accepted during a round. This makes the protocol more expensive than Paxos when conflicts are frequent. Hopefully two possible refinements of Generalized Paxos are possible to improve recovery time.[18]

- First, if the coordinator is part of every quorum of acceptors (round N is said *centered*), then to recover at round N+1 from a collision at round N, the coordinator skip phase 1 and proposes at phase 2 the sequence it accepted last during round N. This reduces the cost of recovery to a single round trip.
- Second, if both rounds N and N+1 are centered around the same coordinator, when an acceptor detects a collision at round N, it proposes at round N+1 a sequence suffixing both (i) the sequence accepted at round N by the coordinator and (ii) the greatest non-conflicting prefix it accepted at round N. For instance, if the coordinator and the acceptor accepted respectively at round N <WriteB, ReadB> and <ReadB, ReadA> , the acceptor will spontaneously accept <WriteB, ReadB, ReadA> at round N+1. With this variation, the cost of recovery is a single message delay which is obviously optimal.

Byzantine Paxos

Paxos may also be extended to support arbitrary failures of the participants, including lying, fabrication of messages, collusion with other participants, selective non-participation, etc. These types of failures are called Byzantine failures, after the solution popularized by Lamport.[19]

Byzantine Paxos[8][10] adds an extra message (Verify) which acts to distribute knowledge and verify the actions of the other processors:

Message flow: Byzantine Multi-Paxos, steady state

```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Accept!(N,I,V)
   |         |          X<>X<>X       |  |  Verify(N,I,V) - BROADCAST
   |         |<---------X--X--X------>|->|  Accepted(N,V)
```

```
   |<--------------------------------X--X  Response(V)
   |         |          | | |       | |
```

Fast Byzantine Paxos removes this extra delay, since the client sends commands directly to the Acceptors.[8]

Note the *Accepted* message in Fast Byzantine Paxos is sent to all Acceptors and all Learners, while Fast Paxos sends *Accepted* messages only to Learners):

Message flow: Fast Byzantine Multi-Paxos, steady state

```
Client    Acceptor     Learner
   |       | | |        | |
   X----->|->|->|        | |  Accept!(N,I,V)
   |       X<>X<>X------>|->|  Accepted(N,I,V) - BROADCAST
   |<-------------------X--X  Response(V)
   |       | | |        | |
```

The failure scenario is the same for both protocols; Each Learner waits to receive F+1 identical messages from different Acceptors. If this does not occur, the Acceptors themselves will also be aware of it (since they exchanged each other's messages in the broadcast round), and correct Acceptors will re-broadcast the agreed value:

Message flow: Fast Byzantine Multi-Paxos, failure

```
Client    Acceptor     Learner
   |       | |  !       | |  !! One Acceptor is faulty
   X----->|->|->!       | |  Accept!(N,I,V)
   |       X<>X<>X------>|->|  Accepted(N,I,{V,W}) - BROADCAST
   |       | |  !       | |  !! Learners receive 2 different commands
   |       | |  !       | |  !! Correct Acceptors notice error and choose
   |       X<>X<>X------>|->|  Accepted(N,I,V) - BROADCAST
   |<-------------------X--X  Response(V)
   |       | |  !       | |
```

Production use of Paxos

- The Petal project from DEC SRC was likely the first system to use Paxos, in this case for widely replicated global information (e.g., which machines are in the system).[20]
- Google uses the Paxos algorithm in their Chubby distributed lock service in order to keep replicas consistent in case of failure. Chubby is used by BigTable which is now in production in Google Analytics and other products.
- The Infinit peer-to-peer file system relies on Paxos to maintain consistency among replicas while allowing for quorums to evolve in size.
- Google Spanner and Megastore use the Paxos algorithm internally.
- The OpenReplica replication service (http://openreplica.org) uses Paxos to maintain replicas for an open access system that enables users to create fault-tolerant objects. It provides high performance through concurrent rounds and flexibility through dynamic membership changes.
- IBM supposedly uses the Paxos algorithm in their IBM SAN Volume Controller product to implement a general purpose fault-tolerant virtual machine used to run the configuration and control components of the storage virtualization services offered by the cluster.

- Microsoft uses Paxos in the Autopilot cluster management service (http://research.microsoft.com/pubs/64604/osr2007.pdf) from Bing.
- WANdisco have implemented Paxos within their DConE active-active replication technology.[21]
- XtreemFS uses a Paxos-based lease negotiation algorithm for fault-tolerant and consistent replication of file data and metadata.[22]
- Heroku uses Doozerd (https://github.com/ha/doozerd) which implements Paxos for its consistent distributed data store.
- Ceph uses Paxos as part of the monitor processes to agree which OSDs are up and in the cluster.
- The Clustrix distributed SQL database uses Paxos for distributed transaction resolution (http://docs.clustrix.com/display/CLXDOC/Consistency,+Fault+Tolerance,+and+Availability).
- Neo4j HA graph database implements Paxos, replacing Apache ZooKeeper from v1.9
- VMware NSX Controller uses Paxos-based algorithm within NSX Controller cluster.
- Amazon Web Services uses the Paxos algorithm extensively to power its platform.[23]
- Nutanix implements the Paxos algorithm in Cassandra for metadata (http://stevenpoitras.com/the-nutanix-bible/#scalable_metadata).
- Apache Mesos uses Paxos algorithm for its replicated log (http://mesos.apache.org/blog/mesos-0-17-0-released-featuring-autorecovery/) coordination.
- Windows Fabric used by many of the Azure services make use of the paxos algorithm for replication between nodes in a cluster
- Oracle NoSQL Database leverages Paxos-based automated fail-over election process in the event of a master replica node failure to minimize downtime.[24]

See also

- Blockchain database, a distributed consensus model used in Bitcoin
- Chandra–Toueg consensus algorithm
- Distributed algorithm
- Vsync, a library implementing Paxos in a virtually synchronous group model
- Raft consensus algorithm
- State machine replication

References

1. Pease, Marshall; Shostak, Robert; Lamport, Leslie (April 1980). "Reaching Agreement in the Presence of Faults". *Journal of the Association for Computing Machinery*. **27** (2): 228–234. doi:10.1145/322186.322188. Retrieved 2007-02-02.
2. Lamport, Leslie (July 1978). "Time, Clocks and the Ordering of Events in a Distributed System". *Communications of the ACM*. **21** (7): 558–565. doi:10.1145/359545.359563. Retrieved 2007-02-02.
3. Schneider, Fred (1990). "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial" (PDF). *ACM Computing Surveys*. **22**: 299–319. doi:10.1145/98163.98167.
4. Leslie Lamport's history of the paper (http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#lamport-paxos)
5. Lamport, Leslie (May 1998). "The Part-Time Parliament". *ACM Transactions on Computer Systems*. **16** (2): 133–169. doi:10.1145/279227.279229. Retrieved 2007-02-02.
6. Fischer, M.; Lynch, N.; Paterson, M. (April 1985). "Impossibility of distributed consensus with one faulty process". *Journal of the ACM*. **32** (2): 374–382. doi:10.1145/3149.214121.

7. Lamport, Leslie; Massa, Mike (2004). *Cheap Paxos*. Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004).
8. Lamport, Leslie (2005). "Fast Paxos".
9. Lamport, Leslie (2005). "Generalized Consensus and Paxos".
10. Castro, Miguel (2001). "Practical Byzantine Fault Tolerance" (PDF).
11. Dwork, Cynthia; Lynch, Nancy; Stockmeyer, Larry (April 1988). "Consensus in the Presence of Partial Synchrony" (PDF). *Journal of the ACM*. **35** (2): 288–323. doi:10.1145/42282.42283.
12. Oki, Brian; Liskov, Barbara (1988). *Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems*. PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of Distributed Computing. pp. 8–17. doi:10.1145/62546.62549.
13. Birman, Kenneth; Joseph, Thomas (February 1987). "Reliable Communication in the Presence of Failures". *ACM Transactions on Computer Systems*. **5**: 47–76. doi:10.1145/7351.7478.
14. Lamport, Leslie; Malkhi, Dahlia; Zhou, Lidong (March 2010). "Reconfiguring a State Machine". *SIGACT News*. **41** (1): 63–73. doi:10.1145/1753171.1753191.
15. Lamport, Leslie (2004). "Lower Bounds for Asynchronous Consensus".
16. Chandra, Tushar; Griesemer, Robert; Redstone, Joshua (2007). "Paxos Made Live – An Engineering Perspective". *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*.
17. Lamport, Leslie (2001). Paxos Made Simple (http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#paxos-simple) *ACM SIGACT News (Distributed Computing Column) 32*, 4 (Whole Number 121, December 2001) 51-58.
18. Pierre, Sutra; Marc, Shapiro (2011). *Fast Genuine Generalized Consensus* (PDF). SRDS'11: 30th IEEE Symposium on Reliable Distributed Systems.
19. Lamport, Leslie; Shostak, Robert; Pease, Marshall (July 1982). "The Byzantine Generals Problem". *ACM Transactions on Programming Languages and Systems*. **4** (3): 382–401. doi:10.1145/357172.357176. Retrieved 2007-02-02.
20. Edward Lee and Chandramohan Thekkath. "Petal: Distributed Virtual Disks" (http://dl.acm.org/citation.cfm?id=237157). ASPLOS VII, 1996.
21. Aahlad et al. (2011). "The Distributed Coordination Engine (DConE)" (https://www.wandisco.com/get?f=documentation%2Fwhitepapers%2FWANdisco_DConE_White_Paper.pdf). WANdisco white paper.
22. Kolbeck, Björn; Högqvist, Mikael; Stender, Jan; Hupfeld, Felix (2011). "Flease - Lease Coordination without a Lock Server" (http://www.xtreemfs.org/publications/flease_paper_ipdps.pdf). 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011).
23. Vermeulen, Al; Sivasubramanian, Swami (2014). "Under the Covers of AWS: Core Distributed Systems Primitives That Power Our Platform" (http://www.slideshare.net/AmazonWebServices/spot302-under-the-covers-of-aws-core-distributed-systems-primitives-that-power-our-platform-aws-reinvent-2014). AWS re:invent 2014.
24. "Oracle NoSQL Database 12.1.4.0.9, Community Edition Data Sheet" (http://www.oracle.com/technetwork/database/database-technologies/nosqldb/documentation/datasheet-oracle-nosql-db-ce-1876735.pdf). Oracle Data Sheet (2016).

External links

- erlpaxos, Paxos by Erlang (http://code.google.com/p/erlpaxos/)
- FTFile: Fault Tolerant File library (http://ftfile.rodia.pl/)
- Google Whitepaper: Bigtable A Distributed Storage System for Structured Data (http://research.google.com/archive/bigtable.html)
- Google Whitepaper: Chubby Distributed Lock Service (http://research.google.com/archive/chubby.html)
- HA Clustering with Neo4j (http://de.slideshare.net/jexp/new-neo4j-auto-ha-cluster)
- HT-Paxos (http://arxiv.org/abs/1407.1237)
- Vsync library (the SafeSend primitive is a free, open source implementation of Paxos) (http://vsync.codeplex.com)

- JBP - Java Byzantine Paxos (http://www.navigators.di.fc.ul.pt/software/jitt/jbp.html)
- Leslie Lamport's home page (http://www.lamport.org/)
- libpaxos, a collection of open source implementations of the Paxos algorithm (http://libpaxos.sourceforge.net/)
- libpaxos-cpp, a C++ implementation of the paxos distributed consensus algorithm (http://www.leonmergen.com/libpaxos-cpp/)
- Mencius - Circular rotating Paxos for geo-distributed systems (http://www.usenix.org/event/osdi08/tech/full_papers/mao/mao.pdf)
- OpenReplica Open Replication Service (http://openreplica.org)
- paxos - Straight-forward paxos implementation in Python & Java (https://github.com/cocagne/paxos/)
- *Consensus on Transaction Commit (https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2003-96.pdf)*, Jim Gray and Leslie Lamport, January 1, 2004.
- *Paxos Made Simple (http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf)*, Leslie Lamport, November 1, 2001.
- Revisiting the Paxos Algorithm (http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.5607)
- Survey of Paxos Algorithms (2007) (http://www.fractalscape.org/2007/10/01/paxos-family.html)
- WANdisco - Active-Active Replication solutions for Hadoop, Subversion & GIT (http://www.wandisco.com/)
- PhxPaxos - C++ Paxos library that has been used in Wechat production environment (https://github.com/tencent-wechat/phxpaxos)
- [1] (https://dev.mysql.com/doc/refman/5.7/en/group-replication-background.html) MySQL Group Replication

Retrieved from "https://en.wikipedia.org/w/index.php?title=Paxos_(computer_science)&oldid=764100648"

Categories: Distributed algorithms | Fault-tolerant computer systems

---

- This page was last modified on 7 February 2017, at 02:30.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

September 1995

Classic of the month

Reflections on Trusting Trust

Ken Thompson

Reprinted from *Communication of the ACM*, Vol. 27, No. 8, August 1984, pp. 761-763. Copyright © 1984, Association for Computing Machinery, Inc. Also appears in *ACM Turing Award Lectures: The First Twenty Years 1965-1985* Copyright © 1987 by the ACM press and *Computers Under Attack: Intruders, Worms, and Viruses* Copyright © 1990 by the ACM press.

This is a digitized copy derived from an ACM copyrighted work. It is not guaranteed to be an accurate copy of the author's original work.

Introduction

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX swept into popularity with an industry-wide change from central main frames to autonomous minis. I suspect that Daniel Bobrow (1) would be here instead of me if he could not afford a PDP-10 and ad had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

Stage I

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that

three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

```
char s[ ] = {
        '\t',
        '0',
        '\n',
        '{',
        ':',
        '\n',
        '\n',
        '/',
        '*',
        '\n',
        (213 lines deleted)
        0
};

/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end.
 */

main( )
{
        int i;

        printf("char\ts[ ] = {\n");
        for(i=0; s[i]; i++)
                printf("\t%d, \n", s[i]);
        printf("%s", s);
}
Here are some simple transliterations to allow
     a non-C programmer to read this code.
=       assignment
==      equal to .EQ.
!=      not equal to .NE.
++      increment
'x'     single character constant
"xxx"   multiple character string
%d      format to convert to decimal
%s      format to convert to string
\t      tab character
\n      newline character
```

FIGURE 1

Figure I shows a self-reproducing program in the C programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: (I) This program can be easily written by another program. (2) This pro-

gram can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm. In the example, even the comment is reproduced.

Stage II

The C compiler is written in C. What I am about to describe is one of many "chicken and egg" problems that arise when compilers are written in their own language. In this ease, I will use a specific example from the C compiler.

C allows a string construct to specify an initialized character array. The individual characters in the string can be escaped to represent unprintable characters. For example,

"Hello world\n"

represents a string with the character "\n," representing the new line character.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
...
```

FIGURE 2

Figure 2 is an idealization of the code in the C compiler that interprets the character escape sequence. This is an amazing piece of code. It "knows" in a completely portable way what character code is compiled for a new line in any character set. The act of knowing then allows it to recompile itself, thus perpetuating the knowledge.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return('\v');
...
```

FIGURE 3

Suppose we wish to alter the C compiler to include the sequence "\v" to represent the vertical tab character. The extension to Figure 2 is obvious and is presented in Figure 3. We then recompile the C compiler, but we get a diagnostic. Obviously, since the binary version of the compiler does not know about "\v," the source is not legal C. We must "train" the compiler. After it "knows" what "\v" means, then our new change will

become legal C. We look up on an ASCII chart that a vertical tab is decimal 11. We alter our source to look like Figure 4. Now the old compiler accepts the new source. We install the resulting binary as the new official C compiler and now we can write the portable version the way we had it in Figure 3.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return(11);
...
```

FIGURE 4

This is a deep concept. It is as close to a "learning" program as I have seen. You simply tell it once, then you can use this self-referencing definition.

Stage III

```
compile(s)
char *s;
{
        ...
}
```

FIGURE 5

Again, in the C compiler, Figure 5 represents the high-level control of the C compiler where the routine "compile" is called to compile the next line of source. Figure 6 shows a simple modification to the compiler that will deliberately miscompile source whenever a particular pattern is matched. If this were not deliberate, it would be called a compiler "bug." Since it is deliberate, it should be called a "Trojan horse."

```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

FIGURE 6

The actual bug I planted in the compiler would match code in the UNIX "login" command. The replacement code would miscompile the login command so that it would accept either the intended encrypted password or a particular known password. Thus if this code were installed in binary and the binary were used to

compile the login command, I could log into that system as any user.

Such blatant code would not go undetected for long. Even the most casual perusal of the source of the C compiler would raise suspicions.

```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        ...
}
```

FIGURE 7

The final step is represented in Figure 7. This simply adds a second Trojan horse to the one that already exists. The second pattern is aimed at the C compiler. The replacement code is a Stage I self-reproducing program that inserts both Trojan horses into the compiler. This requires a learning phase as in the Stage II example. First we compile the modified source with the normal C compiler to produce a bugged binary. We install this binary as the official C. We can now remove the bugs from the source of the compiler and the new binary will reinsert the bugs whenever it is compiled. Of course, the login command will remain bugged with no trace in source anywhere.

Moral

The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well installed microcode bug will be almost impossible to detect.

After trying to convince you that I cannot be trusted, I wish to moralize. I would like to criticize the press in its handling of the "hackers," the 414 gang, the Dalton gang, etc. The acts performed by these kids are vandalism at best and probably trespass and theft at worst. It is only the inadequacy of the criminal code that saves the hackers from very serious prosecution. The companies that are vulnerable to this activity (and most large companies are very vulnerable) are pressing hard to update the criminal code. Unauthorized access to computer systems is already a serious crime in a few states and is currently being addressed in many more state legislatures as well as Congress.

There is an explosive situation brewing. On the one hand, the press, television, and movies make heroes of vandals by calling them whiz kids. On the other hand, the acts performed by these kids will soon be punishable by years in prison.

I have watched kids testifying before Congress. It is clear that they are completely unaware of the seriousness of their acts. There is obviously a cultural gap. The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's

door is unlocked. The press must learn that misguided use of a computer is no more amazing than drunk driving of an automobile.

Acknowledgment

I first read of the possibility of such a Trojan horse in an Air Force critique (4) of the security of an early implementation of Multics. I can- not find a more specific reference to this document. I would appreciate it if anyone who can supply this reference would let me know.

References

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time-sharing system for the PDP-IO. Commun. ACM 15, 3 (Mar. 1972), 135-143.
2. Kernighan, B.W., and Ritchie, D.M. The C Programming Language. Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. Commun. ACM 17, 7(July 1974), 365-375.
4. 4. Unknown Air Force Document.

The Multikernel: A new OS architecture for scalable multicore systems

Andrew Baumann[*], Paul Barham[†], Pierre-Evariste Dagand[‡], Tim Harris[†], Rebecca Isaacs[†], Simon Peter[*], Timothy Roscoe[*], Adrian Schüpbach[*], and Akhilesh Singhania[*]

[*] Systems Group, ETH Zurich [†] Microsoft Research, Cambridge [‡] ENS Cachan Bretagne

ABSTRACT

Commodity computer systems contain more and more processor cores and exhibit increasingly diverse architectural tradeoffs, including memory hierarchies, interconnects, instruction sets and variants, and IO configurations. Previous high-performance computing systems have scaled in specific cases, but the dynamic nature of modern client and server workloads, coupled with the impossibility of statically optimizing an OS for all workloads and hardware variants pose serious challenges for operating system structures.

We argue that the challenge of future multicore hardware is best met by embracing the networked nature of the machine, rethinking OS architecture using ideas from distributed systems. We investigate a new OS structure, the *multikernel*, that treats the machine as a network of independent cores, assumes no inter-core sharing at the lowest level, and moves traditional OS functionality to a distributed system of processes that communicate via message-passing.

We have implemented a multikernel OS to show that the approach is promising, and we describe how traditional scalability problems for operating systems (such as memory management) can be effectively recast using messages and can exploit insights from distributed systems and networking. An evaluation of our prototype on multicore systems shows that, even on present-day machines, the performance of a multikernel is comparable with a conventional OS, and can scale better to support future hardware.

Categories and Subject Descriptors: D.4.7 [Operating Systems]: Organization and Design

General Terms: Design, Experimentation, Performance

Keywords: Scalability, multicore processors, message passing

1. INTRODUCTION

Computer hardware is changing and diversifying faster than system software. A diverse mix of cores, caches, interconnect links, IO devices and accelerators, combined with increasing core counts, leads to substantial scalability and correctness challenges for OS designers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.
Copyright 2009 ACM 978-1-60558-752-3/09/10 ...$10.00.

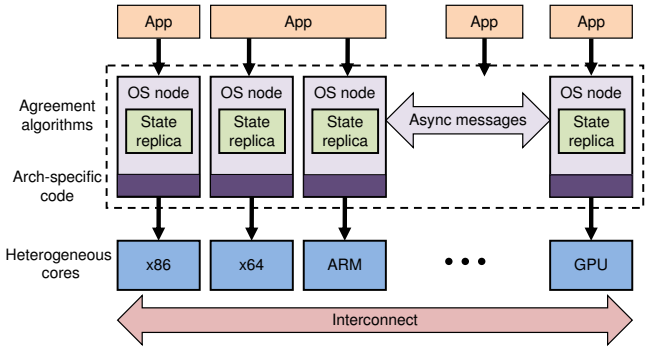


Figure 1: The multikernel model.

Such hardware, while in some regards similar to earlier parallel systems, is new in the general-purpose computing domain. We increasingly find multicore systems in a variety of environments ranging from personal computing platforms to data centers, with workloads that are less predictable, and often more OS-intensive, than traditional high-performance computing applications. It is no longer acceptable (or useful) to tune a general-purpose OS design for a particular hardware model: the deployed hardware varies wildly, and optimizations become obsolete after a few years when new hardware arrives.

Moreover, these optimizations involve tradeoffs specific to hardware parameters such as the cache hierarchy, the memory consistency model, and relative costs of local and remote cache access, and so are not portable between different hardware types. Often, they are not even applicable to future generations of the same architecture. Typically, because of these difficulties, a scalability problem must affect a substantial group of users before it will receive developer attention.

We attribute these engineering difficulties to the basic structure of a shared-memory kernel with data structures protected by locks, and in this paper we argue for rethinking the structure of the OS as a distributed system of functional units communicating via explicit messages. We identify three design principles: (1) make all inter-core communication explicit, (2) make OS structure hardware-neutral, and (3) view state as replicated instead of shared.

The model we develop, called a *multikernel* (Figure 1), is not only a better match to the underlying hardware (which is networked, heterogeneous, and dynamic), but allows us to apply insights from distributed systems to the problems of scale, adaptivity, and diversity in operating systems for future hardware.

Even on present systems with efficient cache-coherent shared memory, building an OS using message-based rather than shared-

data communication offers tangible benefits: instead of sequentially manipulating shared data structures, which is limited by the latency of remote data access, the ability to pipeline and batch messages encoding remote operations allows a single core to achieve greater throughput and reduces interconnect utilization. Furthermore, the concept naturally accommodates heterogeneous hardware.

The contributions of this work are as follows:

- We introduce the multikernel model and the design principles of explicit communication, hardware-neutral structure, and state replication.
- We present a multikernel, Barrelfish, which explores the implications of applying the model to a concrete OS implementation.
- We show through measurement that Barrelfish satisfies our goals of scalability and adaptability to hardware characteristics, while providing competitive performance on contemporary hardware.

In the next section, we survey trends in hardware which further motivate our rethinking of OS structure. In Section 3, we introduce the multikernel model, describe its principles, and elaborate on our goals and success criteria. We describe Barrelfish, our multikernel implementation, in Section 4. Section 5 presents an evaluation of Barrelfish on current hardware based on how well it meets our criteria. We cover related and future work in Sections 6 and 7, and conclude in Section 8.

2. MOTIVATIONS

Most computers built today have multicore processors, and future core counts will increase [12]. However, commercial multiprocessor servers already scale to hundreds of processors with a single OS image, and handle terabytes of RAM and multiple 10Gb network connections. Do we need new OS techniques for future multicore hardware, or do commodity operating systems simply need to exploit techniques already in use in larger multiprocessor systems?

In this section, we argue that the challenges facing a general-purpose operating system on future commodity hardware are different from those associated with traditional ccNUMA and SMP machines used for high-performance computing. In a previous paper [8] we argued that single computers increasingly resemble networked systems, and should be programmed as such. We rehearse that argument here, but also lay out additional scalability challenges for general-purpose system software.

2.1 Systems are increasingly diverse

A general-purpose OS today must perform well on an increasingly diverse range of system designs, each with different performance characteristics [60]. This means that, unlike large systems for high-performance computing, such an OS cannot be optimized at design or implementation time for any particular hardware configuration.

To take one specific example: Dice and Shavit show how a reader-writer lock can be built to exploit the shared, banked L2 cache on the Sun Niagara processor [40], using concurrent writes to the same cache line to track the presence of readers [20]. On Niagara this is highly efficient: the line remains in the L2 cache. On a traditional multiprocessor, it is highly inefficient: the line ping-pongs between the readers' caches.

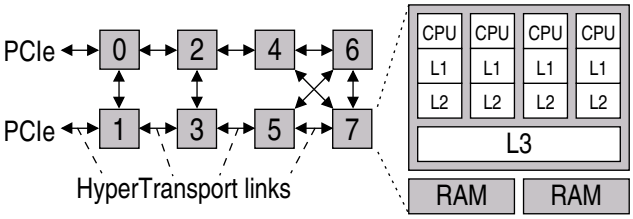


Figure 2: Node layout of an 8×4-core AMD system

This illustrates a general problem: any OS design tied to a particular synchronization scheme (and data layout policy) cannot exploit features of different hardware. Current OS designs optimize for the common hardware case; this becomes less and less efficient as hardware becomes more diverse. Worse, when hardware vendors introduce a design that offers a new opportunity for optimization, or creates a new bottleneck for existing designs, adapting the OS to the new environment can be prohibitively difficult.

Even so, operating systems are forced to adopt increasingly complex optimizations [27, 46, 51, 52, 57] in order to make efficient use of modern hardware. The recent scalability improvements to Windows7 to remove the dispatcher lock touched 6000 lines of code in 58 files and have been described as "heroic" [58]. The Linux read-copy update implementation required numerous iterations due to feature interaction [50]. Backporting receive-side-scaling support to Windows Server 2003 caused serious problems with multiple other network subsystems including firewalls, connection-sharing and even Exchange servers[1].

2.2 Cores are increasingly diverse

Diversity is not merely a challenge across the range of commodity machines. Within a single machine, cores can vary, and the trend is toward a mix of different cores. Some will have the same instruction set architecture (ISA) but different performance characteristics [34,59], since a processor with large cores will be inefficient for readily parallelized programs, but one using only small cores will perform poorly on the sequential parts of a program [31, 42]. Other cores have different ISAs for specialized functions [29], and many peripherals (GPUs, network interfaces, and other, often FPGA-based, specialized processors) are increasingly programmable.

Current OS designs draw a distinction between general-purpose cores, which are assumed to be homogeneous and run a single, shared instance of a kernel, and peripheral devices accessed through a narrow driver interface. However, we are not the only researchers to see an increasing need for OSes to manage the software running on such cores much as they manage CPUs today [55]. Moreover, core heterogeneity means cores can no longer share a single OS kernel instance, either because the performance tradeoffs vary, or because the ISA is simply different.

2.3 The interconnect matters

Even for contemporary cache-coherent multiprocessors, message-passing hardware has replaced the single shared interconnect [18, 33] for scalability reasons. Hardware thus resembles a message-passing network, as in the interconnect topology of the commodity PC server in Figure 2. While on most current hardware the cache-coherence protocol between CPUs ensures that the OS can continue to safely assume a single shared memory, networking problems

[1] See Microsoft Knowledge Base articles 927168, 927695 and 948496.

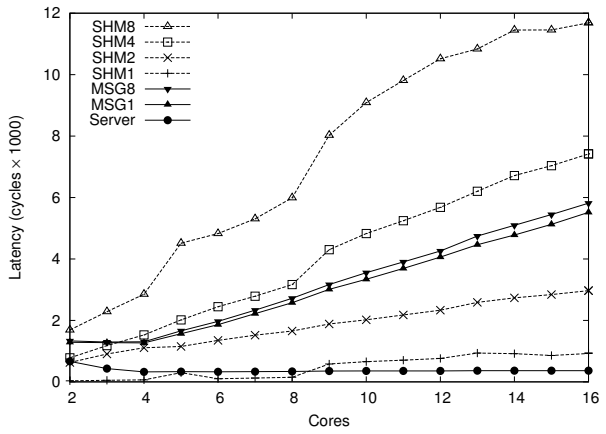


Figure 3: Comparison of the cost of updating shared state using shared memory and message passing.

like routing and congestion are well-known concerns on large-scale multiprocessors, and are now issues in commodity intra-machine interconnects [18]. Future hardware will comprise fewer chips but exhibit network effects inside the chip, such as with ring [38, 61] and mesh networks [68, 70]. The implication is that system software will have to adapt to the inter-core topology, which in turn will differ between machines and become substantially more important for performance than at present.

2.4 Messages cost less than shared memory

In 1978 Lauer and Needham argued that message-passing and shared-memory operating systems are duals, and the choice of one model over another depends on the machine architecture on which the OS is built [43]. Of late, shared-memory systems have been the best fit for PC hardware in terms of both performance and good software engineering, but this trend is reversing. We can see evidence of this by an experiment that compares the costs of updating a data structure using shared memory with the costs using message passing. The graph in Figure 3 plots latency against number of cores for updates of various sizes on the 4×4-core AMD system (described in Section 4.1).

In the shared memory case, threads pinned to each core directly update the same small set of memory locations (without locking) and the cache-coherence mechanism migrates data between caches as necessary. The curves labeled *SHM1–8* show the latency per operation (in cycles) for updates that directly modify 1, 2, 4 and 8 shared cache lines respectively. The costs grow approximately linearly with the number of threads and the number of modified cache lines. Although a single core can perform the update operation in under 30 cycles, when 16 cores are modifying the same data it takes almost 12,000 extra cycles to perform each update. All of these extra cycles are spent with the core stalled on cache misses and therefore unable to do useful work while waiting for an update to occur.

In the case of message passing, client threads issue a lightweight remote procedure call [10], (which we assume fits in a 64-byte cache line), to a single server process that performs the update on their behalf. The curves labeled *MSG1* and *MSG8*, show the cost of this *synchronous* RPC to the dedicated server thread. As expected, the cost varies little with the number of modified cache lines since they remain in the server's local cache. Because each request is likely to experience some queuing delay at the server proportional to the number of clients, the *elapsed* time per operation grows linearly with the number of client threads. Despite this, for updates of four or more cache lines, the RPC latency is lower than shared memory access (*SHM4* vs. *MSG8*). Furthermore, with an asynchronous or pipelined RPC implementation, the client processors can avoid stalling on cache misses and are free to perform other operations.

The final curve, labeled *Server*, shows time spent performing each update operation as measured at the server end of the RPC channel. Since it excludes queuing delay, this cost is largely independent of the number of threads (and in fact decreases initially once there are enough outstanding client requests to keep the server 100% utilized). The cache efficiency of the single server is such that it can perform twice as many updates per second as all 16 shared-memory threads combined. The per-operation cost is dominated by message send and receive, and since these costs are symmetric at the client end, we infer that the difference between the *Server* and *MSGn* lines represents the additional cycles that would be available to the client for useful work if it were to use asynchronous messaging.

This example shows scalability issues for cache-coherent shared memory on even a small number of cores. Although current OSes have point-solutions for this problem, which work on specific platforms or software systems, we believe the inherent lack of scalability of the shared memory model, combined with the rate of innovation we see in hardware, will create increasingly intractable software engineering problems for OS kernels.

2.5 Cache coherence is not a panacea

As the number of cores and the subsequent complexity of the interconnect grows, hardware cache-coherence protocols will become increasingly expensive. As a result, it is a distinct possibility that future operating systems will have to handle non-coherent memory [12, 49, 69], or will be able to realize substantial performance gains by bypassing the cache-coherence protocol [70].

It is already the case that programmable peripherals like NICs and GPUs do not maintain cache coherence with CPUs. Furthermore, many multicore processors have already demonstrated the use of non-coherent shared memory [15, 26, 68], and Mattson *et al.* [49] argue that the overhead of cache coherence restricts the ability to scale up to even 80 cores.

2.6 Messages are getting easier

There are legitimate software engineering issues associated with message-based software systems, and one might therefore question the wisdom of constructing a multiprocessor operating system based on a "shared nothing" model as we propose. There are two principal concerns, the first to do with not being able to access shared data, and the second to do with the event-driven programming style that results from asynchronous messaging.

However, the convenience of shared data is somewhat superficial. There are correctness and performance pitfalls when using shared data structures, and in scalable shared-memory programs (particularly high-performance scientific computing applications), expert developers are very careful about details such as lock granularity and how fields are laid out within structures. By fine-tuning code at a low level, one can minimize the cache lines needed to hold the shared data and reduce contention for cache line ownership. This reduces interconnect bandwidth and the number of processor stalls incurred when cache contents are stale.

The same kind of expertise is also applied to make commodity operating systems more scalable. As we have shown above, this

leads to a challenge in evolving the system as tradeoffs change, because the knowledge required for effective sharing of a particular data structure is encoded implicitly in its implementation. Note that this means programmers must think carefully about a shared-memory program *in terms of messages* sent by the cache-coherence protocol in response to loads and stores to data locations.

The second concern with message passing is the resultant "stack ripping" and obfuscation of control flow due to the event-driven nature of such programs. However, traditional monolithic kernels are essentially event-driven systems, even on multiprocessors. OS developers are perhaps more accustomed to thinking in terms of state machines and message handlers than other programmers.

Finally, we note that a message-passing, event-driven programming model is also the norm for many other programming domains, such as graphical user interface programming, some types of network server, and large-scale computation on clusters (where it has completely replaced the "distributed shared virtual memory" paradigm). This has meant that the programmability of message-passing or event-driven systems is an active research area with promising results that seem a natural fit for the multikernel model, such as the Tame/Tamer C++ libraries [41] and the X10 parallel programming language [16]. As the need for multicore programming environments at the user level becomes more pressing, we expect tools like these to support a message-passing abstraction will become widespread.

2.7 Discussion

The architecture of future computers is far from clear but two trends are evident: rising core counts and increasing hardware diversity, both between cores within a machine, and between systems with varying interconnect topologies and performance tradeoffs.

This upheaval in hardware has important consequences for a monolithic OS that shares data structures across cores. These systems perform a delicate balancing act between processor cache size, the likely contention and latency to access main memory, and the complexity and overheads of the cache-coherence protocol. The irony is that hardware is now changing faster than software, and the effort required to evolve such operating systems to perform well on new hardware is becoming prohibitive.

Increasing system and interconnect diversity, as well as core heterogeneity, will prevent developers from optimizing shared memory structures at a source-code level. Sun Niagara and Intel Nehalem or AMD Opteron systems, for example, already require completely different optimizations, and future system software will have to adapt its communication patterns and mechanisms at runtime to the collection of hardware at hand. It seems likely that future general-purpose systems will have limited support for hardware cache coherence, or drop it entirely in favor of a message-passing model. An OS that can exploit native message-passing would be the natural fit for such a design.

We believe that now is the time to reconsider how the OS should be restructured to not merely cope with the next generation of hardware, but efficiently exploit it. Furthermore, rather than evolving an inherently shared-memory model of OS structure to deal with complex tradeoffs and limited sharing, we take the opposite approach: design and reason about the OS as a distributed, non-shared system, and then employ sharing to optimize the model where appropriate.

Figure 4 depicts a spectrum of sharing and locking disciplines. Traditional operating systems, such as Windows and variants of Unix, have evolved from designs at the far left of the continuum towards finer-grained locking and more replication. These changes

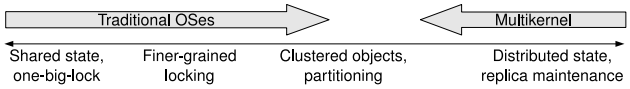


Figure 4: Spectrum of sharing and locking disciplines.

have been driven by hardware developments that exposed scalability bottlenecks, particularly the adoption of multiple processors in commodity PCs. Mainstream OSes are currently moving towards the center, where several "hot" data structures are partitioned or replicated across cores. Research systems take this even further with mechanisms like clustered objects that improve the locality of partitioned data [24]. In contrast, we propose an OS architecture positioned at the extreme right of the spectrum, where all state is replicated by default and consistency is maintained using agreement protocols.

3. THE MULTIKERNEL MODEL

In this section we present our OS architecture for heterogeneous multicore machines, which we call the *multikernel* model. In a nutshell, we structure the OS as a distributed system of cores that communicate using messages and share no memory (Figure 1). The multikernel model is guided by three design principles:

1. Make all inter-core communication explicit.
2. Make OS structure hardware-neutral.
3. View state as replicated instead of shared.

These principles allow the OS to benefit from the distributed systems approach to gain improved performance, natural support for hardware heterogeneity, greater modularity, and the ability to reuse algorithms developed for distributed systems.

After discussing the principles in detail below, in Section 4 we explore the implications of these principles by describing the implementation of Barrelfish, a new operating system based on the multikernel model.

3.1 Make inter-core communication explicit

Within a multikernel OS, all inter-core communication is performed using explicit messages. A corollary is that no memory is shared between the code running on each core, except for that used for messaging channels. As we have seen, using messages to access or update state rapidly becomes more efficient than shared-memory access as the number of cache-lines involved increases. We can expect such effects to become more pronounced in the future. Note that this does not preclude *applications* sharing memory between cores (see Section 4.8), only that the OS design itself does not rely on it.

Explicit communication patterns facilitate reasoning about the use of the system interconnect. In contrast to implicit communication (such as distributed shared memory, or the messages used for cache coherence), the knowledge of *what* parts of shared state are accessed *when* and by *who* is exposed. It is, of course, established practice in OS kernels to design point-solution data structures that can be updated using only one or two cache misses on particular architectures, but it is hard to evolve such operations as hardware changes, and such optimizations ignore the wider picture of larger state updates in the OS involving multiple structures.

We have previously argued that as the machine increasingly resembles a network, the OS will inevitably behave as a distributed

system [8]. Explicit communication allows the OS to deploy well-known networking optimizations to make more efficient use of the interconnect, such as pipelining (having a number of requests in flight at once), and batching (sending a number of requests in one message, or processing a number of messages together). In Section 5.2 we show the benefit of such techniques in the case of distributed capability management.

This approach also enables the OS to provide isolation and resource management on heterogeneous cores, or to schedule jobs effectively on arbitrary inter-core topologies by placing tasks with reference to communication patterns and network effects. Furthermore, the message abstraction is a basic requirement for spanning cores which are not cache-coherent, or do not even share memory.

Message passing allows operations that might require communication to be *split-phase*, by which we mean that the operation sends a request and immediately continues, with the expectation that a reply will arrive at some time in the future. When requests and responses are decoupled, the core issuing the request can do useful work, or sleep to save power, while waiting for the reply. A common, concrete example is remote cache invalidations. In a highly concurrent scenario, provided that completing the invalidation is not required for correctness, it can be more important not to waste time waiting for the operation to finish than to perform it with the smallest possible latency.

Finally, a system based on explicit communication is amenable to human or automated analysis. The structure of a message-passing system is naturally modular, because components communicate only through well-defined interfaces. Consequently it can be evolved and refined more easily [23] and made robust to faults [30]. Indeed, a substantial theoretical foundation exists for reasoning about the high-level structure and performance of a system with explicit communication between concurrent tasks, ranging from process calculi such as Hoare's communicating sequential processes and the π-calculus, to the use of queuing theory to analyze the performance of complex networks

3.2 Make OS structure hardware-neutral

A multikernel separates the OS structure as much as possible from the hardware. This means that there are just two aspects of the OS as a whole that are targeted at specific machine architectures – the messaging transport mechanisms, and the interface to hardware (CPUs and devices). This has several important potential benefits.

Firstly, adapting the OS to run on hardware with new performance characteristics will not require extensive, cross-cutting changes to the code base (as was the case with recent scalability enhancements to Linux and Windows). This will become increasingly important as deployed systems become more diverse.

In particular, experience has shown that the performance of an inter-process communication mechanism is crucially dependent on hardware-specific optimizations (we describe those used in Barrelfish in Section 4.6). Hardware-independence in a multikernel means that we can isolate the distributed communication algorithms from hardware implementation details.

We envision a number of different messaging implementations (for example, a user-level RPC protocol using shared memory, or a hardware-based channel to a programmable peripheral). As we saw in Section 2.5, hardware platforms exist today without cache coherence, and even without shared memory, and are likely to become more widespread. Once the message transport is optimized, we can implement efficient message-based algorithms independently of the hardware details or memory layout.

A final advantage is to enable late binding of both the protocol implementation and message transport. For example, different transports may be used to cores on IO links, or the implementation may be fitted to the observed workload by adjusting queue lengths or polling frequency. In Section 5.1 we show how a topology-aware multicast message protocol can outperform highly optimized TLB shootdown mechanisms on commodity operating systems.

3.3 View state as replicated

Operating systems maintain state, some of which, such as the Windows dispatcher database or Linux scheduler queues, must be accessible on multiple processors. Traditionally that state exists as shared data structures protected by locks, however, in a multikernel, explicit communication between cores that share no memory leads naturally to a model of global OS state replicated across cores.

Replication is a well-known technique for OS scalability [4, 24], but is generally employed as an optimization to an otherwise shared-memory kernel design. In contrast, any potentially shared state in a multikernel is accessed and updated *as if* it were a local replica. In general, the state is replicated as much as is useful, and consistency is maintained by exchanging messages. Depending on the consistency semantics required, updates can therefore be long-running operations to coordinate replicas, and so are exposed in the API as non-blocking and split-phase. We provide examples of different update semantics in Section 4.

Replicating data structures can improve system scalability by reducing load on the system interconnect, contention for memory, and overhead for synchronization. Bringing data nearer to the cores that process it will result in lowered access latencies.

Replication is required to support domains that do not share memory, whether future general-purpose designs or present-day programmable peripherals, and is inherent in the idea of specializing data structures for particular core designs. Making replication of state intrinsic to the multikernel design makes it easier to preserve OS structure and algorithms as underlying hardware evolves.

Furthermore, replication is a useful framework within which to support changes to the set of running cores in an OS, either when hotplugging processors, or when shutting down hardware subsystems to save power. We can apply standard results from the distributed systems literature to maintaining the consistency of OS state across such changes.

Finally, a potentially important optimization of the multikernel model (which we do not pursue in this paper) is to privately share a replica of system state between a group of closely-coupled cores or hardware threads, protected by a shared-memory synchronization technique like spinlocks. In this way we can introduce (limited) sharing behind the interface as an optimization of replication.

3.4 Applying the model

Like all models, the multikernel, while theoretically elegant, is an idealist position: no state is shared and the OS acts like a fully distributed system. This has several implications for a real OS.

We discussed previously (in Section 2.6) the software engineering concerns of message-passing systems in general. Within the operating system kernel, which typically consists of a smaller amount of code written by expert programmers, software development challenges are more easily managed. However, the drawback of a idealist message-passing abstraction here is that certain platform-specific performance optimizations may be sacrificed, such as making use of a shared L2 cache between cores.

The performance and availability benefits of replication are achieved at the cost of ensuring replica consistency. Some operations will inevitably experience longer latencies than others, and the extent of this penalty will depend on the workload, the data volumes and the consistency model being applied. Not only that, the model supports multiple implementations of the agreement protocols used to maintain consistency. This increases the burden on the developer who must understand the consistency requirements for the data, but on the other hand, can also precisely control the degree of consistency. For example, a global flush of the TLB on each CPU is order-insensitive and can be achieved by issuing a single multicast request, whereas other operations may require more elaborate agreement protocols.

From an OS research perspective, a legitimate question is to what extent a real implementation can adhere to the model, and the consequent effect on system performance and scalability. To address this, we have implemented Barrelfish, a substantial prototype operating system structured according to the multikernel model.

Specifically, the goals for Barrelfish are that it:

- gives comparable performance to existing commodity operating systems on current multicore hardware;
- demonstrates evidence of scalability to large numbers of cores, particularly under workloads that stress global OS data structures;
- can be re-targeted to different hardware, or make use of a different mechanism for sharing, without refactoring;
- can exploit the message-passing abstraction to achieve good performance by pipelining and batching messages;
- can exploit the modularity of the OS to place OS functionality according to the hardware topology or load.

In the next section we describe Barrelfish, and in Section 5 we explore the extent to which it achieves these goals.

4. IMPLEMENTATION

While Barrelfish is a point in the multikernel design space, it is not the only way to build a multikernel. In this section we describe our implementation, and note which choices in the design are derived from the model and which are motivated for other reasons, such as local performance, ease of engineering, policy freedom, etc. – we have liberally borrowed ideas from many other operating systems.

4.1 Test platforms

Barrelfish currently runs on x86-64-based multiprocessors (an ARM port is in progress). In the rest of this paper, reported performance figures refer to the following systems:

The *2×4-core Intel system* has an Intel s5000XVN motherboard with 2 quad-core 2.66GHz Xeon X5355 processors and a single external memory controller. Each processor package contains 2 dies, each with 2 cores and a shared 4MB L2 cache. Both processors are connected to the memory controller by a shared front-side bus, however the memory controller implements a snoop filter to reduce coherence traffic crossing the bus.

The *2×2-core AMD system* has a Tyan Thunder n6650W board with 2 dual-core 2.8GHz AMD Opteron 2220 processors, each with a local memory controller and connected by 2 HyperTransport links. Each core has its own 1MB L2 cache.

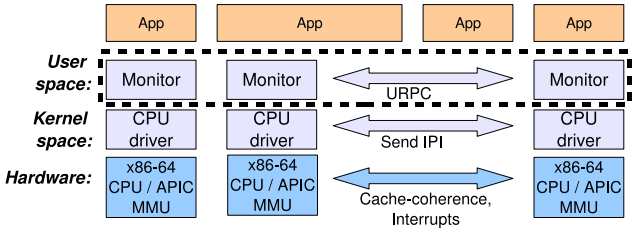


Figure 5: Barrelfish structure

The *4×4-core AMD system* has a Supermicro H8QM3-2 board with 4 quad-core 2.5GHz AMD Opteron 8380 processors connected in a square topology by four HyperTransport links. Each core has a private 512kB L2 cache, and each processor has a 6MB L3 cache shared by all 4 cores.

The *8×4-core AMD system* has a Tyan Thunder S4985 board with M4985 quad CPU daughtercard and 8 quad-core 2GHz AMD Opteron 8350 processors with the interconnect in Figure 2. Each core has a private 512kB L2 cache, and each processor has a 2MB L3 cache shared by all 4 cores.

4.2 System structure

The multikernel model calls for multiple independent OS instances communicating via explicit messages. In Barrelfish, we factor the OS instance on each core into a privileged-mode *CPU driver* and a distinguished user-mode *monitor* process, as in Figure 5 (we discuss this design choice below). CPU drivers are purely local to a core, and all inter-core coordination is performed by monitors. The distributed system of monitors and their associated CPU drivers encapsulate the functionality found in a typical monolithic microkernel: scheduling, communication, and low-level resource allocation.

The rest of Barrelfish consists of device drivers and system services (such as network stacks, memory allocators, etc.), which run in user-level processes as in a microkernel. Device interrupts are routed in hardware to the appropriate core, demultiplexed by that core's CPU driver, and delivered to the driver process as a message.

4.3 CPU drivers

The CPU driver enforces protection, performs authorization, time-slices processes, and mediates access to the core and its associated hardware (MMU, APIC, etc.). Since it shares no state with other cores, the CPU driver can be completely event-driven, single-threaded, and nonpreemptable. It serially processes events in the form of traps from user processes or interrupts from devices or other cores. This means in turn that it is easier to write and debug than a conventional kernel, and is small[2] enabling its text and data to be located in core-local memory.

As with an exokernel [22], a CPU driver abstracts very little but performs dispatch and fast local messaging between processes on the core. It also delivers hardware interrupts to user-space drivers, and locally time-slices user-space processes. The CPU driver is invoked via standard system call instructions with a cost comparable to Linux on the same hardware.

The current CPU driver in Barrelfish is heavily specialized for the x86-64 architecture. In the future, we expect CPU drivers for other processors to be similarly architecture-specific, including

[2]The x86-64 CPU driver, including debugging support and libraries, is 7135 lines of C and 337 lines of assembly (counted by David A. Wheeler's "SLOCCount"), 54kB of text and 370kB of static data (mainly page tables).

System	cycles (σ)	ns
2×4-core Intel	845 (32)	318
2×2-core AMD	757 (19)	270
4×4-core AMD	1463 (21)	585
8×4-core AMD	1549 (20)	774

Table 1: LRPC latency

data structure layout, whereas the monitor source code is almost entirely processor-agnostic.

The CPU driver implements a lightweight, asynchronous (split-phase) same-core interprocess communication facility, which delivers a fixed-size message to a process and if necessary unblocks it. More complex communication channels are built over this using shared memory. As an optimization for latency-sensitive operations, we also provide an alternative, synchronous operation akin to LRPC [9] or to L4 IPC [44].

Table 1 shows the one-way (user program to user program) performance of this primitive. On the 2×2-core AMD system, L4 performs a raw IPC in about 420 cycles. Since the Barrelfish figures also include a scheduler activation, user-level message dispatching code, and a pass through the thread scheduler, we consider our performance to be acceptable for the moment.

4.4 Monitors

Monitors collectively coordinate system-wide state, and encapsulate much of the mechanism and policy that would be found in the kernel of a traditional OS. The monitors are single-core, user-space processes and therefore schedulable. Hence they are well suited to the split-phase, message-oriented inter-core communication of the multikernel model, in particular handling queues of messages, and long-running remote operations.

On each core, replicated data structures, such as memory allocation tables and address space mappings, are kept globally consistent by means of an agreement protocol run by the monitors. Application requests that access global state are handled by the monitors, which mediate access to remote copies of state.

Monitors perform some further housekeeping functions in Barrelfish. As described in Section 4.6, monitors are responsible for interprocess communication setup, and for waking up blocked local processes in response to messages from other cores. A monitor can also idle the core itself (to save power) when no other processes on the core are runnable. Core sleep is performed either by waiting for an inter-processor interrupt (IPI) or, where supported, the use of MONITOR and MWAIT instructions.

4.5 Process structure

The multikernel model leads to a somewhat different process structure than a typical monolithic multiprocessor OS. A process in Barrelfish is represented by a collection of *dispatcher* objects, one on each core on which it might execute. Communication in Barrelfish is not actually between processes but between dispatchers (and hence cores).

Dispatchers on a core are scheduled by the local CPU driver, which invokes an upcall interface that is provided by each dispatcher. This is the mechanism used in Psyche [48] and scheduler activations [3], and contrasts with the Unix model of simply resuming execution. Above this upcall interface, a dispatcher typically runs a core-local user-level thread scheduler.

The threads package in the default Barrelfish user library provides an API similar to POSIX threads. We anticipate that language runtimes and parallel programming libraries will take advantage of the ability to customize its behavior, but in the meantime the library provides enough support for implementing the more traditional model of threads sharing a single process address space across multiple cores, as we describe in Section 4.8.

4.6 Inter-core communication

In a multikernel, all inter-core communication occurs with messages. We expect to use different transport implementations for different hardware scenarios. However, the only inter-core communication mechanism available on our current hardware platforms is cache-coherent memory. Barrelfish at present therefore uses a variant of user-level RPC (URPC) [10] between cores: a region of shared memory is used as a channel to transfer cache-line-sized messages point-to-point between single writer and reader cores.

Inter-core messaging performance is critical for a multikernel, and our implementation is carefully tailored to the cache-coherence protocol to minimize the number of interconnect messages used to send a message. For example, on the fast path for a HyperTransport-based system, the sender writes the message sequentially into the cache line, while the receiver polls on the last word of the line, thus ensuring that in the (unlikely) case that it polls the line during the sender's write, it does not see a partial message. In the common case, this causes two round trips across the interconnect: one when the sender starts writing to invalidate the line in the receiver's cache, and one for the receiver to fetch the line from the sender's cache. The technique also performs well between cores with a shared cache.

As an optimization, pipelined URPC message throughput can be improved at the expense of single-message latency through the use of cache prefetching instructions. This can be selected at channel setup time for workloads likely to benefit from it.

Receiving URPC messages is done by polling memory. Polling is cheap because the line is in the cache until invalidated; in addition, keeping the endpoints in an array can allow a hardware stride prefetcher to further improve performance. However, it is unreasonable to spin forever; instead, a dispatcher awaiting messages on URPC channels will poll those channels for a short period before blocking and sending a request to its local monitor to be notified when messages arrive. At present, dispatchers poll incoming channels for a predetermined time before blocking, however this can be improved by adaptive strategies similar to those used in deciding how long to spin on a shared-memory spinlock [37].

All message transports are abstracted behind a common interface, allowing messages to be marshaled, sent and received in a transport-independent way. As in most RPC systems, marshaling code is generated using a stub compiler to simplify the construction of higher-level services. A name service is used to locate other services in the system by mapping service names and properties to a service reference, which can be used to establish a channel to the service. Channel setup is performed by the monitors.

Table 2 shows the URPC single-message latency and sustained pipelined throughput (with a queue length of 16 messages); hop counts for AMD refer to the number of HyperTransport hops between sender and receiver cores.

Table 3 compares the overhead of our URPC implementation with L4's IPC on the 2×2-core AMD system[3]. We see that inter-core messages are cheaper than intra-core context switches in direct

[3] L4 IPC figures were measured using L4Ka::Pistachio of 2009-02-25.

System	Cache	Latency cycles (σ)	Latency ns	Throughput msgs/kcycle
2×4-core Intel	shared	180 (34)	68	11.97
	non-shared	570 (50)	214	3.78
2×2-core AMD	same die	450 (25)	161	3.42
	one-hop	532 (26)	190	3.19
4×4-core AMD	shared	448 (12)	179	3.57
	one-hop	545 (11)	218	3.53
	two-hop	558 (11)	223	3.51
8×4-core AMD	shared	538 (8)	269	2.77
	one-hop	613 (6)	307	2.79
	two-hop	618 (7)	309	2.75

Table 2: URPC performance

	Latency cycles	Throughput msgs/kcycle	Cache lines used Icache	Cache lines used Dcache
URPC	450	3.42	9	8
L4 IPC	424	2.36	25	13

Table 3: Messaging costs on 2×2-core AMD

cost, but also have less cache impact and do not incur a TLB flush. They can also be pipelined to trade off latency for throughput.

4.7 Memory management

Although a multikernel OS is itself distributed, it must consistently manage a set of global resources, such as physical memory. In particular, because user-level applications and system services may make use of shared memory across multiple cores, and because OS code and data is itself stored in the same memory, the allocation of physical memory within the machine must be consistent – for example, the system must ensure that one user process can never acquire a virtual mapping to a region of memory used to store a hardware page table or other OS object.

Any OS faces the same problem of tracking ownership and type of in-memory objects. Most systems use some kind of unique identifiers together with accounting information, such as file descriptors in Unix or object handles in Windows, managed by data structures in shared memory. For Barrelfish, we decided to use a capability system modeled on that of seL4 [39]. In this model, all memory management is performed explicitly through system calls that manipulate capabilities, which are user-level references to kernel objects or regions of physical memory. The model has the useful property of removing dynamic memory allocation from the CPU driver, which is only responsible for checking the correctness of operations that manipulate capabilities to memory regions through *retype* and *revoke* operations.

All virtual memory management, including allocation and manipulation of page tables, is performed entirely by user-level code [28]. For instance, to allocate and map in a region of memory, a user process must first acquire capabilities to sufficient RAM to store the required page tables. It then retypes these RAM capabilities to page table capabilities, allowing it to insert the new page tables into its root page table; although the CPU driver performs the actual page table and capability manipulations, its sole task is checking their correctness. The user process may then allocate more RAM capabilities, which it retypes to mappable frame capabilities, and finally inserts into its page tables to map the memory.

We chose capabilities so as to cleanly decentralize resource allocation in the interests of scalability. In hindsight, this was a mistake: the capability code is unnecessarily complex, and no more efficient than scalable per-processor memory managers used in conventional OSes like Windows and Linux. All cores must still keep their local capability lists consistent, to avoid situations such as user-level acquiring a mapping to a hardware page table.

However, one benefit has been uniformity: most operations requiring global coordination in Barrelfish can be cast as instances of capability copying or retyping, allowing the use of generic consistency mechanisms in the monitors. These operations are not specific to capabilities, and we would have to support them with any other accounting scheme.

Page mapping and remapping is an operation which requires global coordination – if an address space mapping is removed or its rights are reduced, it is important that no stale information remains in a core's TLB before any action occurs that requires the operation to have completed. This is implemented by a one-phase commit operation between all the monitors.

A more complex problem is capability retyping, of which revocation is a special case. This corresponds to changing the usage of an area of memory and requires global coordination, since retyping the same capability in different ways (e.g. a mappable frame and a page table) on different cores leads to an inconsistent system. All cores must agree on a single ordering of the operations to preserve safety, and in this case, the monitors initiate a two-phase commit protocol to ensure that all changes to memory usage are consistently ordered across the processors.

4.8 Shared address spaces

We are interested in investigating language runtimes that extend the multikernel model to user-space applications. However, for most current programs, Barrelfish supports the traditional process model of threads sharing a single virtual address space across multiple dispatchers (and hence cores) by coordinating runtime libraries on each dispatcher. This coordination affects three OS components: virtual address space, capabilities, and thread management, and is an example of how traditional OS functionality can be provided over a multikernel.

A shared virtual address space can be achieved by either sharing a hardware page table among all dispatchers in a process, or replicating hardware page tables with consistency achieved by message protocols. The tradeoff between these two is similar to that investigated in Corey [13]; the former is typically more efficient, however the latter may reduce cross-processor TLB invalidations (because it is possible to track which processors may have cached a mapping), and is also the only way to share an address space between cores that do not support the same page table format.

As well as sharing the address space, user applications also expect to share capabilities (for example, to mappable memory regions) across cores. However, a capability in a user's address space is merely a reference to a kernel-level data structure. The monitors provide a mechanism to send capabilities between cores, ensuring in the process that the capability is not pending revocation, and is of a type that may be transferred. The user-level libraries that perform capability manipulation invoke the monitor as required to maintain a consistent capability space between cores.

Cross-core thread management is also performed in user space. The thread schedulers on each dispatcher exchange messages to create and unblock threads, and to migrate threads between dispatchers (and hence cores). Barrelfish is responsible only for multiplexing the dispatchers on each core via the CPU driver scheduler, and coordinating the CPU drivers to perform, for example, gang scheduling or co-scheduling of dispatchers. This allows a variety

of spatio-temporal scheduling policies from the literature [62, 65] to be applied according to OS policy.

4.9 Knowledge and policy engine

Dealing with heterogeneous hardware and choosing appropriate system mechanisms is a crucial task in a multikernel. Barrelfish employs a service known as the *system knowledge base* (SKB) [60], which maintains knowledge of the underlying hardware in a subset of first-order logic.[4] It is populated with information gathered through hardware discovery (including ACPI tables, PCI bus probes, and CPUID data), online measurement (such as URPC communication latency and bandwidth between all core pairs in the system), and pre-asserted facts that cannot be discovered or measured (such as the interconnect topology of various system boards, and quirks that correct known flaws in discovered information, such as ACPI tables). Using this rich repository of data, the SKB allows concise expression of optimization queries, for example to allocate device drivers to cores in a topology-aware manner, to select appropriate message transports for inter-core communication, and to provide the topology information necessary for NUMA-aware memory allocation.

As one example, we describe in Section 5.1 how the SKB is used to construct a cache- and topology-aware network for efficient communication within multicore machines. Space precludes further discussion of policy decisions, but we believe that such a high-level, declarative approach to reasoning about the machine hardware (augmented with online measurements) is an essential part of a multikernel-based system.

4.10 Experiences

Barrelfish is one concrete implementation of the multikernel model in Section 3, and its structure differs substantially from that of a monolithic OS like Linux or indeed a hypervisor like Xen [7]. However, we emphasize that Barrelfish is not the only way to build a multikernel.

In particular, the factoring of the OS node into separate CPU driver and monitor components is not required by the model; our experience is that it is *not* optimal for performance, but has compelling engineering advantages. The downside of the separation is that invocations from processes to the OS are now mostly local RPC calls (and hence two context switches) rather than system calls, adding a constant overhead on current hardware of several thousand cycles. However, this is constant as the number of cores increases. Moving the monitor into kernel space would remove this penalty, at the cost of a more complex kernel-mode code base.

In addition, our current network stack (which runs a separate instance of lwIP [47] per application) is very much a placeholder. We are interested in appropriate network stack design for multikernels, and in particular we feel many ideas from RouteBricks [21] are applicable to a scalable end-system multikernel.

The principal difference between current OS designs and the multikernel model is not the location of protection or trust boundaries between system components, but rather the reliance on shared data as the default communication mechanism. Examples from monolithic systems include process, thread, and virtual machine control blocks, page tables, VFS objects, scheduler queues,[5] network sockets and inter-process communication buffers.

[4]The initial implementation of the SKB is based on a port of the ECL^{i}PS^e constraint programming system [5].

[5]Systems with per-processor run queues implement load balancing and thread migration either via a work stealing or work offloading model. In both cases this involves access to the run queues of remote processors.

Some of this state can be naturally partitioned, but (as we saw in Section 1) the engineering effort to do this piecemeal in the process of evolving an OS for scalability can be prohibitive. Furthermore, state which is inherently shared in current systems requires more effort to convert to a replication model. Finally, the shared-memory single-kernel model cannot deal with cores that are heterogeneous at the ISA level.

In contrast, starting with a multikernel model, as in Barrelfish, allows these issues to be addressed from the outset. From a research perspective, Barrelfish can be seen as a useful "proving ground" for experimenting with such ideas in an environment freed from the constraints of an existing complex monolithic system.

5. EVALUATION

In this section we evaluate how well Barrelfish meets the goals in Section 3.4: good baseline performance, scalability with cores, adaptability to different hardware, exploiting the message-passing abstraction for performance, and sufficient modularity to make use of hardware topology-awareness. We start with a detailed case-study of TLB shootdown, and then look at other workloads designed to exercise various parts of the OS.

5.1 Case study: TLB shootdown

TLB shootdown – the process of maintaining TLB consistency by invalidating entries when pages are unmapped – is one of the simplest operations in a multiprocessor OS that requires global coordination. It is also a short, but latency-critical, operation and so represents a worst-case comparison for a multikernel.

In Linux and Windows, inter-processor interrupts (IPIs) are used: a core that wishes to change a page mapping writes the operation to a well-known location and sends an interrupt to every core that might have the mapping in its TLB. Each core takes the trap, acknowledges the IPI by writing to a shared variable, invalidates the TLB entry, and resumes. The initiating core can continue immediately after every IPI has been acknowledged, since by the time each core resumes to user space the TLB entry is guaranteed to be flushed. This has low latency, but can be disruptive as each core immediately takes the cost of a trap (about 800 cycles). The TLB invalidation itself is fast, taking 95-320 cycles on a current x86-64 core.

In Barrelfish, we use messages for shootdown. In the naive algorithm, the local monitor broadcasts invalidate messages to the others and waits for all the replies. We would expect this to show higher latency than the IPI approach, since the remote monitors handle the messages only when "convenient". However, fortunately, the message-passing paradigm allows us to improve on this approach without significantly restructuring the code. In particular, we can exploit knowledge about the specific hardware platform, extracted from the system knowledge base at runtime, to achieve very good TLB shootdown performance.

Figure 6 shows the costs of the raw inter-core messaging mechanisms (without TLB invalidation) for four URPC-based TLB shootdown protocols on the 8×4-core AMD system.

In the *Broadcast* protocol, the master monitor uses a single URPC channel to broadcast a shootdown request to every other core. Each slave core polls the same shared cache, waiting for the master to modify it, and then acknowledges with individual URPCs to the master. This performs badly due to the cache-coherence protocol used by AMD64 processors [1, section 7.3]. When the master updates the line, it is invalidated in all other caches. Each slave core then pulls the new copy from the master's cache. With N

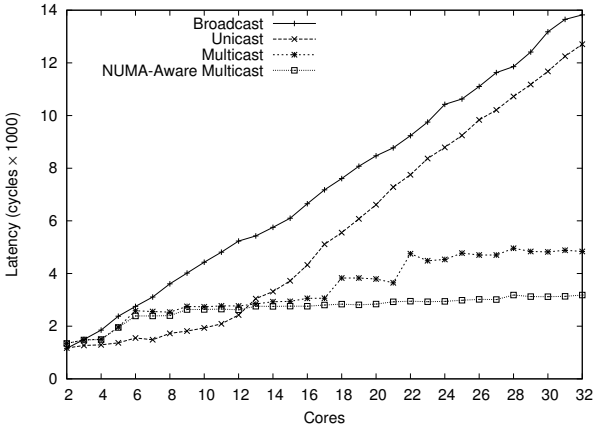


Figure 6: Comparison of TLB shootdown protocols

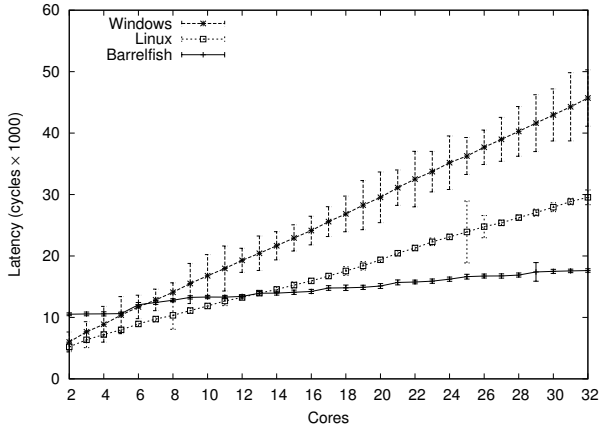


Figure 7: Unmap latency on 8×4-core AMD

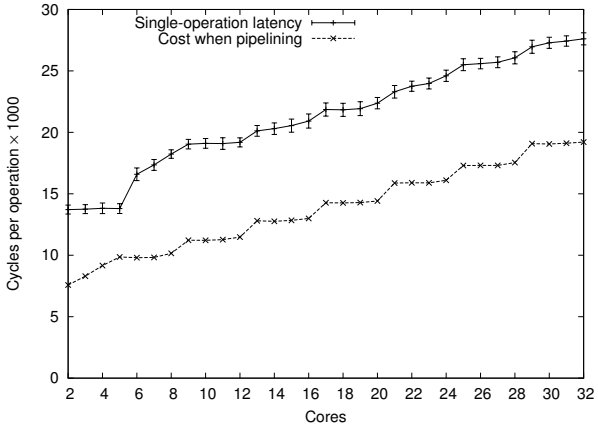


Figure 8: Two-phase commit on 8×4-core AMD

cores, the data crosses the interconnect N times, and latency grows linearly with the number of cores.

The *Unicast* protocol sends individual requests to each slave over unicast URPC so that the cache lines are only shared by two cores. While this performs much better than the broadcast protocol, particularly for a small number of cores, it still has linear scalability. The flatter curve below eight cores is likely to be the processor's hardware "stride prefetcher" predicting correctly which cache lines are likely to be accessed in the master's receive loop.

The HyperTransport interconnect is effectively a broadcast network, where each read or write operation results in probes being sent to all other nodes. However, newer 4-core Opteron processors have a shared on-chip L3 cache and appear as a single HyperTransport node, and so cache lines shared only by these cores will not result in interconnect traffic. This motivates using an explicit two-level multicast tree for shootdown messages. Hence in the *Multicast* protocol, the master sends a URPC message to the first core of each processor, which forwards it to the other three cores in the package. Since they all share an L3 cache, this second message is much cheaper, but more importantly all eight processors can send in parallel without interconnect contention. As shown in Figure 6, the multicast protocol scales significantly better than unicast or broadcast.

Finally, we devised a protocol that takes advantage of the NUMA nature of the machine by allocating URPC buffers from memory local to the multicast aggregation nodes, and having the master send requests to the highest latency nodes first. Once again, there are many analogies to networked systems which motivate these changes. The resulting protocol, labeled *NUMA-Aware Multicast* on Figure 6 scales extremely well across the 32-way system, showing steps only when the number of levels in the tree increases.

This communication protocol has good performance for the 8×4-core AMD system, but relies on hardware knowledge (including the interconnect topology and memory locality) that differs widely between systems, and only works when initiated from the first core. In Barrelfish, we use a query on the system knowledge base to construct a suitable multicast tree at runtime: for every source core in the system, the SKB computes an optimal route consisting of one multicast aggregation node per processor socket, ordered by decreasing message latency, and its children. These routes are calculated online at startup, and used to configure inter-monitor communication.

End-to-end unmap latency: The complete implementation of unmapping pages in Barrelfish adds a number of costs to the baseline protocol, and so in practice is much slower. These include the fixed overhead of LRPC to the local monitor, the per-message cost of marshaling and event demultiplexing, scheduling effects in the monitors, and variable overhead when a core's monitor is not currently its polling channels.

Nevertheless, as Figure 7 shows, the complete message-based unmap operation in Barrelfish quickly outperforms the equivalent IPI-based mechanisms in Linux 2.6.26 and Windows Server 2008 R2 Beta, Enterprise Edition. The error bars on the graphs are standard deviation. We show the latency to change the permissions of a page mapped by a varying number of cores, using `mprotect` on Linux and `VirtualProtect` on Windows. Despite the fact that up to 32 user-level processes are involved in each unmap operation, performance scales better than Linux or Windows, both of which incur the cost of serially sending IPIs. The large overhead on top of the baseline protocol figures is largely due to inefficiencies in the message dispatch loop of the user-level threads package, which has not been optimized.

5.2 Messaging performance

Two-phase commit

As we discussed in Section 4, Barrelfish uses a distributed two-phase commit operation for changing memory ownership and usage via capability retyping. This necessarily serializes more messages

	Barrelfish	Linux
Throughput (Mbit/s)	2154	1823
Dcache misses per packet	21	77
source → sink HT traffic* per packet	467	657
sink → source HT traffic* per packet	188	550
source → sink HT link utilization	8%	11%
sink → source HT link utilization	3%	9%

* HyperTransport traffic is measured in 32-bit dwords.

Table 4: IP loopback performance on 2×2-core AMD

than TLB shootdown, and is consequently more expensive. Nevertheless, as Figure 8 shows, using the same multicast technique as with shootdown we achieve good scaling and performance. If there are enough operations to perform, we can also pipeline to amortize the latency. The "cost when pipelining" line shows that a typical capability retype operation consumes fewer cycles than IPI-based TLB-shootdowns on Windows and Linux.

The cost of polling

It is reasonable to ask whether polling to receive URPC messages is a wasteful strategy under real workloads and job mixes. This depends on a number of factors such as scheduling policy and messaging load, but we present a simplistic model here: assume we poll for P cycles before sleeping and waiting for an IPI, which costs C cycles. If a message arrives at time t, the overhead (cost in extra cycles) of message reception is therefore:

$$overhead = \begin{cases} t & \text{if } t \leq P, \\ P + C & \text{otherwise.} \end{cases}$$

– and the latency of the message is:

$$latency = \begin{cases} 0 & \text{if } t \leq P, \\ C & \text{otherwise.} \end{cases}$$

In the absence of any information about the distribution of message arrival times, a reasonable choice for P is C, which gives upper bounds for message overhead at $2C$, and the latency at C.

Significantly, for Barrelfish on current hardware, C is around 6000 cycles, suggesting there is plenty of time for polling before resorting to interrupts. C in this case includes context switch overhead but not additional costs due to TLB fills, cache pollution, etc.

IP loopback

IP loopback (with no physical network card to impose a bottleneck) can be a useful stress-test of the messaging, buffering, and networking subsystems of the OS, since in many systems a loopback interface is often used for communication between networked services on the same host, such as a web application and database.

Linux and Windows use in-kernel network stacks with packet queues in shared data structures, thus loopback utilization requires kernel interaction and shared-memory synchronization. On Barrelfish, we achieve the equivalent functionality for point-to-point links by connecting two user-space IP stacks via URPC. By executing a packet generator on one core and a sink on a different core, we can compare the overhead induced by an in-kernel shared-memory IP stack compared to a URPC approach.

Our experiment runs on the 2×2-core AMD system and consists of a UDP packet generator on one core sending packets of fixed 1000-byte payload to a sink that receives, reads, and discards the packets on another core on a different socket. We measure application-level UDP throughput at the sink, and also use hardware performance counters to measure cache misses and utilization of the HyperTransport interconnect. We also compare with Linux, pinning both source and sink to cores using `libnuma`.

Table 4 shows that Barrelfish achieves higher throughput, fewer cache misses, and lower interconnect utilization, particularly in the reverse direction from sink to source. This occurs because sending packets as URPC messages avoids any shared-memory other than the URPC channel and packet payload; conversely, Linux causes more cache-coherence traffic for shared-memory synchronization. Barrelfish also benefits by avoiding kernel crossings.

5.3 Compute-bound workloads

In this section we use compute-bound workloads, in the form of the NAS OpenMP benchmark suite [36] and the SPLASH-2 parallel application test suite [63], to exercise shared memory, threads, and scheduling. These benchmarks perform no IO and few virtual memory operations but we would expect them to be scalable.

The Barrelfish user-level scheduler supports POSIX-like threads, and allows processes to share an address space across multiple cores as in a traditional OS. We compare applications running under Linux and Barrelfish on the same hardware, using GCC 4.3.3 as the compiler, with the GNU GOMP OpenMP runtime on Linux, and our own implementation over Barrelfish.

Figure 9 shows the results of five applications from the 4×4-core AMD machine[6]. We plot the compute time in cycles on Barrelfish and Linux, averaged over five runs; error bars show standard deviation. These benchmarks do not scale particularly well on either OS, but at least demonstrate that despite its distributed structure, Barrelfish can still support large, shared-address space parallel code with little performance penalty. The differences we observe are due to our user-space threads library vs. the Linux in-kernel implementation – for example, Linux implements barriers using a system call, whereas our library implementation exhibits different scaling properties under contention (in Figures 9a and 9c).

5.4 IO workloads

The experiments in this section are designed to exercise device drivers, interrupts and OS buffering using high-bandwidth IO, with the aim again of exposing any performance penalty due to the multikernel architecture.

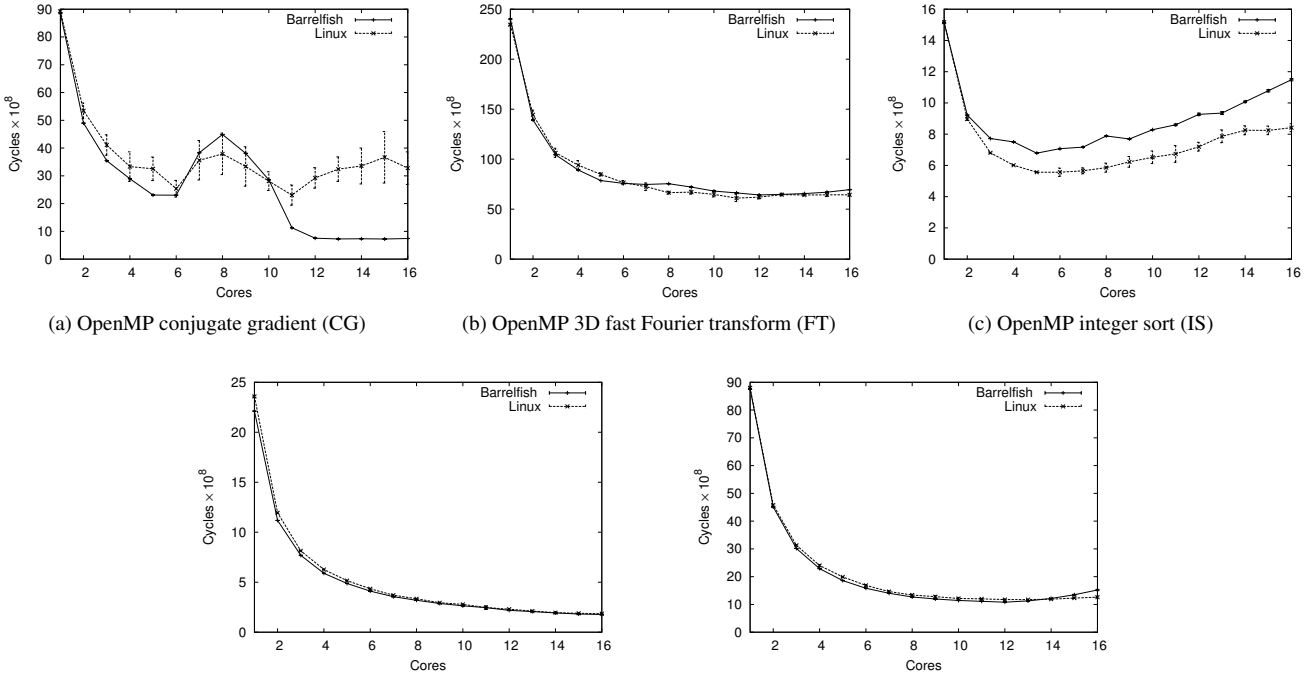
Network throughput

We first measure the UDP throughput of Barrelfish when communicating over Gigabit Ethernet using an Intel e1000 card. This exercises DMA, inter-process communication, networking and scheduling.

Using Barrelfish on the 2×4-core Intel machine, we ran an e1000 driver and a single-core application that listens on a UDP socket and echos every received packet back to the sender. The network stack is lwIP [47] linked as a library in the application's domain. Receive and transmit buffers are allocated by lwIP and passed to the network card driver, which manages the card's receive and transmit rings. The two processes communicate over a URPC channel, allowing us to vary the placement of the driver and client on cores.

We use two load generators, also with e1000 NICs, running Linux and the ipbench daemon [71], which generates UDP traffic

[6]The remaining OpenMP applications depend upon thread-local storage, which Barrelfish does not yet support.



(a) OpenMP conjugate gradient (CG)

(b) OpenMP 3D fast Fourier transform (FT)

(c) OpenMP integer sort (IS)

(d) SPLASH-2 Barnes-Hut

(e) SPLASH-2 radiosity

Figure 9: Compute-bound workloads on 4×4-core AMD (note different scales on y-axes)

at a configurable rate and measures the achieved echo throughput. We obtained UDP echo payload throughput of up to 951.7 Mbit/s, at which point we are close to saturating the card. By comparison, Linux 2.6.26 on the same hardware also achieves 951 Mbit/s; note that we pinned the Linux `inetd` server to a single core to prevent sub-optimal process migration in this experiment.

Web server and relational database

Our final IO example is of serving both static and dynamic web content from a relational database. This scenario, while still synthetic, is closer to a realistic I/O bound application configuration.

The 2×2-core AMD machine is used with an Intel e1000 NIC. First, the machine serves a 4.1kB static web page to a set of clients, and we measure the throughput of successful client requests using `httperf` [54] on a cluster of 17 Linux clients.

On Barrelfish, we run separate processes for the web server (which uses the lwIP stack), e1000 driver, and a timer driver (for TCP timeouts). These communicate over URPC, allowing us to experiment with placement of domains on cores. The best performance was achieved with the e1000 driver on core 2, the web server on core 3 (both cores on the same physical processor), and other system services (including the timer) on core 0.

For comparison, we also run lighttpd [45] 1.4.23 over Linux 2.6.26 on the same hardware; we tuned lighttpd by disabling all extension modules and logging, increasing the maximum number of connections and file descriptors to 1,500,000, using the Linux `epoll` event handler mechanism, and enabling hardware checksumming, scatter gather and TCP segmentation offload on the network interface.

The Barrelfish e1000 driver does not yet support the offload features, but is also substantially simpler. It sustained 18697 requests per second (640 Mbit/s), versus 8924 for lighttpd on Linux (316 Mbit/s). The performance gain is mainly due to avoiding kernel-user crossings by running entirely in user space and communicating over URPC.

Finally, we use the same load pattern to execute web-based SELECT queries modified from the TPC-W benchmark suite on a SQLite [64] database running on the remaining core of the machine, connected to the web server via URPC. In this configuration we can sustain 3417 requests per second (17.1 Mbit/s), and are bottlenecked at the SQLite server core.

5.5 Summary

It would be wrong to draw any quantitative conclusions from our large-scale benchmarks; the systems involved are very different. An enormous investment has been made in optimizing Linux and Windows for current hardware, and conversely our system is inevitably more lightweight (it is new, and less complete). Instead, they should be read as indication that Barrelfish performs *reasonably* on contemporary hardware, our first goal from Section 3.4.

We make stronger claims for the microbenchmarks. Barrelfish can scale well with core count for these operations, and can easily adapt to use more efficient communication patterns (for example, tailoring multicast to the cache architecture and hardware topology). Finally we can also demonstrate the benefits of pipelining and batching of request messages without requiring changes to the OS code performing the operations.

Since the Barrelfish user environment includes standard C and math libraries, virtual memory management, and subsets of the POSIX threads and file IO APIs, porting applications is mostly straightforward. In the course of this evaluation we ported a web server, network stack [47], and various drivers, applications and libraries to Barrelfish, which gives us confidence that our OS design offers a feasible alternative to existing monolithic systems.

Nevertheless, bringing up a new OS from scratch is a substantial undertaking, and limits the extent to which we can fully evaluate the multikernel architecture. In particular, this evaluation does not address complex application workloads, or higher-level operating system services such as a storage system. Moreover, we have not evaluated the system's scalability beyond currently-available commodity hardware, or its ability to integrate heterogeneous cores.

6. RELATED WORK

Although a new point in the OS design space, the multikernel model is related to much previous work on both operating systems and distributed systems.

In 1993 Chaves *et al.* [17] examined the tradeoffs between message passing and shared data structures for an early multiprocessor, finding a performance tradeoff biased towards message passing for many kernel operations.

Machines with heterogeneous cores that communicate using messages have long existed. The Auspex [11] and IBM System/360 hardware consisted of heterogeneous cores with partially shared memory, and unsurprisingly their OSes resembled distributed systems in some respects. We take inspiration from this; what is new is the scale of parallelism and the diversity of different machines on which a general-purpose OS must run. Similarly, explicit communication has been used on large-scale multiprocessors such as the Cray T3 or IBM Blue Gene, to enable scalability beyond the limits of cache-coherence.

The problem of scheduling computations on multiple cores that have the same ISA but different performance tradeoffs is being addressed by the Cypress project [62]; we see this work as largely complementary to our own. Also related is the fos system [69] which targets scalability through space-sharing of resources.

Most work on OS scalability for multiprocessors to date has focused on performance optimizations that reduce sharing. Tornado and K42 [4, 24] introduced clustered objects, which optimize shared data through the use of partitioning and replication. However, the base case, and the means by which replicas communicate, remains shared data. Similarly, Corey [13] advocates reducing sharing within the OS by allowing applications to specify sharing requirements for OS data, effectively relaxing the consistency of specific objects. As in K42, however, the base case for communication is shared memory. In a multikernel, we make no specific assumptions about the application interface, and construct the OS as a shared-nothing distributed system, which may locally share data (transparently to applications) as an optimization.

We see a multikernel as distinct from a microkernel, which also uses message-based communication between processes to achieve protection and isolation but remains a shared-memory, multithreaded system in the kernel. For instance, Barrelfish has some structural similarity to a microkernel, in that it consists of a distributed system of communicating user-space processes which provide services to applications. However, unlike multiprocessor microkernels, each core in the machine is managed completely independently – the CPU driver and monitor share no data structures with other cores except for message channels.

That said, some work in scaling microkernels is related: Uhlig's distributed TLB shootdown algorithm is similar to our two-phase commit [67]. The microkernel comparison is also informative: as we have shown, the cost of a URPC message is comparable to that of the best microkernel IPC mechanisms in the literature [44], without the cache and TLB context switch penalties.

Disco and Cellular Disco [14, 25] were based on the premise that large multiprocessors can be better programmed as distributed systems, an argument complementary to our own. We see this as further evidence that the shared-memory model is not a complete solution for large-scale multiprocessors, even at the OS level.

Prior work on "distributed operating systems" [66] aimed to build a uniform OS from a collection of independent computers linked by a network. There are obvious parallels with the multikernel approach, which seeks to build an OS from a collection of cores communicating over links within a machine, but also important differences: firstly, a multikernel may exploit reliable in-order message delivery to substantially simplify its communication. Secondly, the latencies of intra-machine links are lower (and less variable) than between machines. Finally, much prior work sought to handle partial failures (i.e. of individual machines) in a fault-tolerant manner, whereas in Barrelfish the complete system is a failure unit. That said, extending a multikernel beyond a single machine to handle partial failures is a possibility for the future that we discussion briefly below.

Despite much work on distributed shared virtual memory systems [2, 56], performance and scalability problems have limited their widespread use in favor of explicit message-passing models. There are parallels with our argument that the single-machine programming model should now also move to message passing. Our model can be more closely compared with that of distributed shared objects [6, 32], in which remote method invocations on objects are encoded as messages in the interests of communication efficiency.

7. EXPERIENCE AND FUTURE WORK

The multikernel model has been strongly influenced by the process of building a concrete implementation. Though not by any means yet a mature system, we have learned much from the process.

Perhaps unsurprisingly, queuing effects become very important in a purely message-oriented system, and we encountered a number of performance anomalies due to extreme sensitivity to message queue lengths. Understanding such effects has been essential to performance debugging. Scheduling based on queue lengths, as in Scout [53], may be a useful technique to apply here.

When implementing transports based on cache-coherent shared memory, we found it important to understand exactly the cache-coherence protocol. We experimented with many URPC implementations, with often unexpected results that were explained by a careful analysis of cache line states and interconnect messages.

Related future work includes efficient multicast, incast and anycast URPC transports. As we found in the case of unmap, there are cases where a multicast tree is more efficient for performing global operations than many point-to-point links, and we expect such transports will be important for scalability to many cores.

Our current implementation is based on homogeneous Intel and AMD multiprocessors, and so does not represent a truly heterogeneous environment. A port is in progress to the ARM processor architecture, which will allow us to run a Barrelfish CPU driver and monitor on programmable network cards. This will also allow us to experiment with specializing data structures and code for different processors within the same operating system.

There are many ideas for future work that we hope to explore. Structuring the OS as a distributed system more closely matches the structure of some increasingly popular programming models for datacenter applications, such as MapReduce [19] and Dryad [35], where applications are written for aggregates of machines. A distributed system inside the machine may help to reduce the "impedance mismatch" caused by the network interface – the same programming framework could then run as efficiently inside one machine as between many.

Another area of interest is file systems. Barrelfish currently uses a conventional VFS-style file-oriented API backed by NFS. It may be fruitful to draw on techniques from high-performance cluster file systems and the parallel IO models of cloud computing providers to construct a scalable, replicated file system *inside* the computer.

Barrelfish is at present a rather uncompromising implementation of a multikernel, in that it never shares data. As we noted in Section 2.1, some machines are highly optimized for fine-grained sharing among a subset of processing elements. A next step for Barrelfish is to exploit such opportunities by limited sharing behind the existing replica-oriented interfaces. This also raises the issue of how to decide *when* to share, and whether such a decision can be automated.

8. CONCLUDING REMARKS

Computer hardware is changing faster than system software, and in particular operating systems. Current OS structure is tuned for a coherent shared memory with a limited number of homogeneous processors, and is poorly suited to efficiently manage the diversity and scale of future hardware architectures.

Since multicore machines increasingly resemble complex networked systems, we have proposed the multikernel architecture as a way forward. We view the OS as, first and foremost, a distributed system which may be amenable to local optimizations, rather than centralized system which must somehow be scaled to the network-like environment of a modern or future machine. By basing the OS design on replicated data, message-based communication between cores, and split-phase operations, we can apply a wealth of experience and knowledge from distributed systems and networking to the challenges posed by hardware trends.

Barrelfish, an initial, relatively unoptimized implementation of the multikernel, already demonstrates many of the benefits, while delivering performance on today's hardware competitive with existing, mature, monolithic kernels. Its source code is available at http://www.barrelfish.org/.

Acknowledgments

We would like to thank our shepherd, Jeff Dean, the anonymous reviewers, and Tom Anderson, Steven Hand, and Michael Scott for their helpful suggestions for how to improve this paper and Barrelfish in general. We would also like to thank Jan Rellermeyer, Ankush Gupta, and Animesh Trivedi for their contributions to Barrelfish, and Charles Gray for the term "CPU driver".

References

[1] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Sept. 2007. Publication number 24593.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2), 1996.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazoswka, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of threads. *ACM Transactions on Computer Systems*, 10:53–79, 1992.

[4] J. Appavoo, D. Da Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems*, 25(3), 2007.

[5] K. R. Apt and M. G. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[6] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, 1998.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Oct. 2003.

[8] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, May 2009.

[9] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990.

[10] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, 1991.

[11] S. Blightman. Auspex Architecture – FMP Past & Present. Internal document, Auspex Systems Inc., September 1996. http://www.bitsavers.org/pdf/auspex/eng-doc/848_Auspex_Architecture_FMP_Sep96.pdf.

[12] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, pages 746–749, 2007.

[13] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 43–57, Dec. 2008.

[14] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[15] C. Caşcaval, J. G. Castaños, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H. S. Warren, Jr. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the 8th IEEE Symposium on High-Performance Computer Architecture*, pages 311–322, 2002.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 519–538, 2005.

[17] E. M. Chaves, Jr., P. C. Das, T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Kernel–Kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, 1993.

[18] P. Conway and B. Hughes. The AMD Opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.

[19] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, 2004.

[20] D. Dice and N. Shavit. TLRW: return of the read-write lock. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2009.

[21] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009.

[22] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Dec. 1995.

[23] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the EuroSys Conference*, pages 177–190, 2006.

[24] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, Feb. 1999.

[25] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 154–169, 1999.

[26] M. Gschwind. The Cell Broadband Engine: exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.

[27] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, 2008.

[28] S. M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 73–86, Feb. 1999.

[29] J. Held, J. Bautista, and S. Koehl. From a few cores to many: A tera-scale computing research overview. White paper, Intel, Sept. 2006. ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf.

[30] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *Operating Systems Review*, 40(3):80–89, July 2006.

[31] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, 2008.

[32] P. Homburg, M. van Steen, and A. Tanenbaum. Distributed shared objects as a communication paradigm. In *Proceedings of the 2nd Annual ASCI Conference*, pages 132–137, June 1996.

[33] Intel Corporation. QuickPath architecture white paper, 2008.

[34] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 186–197, 2007.

[35] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the EuroSys Conference*, pages 59–72, 2007.

[36] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Advanced Supercomputing Division, Moffett Field, CA, USA, Oct. 1999.

[37] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 41–55, 1991.

[38] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[39] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009.

[40] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[41] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proceedings of the 2007 Annual USENIX Technical Conference*, pages 1–14, 2007.

[42] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 64–75, 2004.

[43] H. C. Lauer and R. M. Needham. On the duality of operating systems structures. In *2nd International Symposium on Operating Systems, IRIA*, 1978. Reprinted in *Operating Systems Review*, 13(2), 1979.

[44] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Dec. 1995.

[45] lighttpd webserver. http://www.lighttpd.net/.

[46] Linux scalability effort. http://lse.sourceforge.net/.

[47] lwIP. http://savannah.nongnu.org/projects/lwip/.

[48] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 110–121, Oct. 1991.

[49] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 International Conference on Supercomputing*, pages 1–11, 2008.

[50] P. E. McKenney and J. Walpole. Introducing technology into the Linux kernel: A case study. *Operating Systems Review*, 42(5):4–17, July 2008.

[51] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, 1991.

[52] Microsoft. Receive-side Scaling enhancements in Windows Server 2008. http://www.microsoft.com/whdc/device/network/ndis_rss.mspx.

[53] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 153–167, Oct. 1996.

[54] D. Mosberger and J. Tin. httperf: A tool for measuring web server performance. *Performance Evaluation Review*, 26(3):31–37, Dec. 1998.

[55] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[56] J. Protić, M. Tomašević, and V. Milutinović. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology*, 4(2):63–79, 1996.

[57] M. Russinovich. Inside Windows Server 2008 kernel changes. *Microsoft TechNet Magazine*, Mar. 2008.

[58] M. Russinovich. Inside Windows 7. *Microsoft MSDN Channel9*, Jan. 2009.

[59] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 422–433, 2003.

[60] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, June 2008.

[61] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[62] D. Shelepov and A. Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.

[63] Stanford parallel applications for shared memory (SPLASH-2). http://www-flash.stanford.edu/apps/SPLASH/.

[64] SQLite database engine. http://www.sqlite.org/.

[65] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the EuroSys Conference*, pages 47–58, 2007.

[66] A. S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, 1985.

[67] V. Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, Computer Science Department, University of Karlsruhe, Germany, June 2005.

[68] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *International Solid-State Circuits Conference*, pages 98–589, Feb. 2007.

[69] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *Operating Systems Review*, 43(2), Apr. 2009.

[70] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown, III, and A. Agarwal. On-chip interconnection architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, 2007.

[71] I. Wienand and L. Macpherson. ipbench: A framework for distributed network benchmarking. In *AUUG Winter Conference*, Melbourne, Australia, Sept. 2004.

Speculative Execution in a Distributed File System

Edmund B. Nightingale, Peter M. Chen, and Jason Flinn
Department of Electrical Engineering and Computer Science
University of Michigan

ABSTRACT

Speculator provides Linux kernel support for speculative execution. It allows multiple processes to share speculative state by tracking causal dependencies propagated through interprocess communication. It guarantees correct execution by preventing speculative processes from externalizing output, e.g., sending a network message or writing to the screen, until the speculations on which that output depends have proven to be correct. Speculator improves the performance of distributed file systems by masking I/O latency and increasing I/O throughput. Rather than block during a remote operation, a file system predicts the operation's result, then uses Speculator to checkpoint the state of the calling process and speculatively continue its execution based on the predicted result. If the prediction is correct, the checkpoint is discarded; if it is incorrect, the calling process is restored to the checkpoint, and the operation is retried. We have modified the client, server, and network protocol of two distributed file systems to use Speculator. For PostMark and Andrew-style benchmarks, speculative execution results in a factor of 2 performance improvement for NFS over local-area networks and an order of magnitude improvement over wide-area networks. For the same benchmarks, Speculator enables the Blue File System to provide the consistency of single-copy file semantics and the safety of synchronous I/O, yet still outperform current distributed file systems with weaker consistency and safety.

General Terms

Performance, Design

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management—*Distributed file systems*; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'05, October 23–26, 2005, Brighton, United Kingdom.
Copyright 2005 ACM 1-59593-079-5/05/0010 ...$5.00.

Keywords

Distributed file systems, speculative execution, causality

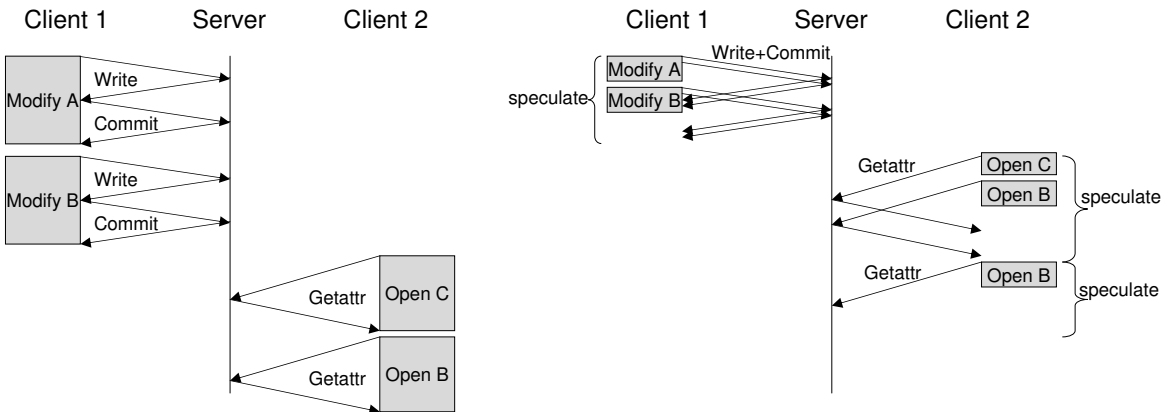
1. INTRODUCTION

Distributed file systems often perform substantially worse than local file systems because they perform synchronous I/O operations for cache coherence and data safety. File systems such as AFS [13] and NFS [3] present users with the abstraction of a single, coherent namespace shared across multiple clients. Although caching data on local clients improves performance, many file operations still use synchronous message exchanges between client and server to maintain cache consistency and protect against client or server failure. Even over a local-area network, the performance impact of this communication is substantial. As latency increases due to physical distance, middleboxes, and routing delays, the performance cost may become prohibitive.

Many distributed file systems weaken consistency and safety to improve performance. Whereas local file systems typically guarantee that a process that reads data from a file will see all modifications previously completed by other processes, distributed file systems such as AFS and NFS provide no such guarantee. For example, most NFS implementations provide *close-to-open* consistency, which guarantees only that a client that opens a file will see modifications made by other clients that have previously closed the file. Weaker consistency semantics improve performance by reducing the number of synchronous messages that are exchanged. Nevertheless, as our results show, even these weaker semantics are time-consuming.

We demonstrate that, with operating system support for lightweight checkpointing, speculative execution, and tracking of causal interdependencies between processes, distributed file systems can be fast, safe, and consistent. Rather than block a process while waiting for the result of a remote communication with a file server, the operating system checkpoints its state, predicts the result of the communication, and continues to execute the process speculatively. If the prediction is correct, the checkpoint is discarded; if it is false, the application is rolled back to the checkpoint.

Our solution relies on three observations. First, file system clients can correctly predict the result of many operations. For instance, consistency checks seldom fail since concurrent file updates are rare. Second, the time to take a lightweight checkpoint is often much less than network round-trip time to the server, so substantial work can be done while waiting for a remote request to complete. Finally, modern computers often have spare resources that can be used to execute



(a) Unmodified NFS

(b) Speculative NFS

Figure 1: Example of speculative execution for NFS

processes speculatively. Encouraged by these observations, and by the many prior successful applications of speculation in processor design, we have added support for speculative execution, which we call Speculator, to the Linux kernel.

In our work, the distributed file system controls when speculations start, succeed, and fail. Speculator provides a mechanism for correct execution of speculative code. It does not allow a process that is executing speculatively to externalize output, e.g., make network transmissions or display output to the screen, until the speculations on which that output depends prove to be correct. If a speculative process tries to execute a potentially unrecoverable operation, e.g., it calls the `reboot` system call, it is blocked until its speculations are resolved. Speculator tracks causal dependencies between kernel objects in order to share speculative state among multiple processes. For instance, if a speculative process sends a signal to its non-speculative parent, Speculator checkpoints the parent and marks it as speculative before it delivers the signal. If a speculation on which the child depends fails, both the child and parent are restored to their checkpoints (since the parent might not receive the signal on the correct execution path). Speculator tracks dependencies passed through fork, exit, signals, pipes, fifos, Unix sockets, and files in local and distributed file systems. All other forms of IPC currently block the speculative process until the speculations on which it depends are resolved.

Since speculation is implemented entirely in the operating system, no application modification is required. Speculative state is never externally visible. In other words, the semantics of the speculative version of a file system are identical to the semantics of the non-speculative version; however, the performance of the speculative version is better.

Results from PostMark and Andrew-style benchmarks show that Speculator improves the performance of NFS by more than a factor of 2 over local-area networks; over networks with 30 ms of round-trip latency, speculation makes NFS more than 14 times faster. We have also created a version of the Blue File System [24] that uses Speculator to provide *single-copy* semantics, in which the file consistency seen by two processes sharing a file and running on two different file clients is identical to the consistency that they would see if they were running on the same client. In addition, our version of BlueFS provides *synchronous I/O* in which all file modifications are safe on the server's disk before an operation is observed to complete. Despite providing these strong guarantees, BlueFS is 66% faster than non-speculative NFS over a LAN and more than 11 times faster with a 30 ms delay.

2. MOTIVATION: SPECULATION IN NFS

Figure 1 illustrates how Speculator improves distributed file system performance. Two NFS version 3 clients collaborate on a shared project that consists of three files: A, B, and C. At the start of the scenario, each client has up-to-date copies of all files cached. Client 1 modifies A and B; client 2 then opens C and B. Client 2 should see the modified version of B since that file was closed by client 1 before it was opened by client 2.

When an application closes a file, the Linux 2.4.21 NFSv3 client first sends asynchronous `write` remote procedure calls (RPCs) to the server to write back any data for that file that is dirty in its file cache—these RPCs are necessary to provide close-to-open consistency. After receiving replies for all `write` RPCs, the client sends a synchronous `commit` RPC to the server. The server replies only after it has committed all modifications for that file to disk. The NFS client returns from the `close` system call after receiving the `commit` reply. The `commit` RPC provides a safety guarantee, namely that no file modifications will be lost due to a server crash after the file has been closed. Thus, a Linux application that modifies a file in NFS incurs a performance penalty on close of at least two network round-trips and one synchronous disk access. Some other operating systems have NFS clients that do not wait for a `commit` reply before returning from `close`—these clients sacrifice safety, but improve performance since they block only until replies for all outstanding `write` RPCs have been received.

When an NFS client opens a file that it has previously cached, it issues a `getattr` RPC to the server. The file attributes returned by the server indicate whether the file has been modified since it was cached (in which case the cached copy is discarded and a new copy is fetched). Since the NFS server is a single point of synchronization, the `getattr` RPC

guarantees that the cached copy is fresh; if another client had modified and closed the file, the returned attributes would show the modification. For instance, in Figure 1(a), when client 2 reads file B, the attributes returned by `getattr` indicate that the file was modified. Hence, client 2 discards its cached copy of file B.

Cache coherence in NFS is time-consuming because a process blocks each time a file is closed after being modified or opened, as well as on directory lookups, permission checks, and modifications. This cost is magnified many times during activities such as listing a directory or compiling a program because each activity invokes several file system operations; for instance, most applications that show directory listings fetch the attributes of all files within the directory to display file types, sizes, or other metadata.

Our speculative version of NFS is shown in Figure 1(b). Client 1 asynchronously executes `write` and `commit` RPCs, speculating that all modifications will succeed at the server. Client 2 asynchronously executes the `getattr` RPCs, speculating that its cached copy of C and B are up-to-date. When the latter speculation fails, the calling process is rolled back to the start of the system call that opened B. This system call is re-executed and a new speculation begins.

Speculation improves file system performance because it hides latency: multiple file system operations can be performed concurrently, and computation can be overlapped with I/O. Speculation also improves write throughput. Because speculation transforms sequential operations issued by a single thread of control into concurrent operations, it allows the server to group commit such operations. Without OS support for speculative execution, the system call interface prevents these optimizations. The file system cannot return to the application until it receives the results of a remote operation, since that operation might fail.

3. CONDITIONS FOR SUCCESS

We believe that adding support for speculative execution to a commodity OS kernel will eventually benefit many applications. However, we have targeted distributed file systems as the first clients of Speculator because they exhibit three ideal characteristics:

1. **The results of speculative operations are highly predictable**. File system clients cache data in memory and on disk to improve performance. Due to ever-increasing memory and disk capacities, today's workstations can cache data for long periods of time. In the absence of concurrent modifications, cached data remains valid and can be used to successfully predict the outcome of remote operations. Speculation is a form of optimistic concurrency, since a client detects a conflicting update by another client only when it next accesses the modified file. Since concurrent updates are rare in distributed file systems [13], we expect the vast majority of speculations to succeed. However, our results show that even when many speculations fail, speculative file systems still substantially outperform non-speculative ones.

2. **Checkpointing is often faster than remote I/O**. Speculator checkpoints are essentially copy-on-write forks where the forked child is typically never executed. The time to take and discard a checkpoint of a small process is 52 μs; considerably less than the cost of a disk or network I/O. Although this time is greater for larger processes (6.3 ms for a 64 MB process), checkpoint cost can be amortized across several speculations by having those speculations share a single process checkpoint. Thus, there is considerable time available to execute applications speculatively when they would normally block on file system operations.

3. **Modern computers often have spare resources**. Speculative execution requires CPU cycles, and checkpoint storage requires memory. Fortunately, modern workstations typically have these resources in abundance. Since I/O is increasingly the performance bottleneck in modern computers [25], we can improve application performance by using these spare resources to hide I/O latency and improve I/O throughput.

```
create_speculation   (OUT spec_id, OUT dependencies);
commit_speculation   (IN spec_id);
fail_speculation     (IN spec_id);
```

Figure 2: Speculator interface

4. AN INTERFACE FOR SPECULATION

Our design for speculative execution exhibits a separation of concerns between policy and mechanism. The distributed file system determines when speculations begin, succeed, and fail. Speculator provides a lightweight checkpoint and rollback mechanism that allows speculative process execution. Speculator ensures that speculative state is never externalized or directly observed by non-speculative processes.

Speculator is implemented as part of the core Linux 2.4.21 kernel and consists of roughly 7,500 lines of C source code. Figure 2 shows Speculator's interface. A process must be executing in kernel mode (e.g., within a system call) to use this interface. To initiate speculative execution, a process calls `create_speculation`. This function returns a *spec_id* that uniquely identifies the particular speculation and a list of prior speculations on which the new speculation depends. Any process may later declare whether that speculation succeeds or fails by calling `commit_speculation` or `fail_speculation` with that spec_id. This design enables Speculator to remain ignorant of the particular hypothesis that underlies each speculation, as well as the semantics for success and failure. In turn, a Speculator client, e.g., a distributed file system, need not concern itself with the details of how speculative execution is performed.

The next section describes our basic implementation of speculative execution in the Linux kernel which allows processes to execute speculatively in isolation. Section 6 extends this implementation by allowing multiple processes to share speculative state. Section 7 describes how distributed file systems use Speculator.

5. IMPLEMENTING SPECULATION

5.1 Process checkpoint and rollback

Speculator implements checkpointing by performing a copy-on-write fork of the currently running process. It also saves the state of any open file descriptors and copies any

signals pending for the checkpointed process. In contrast to a normal fork, the child process is not placed on the run queue—it is simply a vessel for storing state. If all speculations on which a checkpoint depends prove to be correct, Speculator discards the checkpoint by reclaiming the kernel data structures associated with the child process.

If one of the speculations on which a checkpoint depends fails, Speculator restores the process to the state captured during the checkpoint. The process that is currently executing speculatively, called the *failed process*, is marked for termination by setting a flag in its task structure. When the failed process is next scheduled, Speculator forces it to exit. Speculator ensures that the failed process performs no externally visible operations prior to termination.

The process that was forked during the checkpoint, called the *checkpoint process*, assumes the identity of the failed process. Speculator gives the checkpoint process the process identifier, thread group identifier, and other distinguishing characteristics of the failed process. It also changes its file descriptors and pending signals to match the values saved during the checkpoint. The program counter of the checkpoint process is set to the system call entry point, and its kernel stack pointer is set to the initial kernel stack frame. Thus, after Speculator places the checkpoint process on the run queue, the process re-executes the system call that was being executed when the checkpoint was taken. Since the checkpoint process steals the identity of the failed process, this manipulation is hidden from observers outside the kernel; to the user, it appears that the speculative execution never happened.

5.2 Speculation

Speculator adds two new data structures to the kernel to track speculative state. A *speculation* structure is created during `create_speculation` to track the set of kernel objects that depend on the new speculation. An *undo log* is associated with each kernel object that has speculative state—this log is an ordered list of speculative operations that have modified the object. Each entry in the log contains sufficient information to undo the modifying operation, as well as references to all new speculative dependencies that were introduced by the operation. The presence of an entry in an object's undo log indicates that the object will be rolled back to the state associated with that entry if any of the referenced speculations fail. We say that a kernel object *depends* on all speculations for which references exist in its undo log. The speculations on which an object depends are *resolved* if all speculations commit (in which case, the object becomes non-speculative) or if any speculation fails (in which case, the object is rolled back to a prior state).

When a previously non-speculative process calls `create_speculation`, Speculator creates a new speculation structure and an undo log for the calling process. It checkpoints the calling process as described in the previous section and inserts an entry containing the checkpoint into its undo log. The new entry and the new speculation structure refer to each other. If the speculation fails, the checkpoint is used to restore the process to its prior state. If the process calls `create_speculation` again, Speculator creates a new speculation structure and appends a new entry to the process undo log. If the previous speculation was caused by a read-only operation such as a `stat` on a file in the distributed file system, no new checkpoint is needed. In this case, the new undo log entry shares a reference to the checkpoint contained in the previous log entry. If either speculation fails, the process is rolled back to the common checkpoint. Allowing speculations to share checkpoints improves performance in the common case where speculations succeed. Of course, in the uncommon case where the second speculation fails, the application must re-execute more work than if separate checkpoints had been taken.

Speculator caps the amount of work unnecessarily re-executed by taking a new checkpoint if the prior checkpoint for the process is more than 500 ms old. Speculator also caps the number of outstanding speculations at 2000 to prevent speculation from consuming too many system resources. As we gain more experience with the system, it may also prove useful to limit specific resources such as the amount of physical memory used for speculation.

Currently, two operations do not share a common checkpoint if the first operation modifies state. For example, if the first operation is a `mkdir` in the distributed file system, a common checkpoint cannot be used since the file server might make the effects of the `mkdir` visible to other clients, then fail the second operation. When the second operation fails, the client must roll back to the common checkpoint. Any clients that view the new directory would see incorrect state if the process that performed the `mkdir` does not recreate the directory when it re-executes. If necessary, Speculator could allow two mutating operations to share a checkpoint by modifying the file server to atomically perform all operations that share a checkpoint. However, this further optimization requires substantial server modifications, and our performance results indicate it is not needed.

5.3 Ensuring correct speculative execution

We define the speculative execution of a process to be *correct* if two invariants hold. First, speculative state should never be visible to the user or any external device. Enforcing this invariant requires that Speculator prevent a speculative process from externalizing output to the screen, network, or other interfaces. Second, a process should never view speculative state unless it is already speculatively dependent upon that state (because it could produce output that depends on that state). If a non-speculative process tries to view speculative state, Speculator either blocks the process until the state becomes non-speculative, or it makes the process speculative and rolls it back if a speculation on which that state depends fails.

Since interactions between a process and its external environment pass through the operating system, Speculator can prevent a speculative process from performing potentially incorrect operations by blocking that process until the speculations on which it depends are resolved. If all of those speculations prove successful, the process is unblocked and allowed to execute the operation (which is correct since the process is no longer speculative). If a speculation fails, the process is terminated.

We observe that blocking a speculative process is always correct, but that blocking limits the amount of work that can be done while waiting for remote operations to complete. Our approach to developing Speculator was to first create a correct but slow implementation that blocked whenever a speculative process performed a system call. We created a new system call jump table and modified the Linux system call entry point to use this table if the task structure of the

currently executing process is marked as speculative. Initially, we set all entries in this jump table to `sys_spec_deny`, a function we created to block the calling process until its speculations are resolved.

Next, we observed that system calls that do not modify state (e.g., `getpid`) are correct if performed by a speculative process. We let speculative processes perform these calls by replacing `sys_spec_deny` entries with the addresses of the functions that implement these syscalls. We also found that several system calls modify only state that is private to the calling process; these calls are correct to perform while speculative since their effects are not observed by other processes and, on speculation failure, their effects are undone as a side effect of restoring the checkpoint process. For example, `dup2` creates a new file descriptor that is a copy of an existing descriptor. This state is private to a process since file descriptors are not shared (except on fork, which is handled in Section 6.8). Further, when Speculator restores the checkpoint process, the effect of `dup2` is undone since the restored checkpoint contains the descriptor state of the failed process prior to calling `dup2`.

We next allowed speculative processes to perform operations on files in speculative file systems. For these system calls, it is insufficient to replace `sys_spec_deny` with syscalls such as `mkdir` in the speculative jump table because the OS may mount some file systems that support speculative execution and some that do not. For instance, one might use a non-speculative version of ext3 along with a speculative version of NFS—in this case, it is correct to allow speculative processes to modify state in NFS but not in ext3.

When a speculative process performs a file system operation, Speculator inspects the file system type to determine whether to block the calling process or allow speculative execution. On mount, a file system may set a flag in its superblock that indicates that speculative processes are allowed to read and write the files and directories it contains. Another flag allows just speculative read-only operations. All file system syscalls check these flags if the current process is speculative to decide whether to block or permit the operation. For example, `mkdir` blocks speculative processes unless the superblock of the parent directory indicates that mutating operations are allowed, and `stat` blocks unless read-only operations are allowed. Our speculative versions of NFS and BlueFS set the read/write flag. Because file systems that we have not modified do not set either flag, operations in those file systems block speculative processes until their speculations are resolved.

Speculator uses a similar strategy for the `read` and `write` system calls. When a file descriptor is opened, the type-specific VFS `open` function can set flags in the Linux file structure to indicate that speculative reads and/or writes are permitted. If a speculative process tries to read from or write to a file descriptor for which the appropriate flag is not set, that process is blocked until its speculations are resolved. The `read` flag is needed because read is a mutating operation for some inode types—for instance, reading from a Unix socket consumes the data that is read. For this inode type, speculative reads are incorrect. For other inode types, such as files in local file systems that do not update access times, reads are correct since they do not change object state.

If a speculative process writes to a tty or other external device, its action is incorrect since it is externalizing output. Yet, blocking such writes greatly limits the amount of work that can be done while speculative. This led us to support a third behavior for `write`: the data being written can be buffered in the kernel until the speculations on which it depends have been resolved. Speculator first validates such output operations to ensure that they can be performed. It then stores the output in a queue associated with the last checkpoint of the current process. After all speculations associated with that checkpoint commit, the buffered output is delivered to the device for which it was destined. If a speculation fails, the output is discarded. Currently, Speculator uses this strategy for output to the screen and network.

Output from a speculative process can appear before all of the speculations for that process are resolved. Consider a process that speculates on a remote operation, outputs a message, and then performs another speculative operation. Since the output depends only on the first speculation, Speculator can deliver it to the output device once the first speculation succeeds—Speculator need not wait for the second speculation to succeed or fail. In a non-speculative system, the output would be delayed while the process blocked on the remote I/O operation. Thus, the condition on which the output awaits, the completion of the remote I/O, is the same in non-speculative and speculative systems. In fact, output in the speculative system often appears *faster* than in the non-speculative system since the remote operations on which that output waits complete faster. An exception occurs if two speculations share a checkpoint since output that depends on only one speculation must wait for both to complete—this is another reason Speculator limits the maximum time difference between speculations that depend on the same checkpoint.

6. MULTI-PROCESS SPECULATION

We next allowed speculative processes to participate in inter-process communication. This significantly extends the amount of speculative work done by applications composed of multiple cooperating processes. For example, `make` forks children to perform compilation and linking; these processes communicate via pipes, signals, and files. If we limit speculation to a single process, `make` would often block waiting for a signal to be delivered or for data to arrive on a pipe.

6.1 Generic strategy

Speculator's strategy for IPC allows selected data structures within the kernel to have speculative state. Figure 3 illustrates how speculative state is propagated. In Figure 3(a), processes 8000 and 8001 both `stat` different files in BlueFS—BlueFS calls `create_speculation`, sends an asynchronous RPC to check cache consistency, and continues execution assuming that the cached attributes are up-to-date. Speculations 1 and 2 track the state associated with each speculation. Each process is marked as speculative, associated with an undo log, and checkpointed. Each undo log entry contains the process checkpoint and a reference to the speculation on which the process depends.

In Figure 3(b), process 8000 writes data to a file in /tmp. This causes Speculator to *propagate dependencies* from process 8000 to inode 3556. After this operation, the file contains speculative state; for example, if speculation 1 fails, process 8000 may write different data to the file on re-execution. Speculator therefore creates an undo log for inode 3556 and marks it as speculative. The entry in the file's

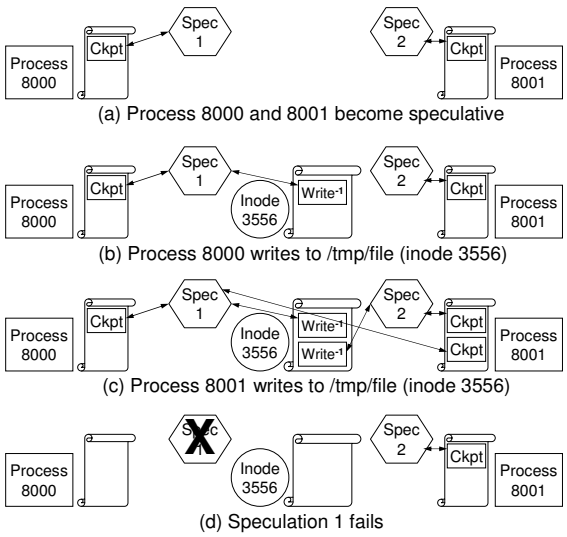


Figure 3: Propagating causal dependencies

undo log describes how to restore it to its previous state; in this case, the entry describes how to reverse the write operation that modified the file. The association between speculation 1 and this entry indicates that the write operation will be undone if speculation 1 fails.

In general, Speculator propagates dependencies from a process P to an object X whenever P modifies X and P depends on speculations that X does not. The entry in X's undo log for that operation is associated with all speculations on which P depended but X did not.

In Figure 3(c), process 8001 writes data to the same file. This operation creates another entry in the inode's undo log—this entry is associated with speculation 2. This operation also creates an entry in the process 8001 undo log since the process may have observed speculative state as a result of writing to the file. For example, if a speculative operation changes file permissions, the return value of a later write might depend on whether that speculation succeeds.

In general, Speculator propagates dependencies from an object X to a process P whenever P observes X's state and X depends on speculations that P does not. The entry in P's undo log for that operation is associated with all speculations on which X depended but P did not.

In our experience, almost all operations that modify kernel objects also observe the object being modified (in fact, the only exception we have seen is a signal sent by an exiting process). Thus, mutating operations normally propagate dependencies bi-directionally between the mutating process and the mutated object.

Process undo entries are not needed for many operations. For instance, an undo entry is not created for process 8000 when it modifies the file because the file depended on no speculations. Similarly, if process 8001 modifies the file again after step (c), an entry is not put in its undo log since the file and process 8001 depend on the same set of speculations.

During `commit_speculation`, Speculator deletes the speculation structure and removes its association with any undo log entries. Entries at the front of an undo log are deleted once they depend on no more speculations. When an undo log has no more entries, it is deleted, and its associated object becomes non-speculative.

Figure 3(d) shows what happens when a speculation fails. The failed speculation, speculation 1, is deleted. Each kernel object that depends on that speculation is rolled back to the state captured by the undo entry with which the failed speculation is associated. In this example, process 8000 is restored to its checkpoint and will retry the failed operation. Inode 3556 is restored to the state that existed before it was modified by process 8000—this is done by applying the two inverse operations in its undo log. Process 8001 is rolled back to its second checkpoint (because it could have observed incorrect speculative state in inode 3556). When process 8001 is restarted, it will attempt to write to inode 3556 again. This rollback is performed atomically during `fail_speculation`.

The undo log, undo entries, and speculations are generic kernel data structures. However, each undo log entry contains pointers to type-specific state and functions that implement type-specific rollback and roll forward processing. This design allows a common implementation for the type-independent logic associated with propagating dependencies, rolling state forward, and rolling state back. We next describe the type-specific logic for each form of IPC that Speculator currently supports.

6.2 Objects in a distributed file system

In our design, the file server always knows the correct state of each object (file, directory, etc.) If a speculation on which an object depends fails, Speculator simply invalidates the cached copy. When the object is next accessed, its correct state is retrieved from the server. Thus, undo log entries do not contain type-specific data. However, the undo log must still track each object's speculative dependencies so as to propagate those dependencies when one process reads a locally-cached object previously modified by another speculative process.

6.3 Objects in a local memory file system

We have modified the RAMFS memory-only file system to support speculative state. When a VFS operation modifies a speculative RAMFS object, Speculator inserts an entry in the inode's undo log that describes the inverse operation needed during rollback. For instance, a `rmdir` entry contains a reference to the directory being deleted as well as the name of the directory. The reference prevents the OS from reclaiming the directory object until the speculation is resolved. Thus, if a speculation on which the `rmdir` depends fails, Speculator can reinsert the directory into its original parent.

6.4 Objects in a local disk file system

We also modified the ext3 file system to support speculation. While Speculator uses the same strategy for managing ext3 inode undo logs that it uses for RAMFS, the presence of persistent state in ext3 presents many challenges.

One challenge is dealing with the possibility of system crashes. For instance, it would be incorrect to write speculative state to disk while maintaining undo information only in memory, since uncommitted speculative state would be visible after a crash. While Speculator could potentially write

undo logs to disk to deal with this possibility, we felt a simpler solution is to never write speculative data to disk (i.e., use a *no-steal* policy [11]). Speculator attaches undo logs to kernel buffer head objects. When a speculative process modifies a buffer, Speculator marks the buffer as speculative and inserts a record in its undo log. Kernel daemons that asynchronously write buffers to disk skip any buffer marked as speculative. A process that explicitly forces a speculative buffer to disk, e.g., by calling `fdatasync`, is blocked until the buffer's speculations are resolved. A substantial advantage of using a no-steal policy is that Speculator often does not need to shadow data overwritten by a speculative operation in memory—if the speculation fails, it invalidates the buffer, which will cause the prior state to be re-read from disk.

A second challenge is presented by shared on-disk metadata structures such as the superblock and allocation bitmaps. Ext3 operations often read and modify these structures, potentially creating too many speculative dependencies. For instance, when a speculative process allocates a new disk block for a file, it modifies the count of free blocks in the superblock. Under our previously described policy, any non-speculative process performing an ext3 operation would become speculative since it would observe speculative state in the superblock.

We felt that this policy was too aggressive as these metadata objects are seldom observed outside the kernel. Even in the case of inode numbers, which are externally visible, the impact of observing speculative state is limited (e.g., a new file might receive a different inode number than it would have received in the absence of speculation, yet the particular number received is typically unimportant as long as that number is unique). Speculator therefore allows processes to observe speculative metadata in ext3 superblocks, bitmaps, and group descriptors without propagating causal dependencies.

Speculator must allow these ext3 metadata structures to be occasionally written to disk. For instance, the ext3 superblock could remain speculative indefinitely if it is continuously modified by speculative processes. In this case, writing the superblock to disk is incorrect since it contains speculative data, yet if it is not written out periodically, substantial data could be lost after a crash. Speculator addresses this issue by maintaining shadow buffers for the superblock, bitmaps, and group descriptors. The actual buffer contains the current non-speculative state, while the shadow buffer contains the speculative state. When a speculative operation modifies one of these structures, it supplies redo and undo functions for the modification. The redo function is initially used to modify the speculative state. If the speculative operation commits, the redo function is applied to the non-speculative state. If the operation rolls back, the undo function is applied to the speculative state. Only the actual buffer containing non-speculative state is written to disk. This design relies on metadata operations being commutative (which is the case in Speculator).

A final challenge is presented by the ext3 journal, which groups multiple file system operations into compound transactions. A compound transaction may contain both speculative and non-speculative operations. To preserve the invariant that no speculative state is written to disk, we initially decided to block the commit of a compound transaction until all of its operations became non-speculative. However, this proved too restrictive since there are many instances in the ext3 code that wait for the current transaction to commit. We therefore adopted an approach that leverages Speculator's tracking of causal dependencies. Any file system operation that is causally dependent upon a speculative operation must itself be speculative. Thus, any non-speculative operation within a compound transaction cannot depend on a speculative operation within that transaction. Prior to committing a compound transaction, Speculator moves any speculative buffers in that transaction to the next compound transaction. For shared metadata buffers such as the superblock, Speculator commits the non-speculative version of the buffer as part of the current transaction. This approach lets the journal immediately commit non-speculative operations without writing speculative data to disk.

6.5 Pipes and fifos

Speculator handles pipes and fifos with a strategy similar to that used for local file systems. Since the `read` and `write` system calls both modify and observe pipe state, they propagate dependencies bi-directionally between the pipe and the calling process. The undo entry for `read` saves the data read so that it can be restored to the pipe if a speculation fails. The `write` entry saves the amount and location of data written so that it can be deleted on failure. Undo entries are also created when a pipe is created or deleted—the latter entry prevents the pipe from being destroyed until its speculations are resolved.

6.6 Unix sockets

Speculator tracks speculative dependencies propagated through Unix sockets by allowing kernel socket structures to contain speculative state. When data is written to a Unix socket, it is placed in a queue associated with the destination socket. Thus, any speculative dependencies are propagated bi-directionally between the sending process and the destination socket. Additionally, since the process sending data observes the state of the source socket, any speculative dependencies associated with that socket are propagated to both the sending process and destination socket. When a process reads data from the destination socket, dependencies are propagated bi-directionally between the process and the socket.

6.7 Signals

Signals proved difficult to handle. Our initial design had the sending process propagate dependencies directly to the process that receives the signal. However, this design requires Speculator to be able to checkpoint the receiving process at arbitrary points in its execution. The receiving process could receive a signal while it is in the middle of a system call. If the process has already performed some non-idempotent operation during the system call, restarting from the beginning of the system call is incorrect. If the checkpoint process is instead restarted from the location at which the signal was received, it would not have any kernel resources such as locks that were previously acquired during the system call by the process that received the signal.

Speculator solves this problem by requiring that all checkpoints be taken by the process being checkpointed. Checkpoints are only taken at well-known code locations where the checkpoint process will behave correctly if a rollback occurs.

Speculator delivers signals using a two-step process. First, when the signal is sent, it creates an undo log for that

signal—the log's only entry depends on the same speculations as the sending process. The signal is put in a *speculative sigqueue* associated with the receiving process. Signals in this queue are not yet considered delivered; thus, these signals are not visible to the receiving process when it checks its pending signals. During this step, Speculator propagates dependencies from the receiving process to the sending process, since the sender observes that it is *allowed* to send a signal to the destination process.

The second step occurs immediately before a process returns from kernel mode. At this time, Linux delivers any pending signals. We modified this code to first move any signals from the speculative sigqueue to the list of pending signals. If necessary, the receiving process is checkpointed and dependencies are propagated from the signals being delivered to the process. A flag is set in the checkpoint so that the checkpoint process will exit the kernel instead of restarting the system call if a rollback occurs. This preserves the invariant that checkpoints are only taken by the checkpointed process.

When a signal is moved from the speculative sigqueue to the list of pending signals, a new entry is inserted into the signal undo log. This ensures that if the receiving process rolls back to a point before the signal was delivered, that signal will be re-delivered. On the other hand, if a speculation on which the signal depends fails, the signal is destroyed and the receiving process rolls back to the checkpoint taken before delivery.

If a signal becomes non-speculative while it waits in the speculative sigqueue (because all speculations on which it depends commit), it is immediately delivered to the receiving process. This ensures that a process will eventually proceed if it blocks in the kernel waiting for a speculative signal to be delivered. However, to maximize performance, such signals should be delivered immediately. Speculator therefore interrupts `wait` and `select` when a signal arrives in the speculative sigqueue (just as they are interrupted for a pending signal). In these instances, we have ensured that these system calls can be restarted correctly if a speculation on which the delivered signal depends fails.

6.8 Fork

When a process forks a child, that child inherits all dependencies of its parent. If one of these speculations fails, the forked process is simply destroyed since it may never exist on the correct execution path.

6.9 Exit

When a speculative process exits, its `pid` is not deallocated until all of its dependencies are resolved. This ensures that checkpoints can be restarted with that `pid` if a speculation fails. The rest of the Linux exit code is executed, including the sending of signals to the parent of the exiting process. Speculating through exit is important for supporting applications such as `make` that fork children to perform sub-tasks. Without this support, the parent process would block until its child's speculations are resolved.

6.10 Other forms of IPC

Speculator does not currently let speculative processes communicate via System V IPC, futexes, or shared memory. It does allow speculative execution of processes with read-only access to shared memory segments or write access to private segments. However, it blocks processes that have write access to one or more shared memory segments—this precludes multi-threaded applications from speculating. Rather than track individual memory accesses, our planned approach to support multi-threaded processes is to consistently checkpoint all threads within a single thread group whenever a checkpoint is needed for any thread.

7. USING SPECULATION

Modifying a distributed file system to use Speculator typically involves changes to the client, server, and network protocol. We first inspect the client code and identify instances in which the client can accurately predict the outcome of remote operations. Predicting the outcome of read-only operations typically requires that the client memory or disk cache contain some information about the state of the file system. For instance, an NFS client can predict the result of a `getattr` RPC only when the inode attributes are cached. In this case, the client is speculating that the file has not been modified after it was cached.

We change the synchronous RPCs in these instances to be asynchronous, and have the client create a new speculation before sending each request. When a reply to the asynchronous RPC arrives, the client calls `commit_speculation` if it correctly predicted the result of the RPC and `fail_speculation` if it did not. Normally, if a prediction fails, the file system invalidates any cached values on which that prediction depended—this forces a non-speculative, synchronous RPC when the operation is re-executed. However, if the reply contains updated state, the file system instead caches the new state—in this case, a new speculation based on the updated values is made when the operation is re-executed.

In the next two subsections, we address two generic issues that occur when modifying a file system to use Speculator: correctly supporting mutating operations and implementing group commit. Sections 7.3 and 7.4 describe our speculative implementations of NFS and BlueFS in more detail. Section 7.5 discusses how other file system might benefit from using Speculator.

7.1 Handling mutating operations

Operations that mutate file system state at the server are challenging since a speculative mutation could potentially be viewed by a client other than the one that made the mutation. For example, a process might read the contents of a cached directory and write them to a file (e.g., `ls /dfs/foo > /dfs/bar`). The `readdir` operation creates a speculation that the cached version of directory `foo` is up-to-date. After the process writes to and closes `bar`, the contents of this file depend on that speculation. If a process running on another file system client reads `/dfs/bar`, it views speculative state. Thus, if the original speculation made during the `readdir` fails, both clients and the server must roll back state.

Potentially, Speculator could allow multiple clients to share speculative state using a mechanism such as Time Warp's virtual time [14]. Alternatively, it could allow each client to share its speculative state with the server, but not with other clients, by having the server block any clients that attempt to read or write data that has been speculatively modified by another client. However, we realized that the restricted nature of communication in a server-based distributed file system enables a much simpler solution.

The file server always knows the true state of the file system; thus, when it receives a speculative mutating operation, it can immediately evaluate whether the hypothesis that underlies that speculation is true or false. Based on this observation, we modify the server to only perform a mutation if its hypothesis is valid. If the hypothesis is false, the server fails the mutation. Thus, the server effectively controls whether a speculation succeeds or fails. A client that receives a failure response cannot commit that speculation.

In each speculative RPC, the client includes the hypothesis underlying that speculation. In the above example, a BlueFS client would send a `check_version` RPC containing the version number of its cached copy of `foo` when it executes a speculative `readdir`. The server checks this version number against the current file version and fails the speculation if the two differ. Part of the hypothesis of any mutating operation is that all prior speculations on which it depends have succeeded. In the example, the modification to `bar` depends on the success of the `readdir` speculation. These causal dependencies are precisely the set of speculations that are associated with the undo log of the process prior to transmitting the mutating operation. This list is returned by `create_speculation` and included in any speculative RPC sent to the server. If the server has previously failed any of the listed speculations, it fails the mutation.

This design requires that the server keep a per-client list of failed speculations. It also requires that the server process messages in the same order that a client generates them (otherwise the server could erroneously allow a mutation because it has not yet processed an invalid speculation on which that mutation depends). We enforce this ordering using client-generated sequence numbers for all RPCs. This sequence number also limits the amount of state that the server must keep. Each client appends the sequence number of the last reply that it received from the server to each new RPC. Once the server learns that the reply associated with a failed speculation has been processed, it discards that state (since the client will not allow any process that depends on that speculation to execute further).

A substantial advantage of this approach is simplicity: the server never stores speculative file data. Because all file modifications must pass through the server, other clients never view speculative state in the file system. Thus, when a speculation fails, only a single client must roll back state. To enable this simple design we had to modify each file system's underlying RPC protocol to support dependencies and asynchronous messages. While we have chosen to explicitly modify each file server, all server modifications (with the exception of group commit described below) could be implemented with a server-side proxy.

Speculator reduces the size of the list sent to the server by representing speculations as <pid,spec_id> tuples and generating the low-order spec_id with a per-process counter that is incremented on each new speculation. The dependency list is encoded as a list of extents of consecutive identifiers. Typically, this produces a list size proportional to the number of processes sharing speculative state, rather than the number of speculations (it is not quite proportional because rollbacks can create gaps in the spec_id sequence).

Speculator assumes that a client communicates with only one file server at a time. In the future, we plan to block any process that attempts to write to a server while it has uncommitted speculations that depend on another server.

7.2 Speculative group commit

Our original intent in using speculative execution was to hide the latency associated with remote communication. However, we soon realized that speculative execution can also improve throughput. Without speculation, synchronous operations performed by a single thread of control must be done sequentially. Speculative execution allows such operations to be performed concurrently. This creates opportunities to amortize expensive activities across multiple operations. A prime example of this is group commit.

In most distributed file systems (e.g., Coda and NFS), the file server commits synchronous mutating operations to disk before replying to clients. Commits limit throughput because each requires a synchronous disk write. A well-known optimization in the presence of concurrent operations is to group commit multiple operations with a single disk write. However, opportunities for group commit in a file server are limited because each thread of control has at most one in-flight synchronous mutating operation (although the Linux NFS server groups the commit of asynchronous writes to the same file). In contrast, speculative execution lets a single thread of control have many synchronous mutating operations in-flight—these can be committed with a single disk command.

We modify the file server to delay committing and replying to operations until either (a) no operations are in its network buffer, or (b) it has processed 100 operations. The server then group commits all operations and replies to clients.

7.3 Network File System

In order to explore how much speculative execution can improve the performance of existing distributed file systems, we modified NFS version 3 to use Speculator. Our modified version, called SpecNFS, preserves existing NFS semantics, including close-to-open consistency. It issues the same RPCs that the non-speculative version issues; however, in SpecNFS, many of these RPCs are speculative and contain the additional hypothesis data described in Section 7.1.

The NFS version 3 specification requires data to be committed to stable storage before returning from a synchronous mutating RPC such as `mkdir` (the Linux 2.4.21 implementation meets this specification unless the `async` mount flag is used). Implicit in this specification is the assumption that the operation should not be observed to complete until its modifications are safe on stable storage. In SpecNFS, although a process may continue to execute speculatively while it waits for a reply, it cannot externalize any output that depends on the speculation. Thus, any operation that is observed to complete has already been committed to stable storage at the server.

The SpecNFS client speculatively executes `getattr` RPCs when it has a locally cached copy of the inode being queried. If the inode is not cached, a blocking, non-speculative `getattr` is issued. The client also speculatively executes `lookup` and `access` RPCs if cached data is available. Mutating RPCs such as `setattr`, `create`, `commit`, and `unlink` always speculate.

One complication with the NFS protocol is that the server generates new file handles and fileids on RPCs such as `create` and `mkdir`. These values are chosen non-deterministically, making it difficult for the client to predict which values the server will choose. The SpecNFS client chooses aliases for these values that it can use until it re-

ceives the reply to its original RPC. After the client receives the reply, it discards the aliases and uses the real values. When the server receives a message that uses an alias, it replaces the alias with the actual value before performing the request. It discards an alias once it learns that the client has received the reply that contains the actual value.

The Linux NFS server does not use a write-ahead log; instead, it updates files in place. The server syncs data to disk whenever it processes `commit` RPCs and RPCs that modify directory data. Rather than sync data during the processing of these individual RPCs, the SpecNFS server syncs its file cache as a whole when no more incoming operations are in its network buffer, or when its commit limit of 100 operations has been reached. After syncing the file cache, the server replies to the RPCs sent by its clients—these clients then commit the speculations associated with those RPCs. This implementation of group commit is less efficient than one that uses a write-ahead log; however, it still improves disk throughput because the kernel orders disk writes during syncs and coalesces writes to the same disk block.

For example, consider a client that quickly modifies and closes a small file twice. The Linux NFS client generates a `write` RPC, followed by a `commit` RPC for the first close. It then generates additional `write` and `commit` RPCs for the second close. Given the Linux client implementation that waits for replies to all `write` RPCs before sending the `commit`, this activity results in four network round-trip delays. Furthermore, the client must wait twice for file data to be committed to disk—once for each `commit` RPC. In comparison, SpecNFS creates four speculations and sends all four RPCs asynchronously. The SpecNFS server delays committing data to disk and replying to the client until it processes all four messages. Committing the file data requires only a single disk write if the two writes are to the same file location.

7.4 Blue File System

We next explored whether speculative execution enables a distributed file system to provide strong consistency and safety, yet still have reasonable performance. Since we are currently developing a new distributed file system, BlueFS [24], we had the opportunity to develop a clean-sheet design that used Speculator. We exploited this opportunity to provide the consistency of single-copy file semantics and the safety of synchronous I/O.

Single-copy file semantics are equivalent to the consistency of a shared local disk, i.e., any read operation sees the effects of all preceding modifications. In contrast, the weaker close-to-open consistency of NFS guarantees only that a process that opens a file on one client will see the modifications of any client that previously closed that file. NFS makes only loose, timeout-based guarantees about data written before a file is closed, file attributes, and directory operations. This creates a window during which clients may view stale data or create inconsistent versions, leading to erroneous results.

In BlueFS, each file system object has a version number that is incremented when it is modified. All file system operations that read data from the client cache speculate and issue an asynchronous RPC that verifies that the version of the cached object matches the version at the server. All operations that modify an object verify that the cached version prior to modification matches the server's version prior to modification. In the event of a write/write or read/write conflict, the version numbers do not match—in this case, the speculation fails and the calling process is restored to a prior checkpoint.

Clearly, an implementation that performs these version checks synchronously provides single-copy semantics since all file system operations are synchronized at the server. Our speculative version (which performs version checks asynchronously) also provides single-copy semantics since no operation is observed to complete until the server's reply is received. Until some output is observed, the status of the operation is unknown (similar to the fate of Schrödinger's cat). In order for a write operation on one client to precede a read (or write) operation on another, the read must begin after the write has been observed to complete. Since that observation can only be made after the write reply has been received from the server, the incrementing of the version number at the server also precedes the beginning of the read operation. Thus, if the read operation uses stale data in its cache that does not include the modification made during the write operation, the version check fails. In this event, a rollback occurs and the read is re-executed with the modified version.

Our second goal was to provide the safety of synchronous I/O. Before the server replies to a client RPC, it commits any modifications associated with that RPC to a write-ahead log. Since the client does not externalize any output that depends on that RPC until it receives the reply, any operation that is observed to complete is already safe on the server's disk. In NFS, this safety guarantee is provided only when a file is closed; our modified version of BlueFS provides safety for *all* file system operations, including each `write`. Since commits occur frequently, BlueFS uses group commit to improve throughput. After a group commit, the server reduces network traffic by summarizing the outcome of all operations committed on behalf of a client in a single reply message

7.5 Discussion: Other file systems

While we have modified only NFS and BlueFS to use speculation, it is useful to consider how Speculator could benefit other distributed file systems. Since speculation improves performance by eliminating synchronous communication, the performance improvement seen by a particular file system will depend on how often it performs synchronous operations.

Both NFS and BlueFS implement cache coherence by polling the file server to verify that cached files are up-to-date. Since polling requires frequent synchronous RPCs to the server, these file systems see substantial benefit from Speculator. Other file systems use cache coherence schemes that reduce the number of synchronous RPCs performed in common usage scenarios. AFS [13] and Coda [18] grant clients *callbacks*, which are promises by the server to notify the client before a file system object is modified. A client holding a callback on an object does not need to poll the server before executing a read-only operation on that object. File delegations in NFSv4 [27] provide similar functionality. SFS [22] also extends the NFS protocol with callbacks and leases on file attributes. Echo [2] uses leases to provide single-copy consistency; a client granted an exclusive lease on an object can read or modify that object without contacting the server.

While the use of callbacks or leases reduces the number of

synchronous RPCs, it can increase the latency of individual RPCs. Before a server can accept a file modification or grant a client exclusive access to a file, it must first synchronously revoke any callbacks or leases held by other clients. Potentially, a well-connected client must wait on one or more poorly-connected clients. Speculator would help hide the latency of these expensive operations. Thus, we expect that file systems that use leases or callbacks would see substantial benefit from Speculator, even though the relative benefit would be less than that seen by file systems that use polling.

Speculator also reduces the cost of providing safety guarantees. AFS, Coda (in its strongly-connected mode), and NFS write file modifications synchronously to the server on close. Directory caching in these file systems is write-through. Speculator would substantially improve write performance in these file systems by hiding the latency of these synchronous operations. Since file modifications may not be written back to the server until the file is closed, data can be lost in the event of a system crash. Echo caches modifications for longer periods of time and writes back modifications asynchronously (unless a lease is revoked). This improves performance by reducing the number of synchronous RPCs but increases the size of the window during which data can be lost due to system crashes.

To date, file system designers have had to choose whether to provide strong consistency guarantees, strong safety guarantees, or good performance. Speculative execution changes this equation by eliminating synchronous communication. As our BlueFS results in the next section demonstrate: a distributed file system using Speculator can provide the safety of synchronous I/O, as well as single-copy semantics, and still perform better than current file systems.

8. EVALUATION

Our evaluation answers two questions:

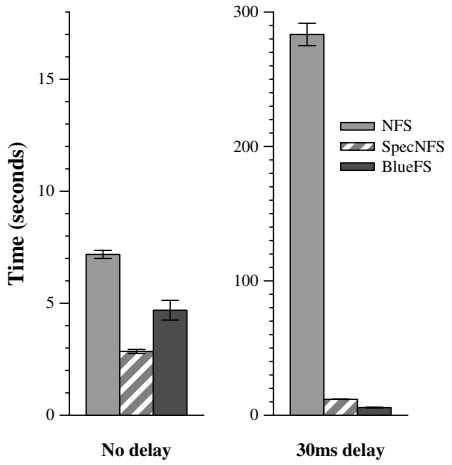
- How much does Speculator improve the performance of an existing file system (NFS)?
- With Speculator, what is the performance impact of providing single-copy file semantics and synchronous I/O (in BlueFS)?

8.1 Methodology

We use two Dell Precision 370 desktops as the client and file server. Each machine has a 3 GHz Pentium 4 processor, 2 GB DRAM, and a 160 GB disk. We run RedHat Enterprise Linux release 3 with kernel version 2.4.21. To insert delays, we route packets through a Dell Optiplex GX270 desktop running the NISTnet [4] network emulator. All computers communicate via 100 Mb/s Ethernet switches—the measured ping time between client and server is 229 μs.

SpecNFS mounts with the -o tcp option to use TCP as the transport protocol. For comparison, we run the non-speculative version of NFS with both UDP and TCP. Although results were roughly equivalent, we always report the best of the two results for non-speculative NFS. While BlueFS can cache data on local disk and portable storage, it uses only the Linux file cache in these experiments—this provides a fair comparison with NFS, which uses only the file cache. The client /tmp directory is a RAMFS memory-only file system for all tests.

We ran each experiment in two configurations: one with no latency, and the other with 15 ms of latency added between client and server (for a 30 ms round-trip time). The former configuration represents the LAN environments in which current distributed file systems perform relatively well, and the latter configuration represents a wide-area link over which current distributed file systems perform poorly.



This figure shows the time to run the PostMark benchmark. Each value is the mean of 5 trials—the error bars are 90% confidence intervals. Note that the scale of the y-axis differs between the two graphs.

Figure 4: PostMark file system benchmark

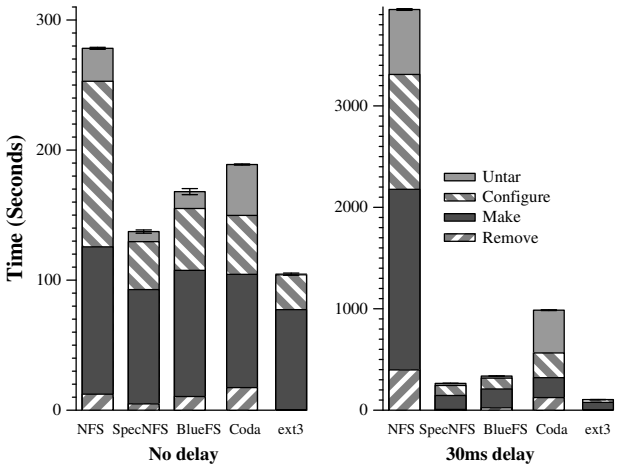
8.2 PostMark

We first ran the PostMark benchmark, which was designed to replicate the small-file workloads seen in electronic mail, netnews, and web-based commerce [16]. We used PostMark version 1.5, running in its default configuration that creates 500 files, performs 500 transactions consisting of file reads, writes, creates, and deletes, and then removes all files.

The left graph in Figure 4 shows benchmark results with no additional delay inserted between the file client and server. The difference between the first two bars shows that NFS is 2.5 times faster with speculation. This speedup is a result of using speculative group commit and the ability to pipeline previously sequential file system operations. Because PostMark is a single process that performs little computation, this benchmark does not show the benefit of propagating speculative state within the OS or the benefit of overlapping communication and computation.

The right graph in Figure 4 shows results with a 30 ms delay. The adverse impact of latency on NFS is apparent by the difference in scales between the two graphs: NFS without speculation is 41 times slower with 30 ms round-trip time than in a LAN environment. In contrast, SpecNFS is much less affected by network latency since it does not block on most remote operations. Thus, it runs the PostMark benchmark 24 times faster than NFS without speculation.

The benefits of speculative execution are even more apparent for BlueFS. BlueFS runs the PostMark benchmark 53% faster than the non-speculative version of NFS with no delay, and BlueFS is 49 times faster with a 30 ms delay. This performance improvement is realized even though BlueFS provides single-copy file semantics and synchronous I/O. Interestingly, BlueFS outperforms the speculative version of NFS with a 30 ms delay. This is attributable to two



This figure shows the time to untar, configure, make and remove the Apache 2.0.28 source tree. Each value is the mean of 5 trials—the error bars are 90% confidence intervals. Note that the scale of the y-axis differs between the two graphs.

Figure 5: Apache build benchmark

factors: BlueFS uses write-ahead logging to achieve better disk throughput, and NFS network throughput is limited by the legacy sunrpc package. The BlueFS server achieves better disk throughput than NFS because its use of a write-ahead log reduces the number of disk seeks compared to the Linux NFS server, which updates file data in place. The sunrpc package used by NFS has some synchronous behavior that we were unable to eliminate; for example, we could only have a maximum of 160 asynchronous RPCs in flight.

8.3 Apache build benchmark

We next ran a benchmark in which we untar the Apache 2.0.48 source tree into a file system, run `configure` in an object directory within that file system, run `make` in the object directory, and remove all files. During the benchmark, `make` and `configure` fork many processes to perform subtasks—these processes communicate via signals, pipes, and files in the /tmp directory. Thus, propagating speculative kernel state is important during this benchmark.

In Figure 5, each bar shows the total time to perform the benchmark, and shadings within each bar show the time to perform each stage. With no delay inserted between client and server, speculative execution improves NFS performance by a factor of 2. The largest speedup comes during the `configure` stage, which is 3.4 times faster. The bar on the far right shows the time to perform the benchmark on the local ext3 file system—this gives a lower bound for performance of a distributed file system. The potential speedup for `make` is limited since computation represents a large portion of the execution of that stage. However, speculative execution still improves `make` time by 30%. With 30 ms of delay, NFS is 14 times faster with speculation. The cost of cache coherence is quite high without speculation since every RPC incurs a 30 ms round-trip delay.

SpecNFS typically has less than 10 rollbacks during the Apache benchmark. When a file is moved to a new directory, the Linux NFS server assigns it a new file handle; when the client next tries to access the file by the old handle, the server returns ESTALE if the entry with the old file handle has been evicted from its file cache. The client then issues a `lookup` RPC to learn the new file handle. Since SpecNFS does not anticipate the file handle changing, an ESTALE response causes a rollback. Like the non-speculative version of NFS, SpecNFS learns the correct file handle when it issues a non-speculative `lookup` on re-execution.

BlueFS is 66% faster than non-speculative NFS with no delay; with a 30 ms delay, BlueFS is 12 times faster. These results demonstrate that, with speculative execution, it is possible for a distributed file system to be safe, consistent, *and* fast. BlueFS does not experience any rollbacks during this benchmark.

To provide a point of comparison with file systems that use callbacks, we ran the benchmark using version 6 of the Coda distributed file system; we executed the benchmark with Coda in strongly-connected mode (which provides AFS-like consistency guarantees). BlueFS, which provides stronger consistency, outperforms Coda by 12%. With 30ms of delay, BlueFS is 3 times faster than Coda. While we would need to implement a speculative version of Coda to determine precisely how much Speculator would improve its performance, our existing results provide some insight. Since Coda's use of callbacks results in fewer synchronous RPCs than NFS, we suspect that a speculative version of Coda would perform better than SpecNFS.

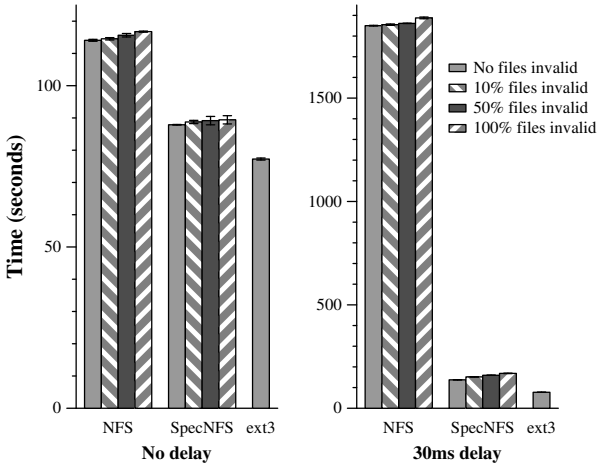
8.4 The cost of rollback

In the previous two benchmarks, almost all speculations made by the distributed file system are correct. We were therefore curious to measure the performance impact of having speculations fail. We quantified this cost by performing the `make` stage of the Apache benchmark while varying the percentage of out-of-date files in the NFS client cache.

To create out-of-date files, we append garbage characters to a randomly selected subset of the files in the Apache source directory—if one of these modified files were to be erroneously used during the `make`, the Apache build would fail. Before the experiment begins, the client caches all files in the source directory—this includes both the unmodified and modified files. We then replace each modified file with its correct version at the server. Thus, before the experiment begins, the client cache contains a certain percentage of out-of-date files that contain garbage characters.

Non-speculative NFS detects that the out-of-date files have changed when it issues synchronous `getattr` RPCs during file open. The client discards its cached copies of these files and fetches new versions from the server. SpecNFS continues to execute with the out-of-date copies and issues asynchronous `getattr` RPCs. When an RPC fails, it invalidates its cached copy and rolls back to the beginning of the `open` system call. In all cases, the compile completes successfully.

Figure 6 shows `make` time when approximately 10%, 50% and 100% of the files are initially out-of-date. As the number of out-of-date files grows, the time to complete the benchmark increases slightly due to the time needed to fetch up-to-date file versions from the server. This delay is small since network bandwidth is high—however, this cost must be incurred by both speculative and non-speculative versions of NFS. Even when all files are initially out-of-date, SpecNFS still substantially outperforms NFS without speculative execution: SpecNFS is 31% faster with no delay and over 11



This figure shows the time to make Apache 2.0.28 with different percentages of files out-of-date in the client cache. Each value is the mean of 5 trials—the error bars are 90% confidence intervals. Note that the scale of the y-axis differs between the two graphs.

Figure 6: Measuring the cost of rollback

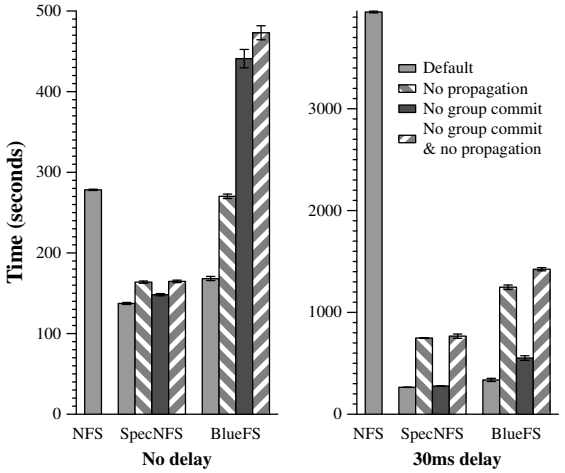
Files out of date	No delay		30 ms delay	
	Rollbacks	Commits	Rollbacks	Commits
0	0	25973	0	25739
80	40	25870	40	26446
374	183	25869	183	26803
758	362	25831	358	27451

This figure shows the average number of speculations that roll back and commit during the `make` stage of the Apache benchmark when different numbers of source files are initially out-of-date in the client cache.

Figure 7: Rollback benchmark detail

times faster with 30 ms delay. These improvements are comparable to results with no files out-of-date (30% and 13 times faster, respectively). Thus, the impact of failed speculations is almost imperceptible in this benchmark. However, the impact could be greater on a loaded system with less spare CPU cycles for speculation, or when the network bandwidth between client and server is limited.

Figure 7 shows more detail about these experiments. The first column shows the number of out-of-date files in the source directory actually accessed by `make`. The remaining columns show the average number of speculations that roll back and commit during execution of the benchmark. We were initially surprised to see that the number of rollbacks is substantially less than the number of out-of-date files accessed by the benchmark, meaning that several out-of-date files do not cause a rollback. This disparity is caused by `make` accessing several out-of-date files in short succession. For example, consider a case where `make` reads the attributes of two out-of-date files before it receives the `getattr` response for the first file. The first response causes a rollback since the attributes in the client cache were incorrect. Since this rollback eliminates all state modified after the first request was issued, a rollback is not needed when the second `getattr` response is received. SpecNFS updates file attributes in



This figure shows how speculative group commit and propagating state within the kernel impact the performance of speculative file systems for the Apache benchmark. Each value is the mean of 5 trials—the error bars are 90% confidence intervals. Note that the scale of the y-axis differs between the two graphs.

Figure 8: Measuring components of speculation

the client cache based upon both responses. Thus, on re-execution, the new speculations for *both* files are correct. In this example, although two files are out-of-date, the client only rolls back once. In effect, this is a type of prefetching of out-of-date file attributes, somewhat similar to the prefetching of file data proposed by Chang and Gibson [6].

8.5 Impact of group commit and sharing state

Figure 8 isolates the impact of two ideas presented in this paper. The left bar of each graph shows the time needed by the non-speculative version of NFS to perform all four stages of the Apache benchmark described in Section 8.3. The next two datasets show the performance of SpecNFS and BlueFS. For each file system, we show the default configuration (with full support for speculative execution), and three configurations that omit various parts of Speculator.

The second bar in each data set shows performance when Speculator does not let processes share speculative state through kernel data structures. This implementation blocks speculative processes that read or write to pipes and files in RAMFS, send signals to other processes, fork, or exit. Only speculative state in the distributed file system is shared between processes. The third bar in each data set shows the performance of speculative execution when the server does not allow speculative group commit. The last bar shows performance when both speculative group commit and sharing of speculative state are disallowed.

The benefit of propagation is most clearly seen with 30 ms delay. Blocking until speculations are resolved is more costly in this scenario since sever replies take longer to arrive. The benefit of speculative group commit is less dependent on latency. Without propagation and speculative group commit, the improved consistency and safety of BlueFS are much more costly. Especially on low-latency networks, both improvements are needed for good performance.

9. RELATED WORK

To the best of our knowledge, Speculator is the first support for multi-process speculative execution in a commodity operating system and the first use of speculative execution to improve cache coherence and write throughput in distributed file systems.

Chang and Gibson [6] and Fraser and Chang [10] use speculative execution to generate I/O prefetch hints for a local file system. In their work, the speculative thread executes only when the foreground process blocks on disk I/O. When the speculative thread attempts to read from disk, a prefetch hint is generated and fake data is returned so that the thread can continue to execute. Their work improves read performance through prefetching, whereas Speculator improves read performance by reducing the cost of cache coherence. Speculator also allows write-path optimizations such as group commit. In Speculator, speculative processes commit their work in the common case where speculations are correct. However, since Chang's speculative threads do not see correct file data, any computation done by a speculative thread must be later re-done by a non-speculative thread. Speculator also allows multiple processes to participate in a speculation; Chang's speculative threads are not allowed to communicate with other processes.

Time Warp [15] explored how speculative execution can improve the performance of distributed simulations. Time Warp is specialized for executing simulations, and imposes certain restrictions that may be onerous to developers of general applications: processes must be deterministic, cannot use heap storage, and must communicate via asynchronous messages. Speculator is targeted for a general-purpose operating system and does not impose these restrictions. Time Warp's abstraction of virtual time [14] tracks causal dependencies using Lamport's clock conditions [19]. Speculator's dependency lists are more general than a one-dimensional value. This generality is most useful with concurrent speculations where a one-dimensional time value creates an unnecessary dependency between two independent events. In contrast to Time Warp, the client-server nature of distributed file systems allows Speculator to simplify its handling of speculative state: failed speculative state created by a client is never committed at the server, nor is it visible to other clients. Speculator's design for buffering output was inspired by Time Warp's handling of external output.

Common applications of speculative execution in processor design range from branch prediction to cache coherence in multiprocessors. Steffan et al. [31] have investigated the use of speculation to extract greater parallelism from application code. Similar proposals advocating hardware speculation are [9, 12, 33]. The difference between implementing speculation in the processor and in the OS is the level of granularity: OS speculation can be applied at much higher levels of abstraction where processor-level speculation is inappropriate.

Speculative execution provides a limited subset of the functionality of a transaction. Thus, transactional systems such as QuickSilver [26], Locus [32], or TABS [28] could potentially supply the needed functionality. However, the overhead associated with providing atomicity and durability might preclude transactions from achieving the same performance gains as speculative execution.

Checkpointing and rollback-recovery are well-studied [8]. However, most prior work in checkpointing focuses on fault tolerance; a key difference in Speculator is that checkpoints capture only volatile and not persistent state. In this manner, they are similar to the lightweight checkpoints used for debugging by Srinivasan et al. [29]. Previous work in logging virtual memory [7] might offer performance optimizations over Speculator's current fork-based checkpoint strategy. Causal dependency tracking has been applied in several other projects, including QuickSilver [26], BackTracker [17], and the Repairable File Service [34].

Prior distributed file systems have provided single-copy file semantics—examples include Sprite [23] and Spritely NFS [30]. However, these semantics came at a performance cost. On the other hand, Liskov and Rodrigues [21] demonstrated that read-only transactions in a file system can be fast, but only by allowing clients to read slightly stale (but consistent) data. A major contribution of Speculator is showing that strong safety and cache coherence guarantees can be provided with minimal performance impact. In this manner, it is possible that speculative execution might improve the performance of distributed file systems such as BFS [5], SUNDR [20], and FARSITE [1] that require additional communication to deal with faulty or untrusted servers.

10. CONCLUSION

Speculator supports multi-process speculative execution within a commodity OS kernel. In this paper, we have shown that Speculator substantially improves the performance of existing distributed file systems. We have also shown how speculation enables the development of new file systems that are safe, consistent, *and* fast, even over high-latency links. Our modified version of BlueFS provides single-copy file semantics and synchronous I/O, yet still performs better than current file systems. While our investigation to date has focused on distributed file systems, we believe that generic OS support for speculative execution and causal dependency tracking will prove useful in many other domains. Our future plans therefore include investigating what other applications can benefit from Speculator.

Acknowledgments

We thank Manish Anand, Dan Peek, Ya-Yunn Su, the anonymous reviewers, and especially our shepherd, Frans Kaashoek, for suggestions that improved the quality of this paper. Jan Harkes and Mahadev Satyanarayanan helped explain details of the Coda file system. We used David A. Wheeler's SLOCCount to estimate the lines of code for our implementation. The work is supported by the National Science Foundation under award CCR-0509093. NSF has also supported development of the Blue File System through award CNS-0306251. Jason Flinn is supported by NSF CAREER award CNS-0346686. Intel Corp and Motorola Corp have provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, Motorola, the University of Michigan, or the U.S. government.

11. REFERENCES

[1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 1–14.

[2] Birrell, A. D., Hisgen, A., Jerian, C., Mann, T., and Swart, G. The Echo distributed file system. Tech. Rep. 111, Digital Equipment Corporation, Palo Alto, CA, USA, October 1993.

[3] Callaghan, B., Pavlowski, B., and Staubach, P. NFS Version 3 Protocol Specification. Tech. Rep. RFC 1813, IETF, June 1995.

[4] Carson, M. *Adaptation and Protocol Testing thorugh Network Emulation*. NIST, http://snad.ncsl.nist.gov/itg/nistnet/slides/index.htm.

[5] Castro, M., and Liskov, B. Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).

[6] Chang, F., and Gibson, G. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), pp. 1–14.

[7] Cheriton, D., and Duda, K. Logged virtual memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995), pp. 26–39.

[8] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (September 2002), 375–408.

[9] Franklin, M., and Sohi, G. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers 45*, 5 (May 1996), 552–571.

[10] Fraser, K., and Chang, F. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the 2003 USENIX Technical Conference* (San Antonio, TX, June 2003), pp. 325–338.

[11] Haerder, T., and Reuter, A. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys 15*, 4 (December 1983), 287–317.

[12] Hammond, L., Willey, M., and Olukotun, K. Data speculation support for a chip multiprocessor. In *Proc. of the 8th Intl. ACM Conf. on Arch. Support for Programming Languages and Operating Systems* (San Jose, CA, October 1998), pp. 58–69.

[13] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[14] Jefferson, D. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 404–425.

[15] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., P.Hontalas, Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Weidel, J., Younger, H., and Bellenot, S. Time Warp operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX, November 1987), pp. 77–93.

[16] Katcher, J. PostMark: A new file system benchmark. Tech. Rep. TR3022, Network Appliance, 1997.

[17] King, S. T., and Chen, P. M. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 223–236.

[18] Kistler, J. J., and Satyanarayanan, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[19] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[20] Li, J., Krohn, M., Mazières, D., and Shasha, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 121–136.

[21] Liskov, B., and Rodrigues, R. Transactional file systems can be fast. In *Proceedings of the 11th SIGOPS European Workshop* (Leuven, Belgium, September 2004).

[22] Mazières, D., Kaminsky, M., Kaashoek, M. F., and Witchel, E. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, December 1999), pp. 124–139.

[23] Nelson, M. N., Welsh, B. B., and Ousterhout, J. K. Caching in the Sprite network file system. *ACM Transactions on Computer Systems 6*, 1 (1988), 134–154.

[24] Nightingale, E. B., and Flinn, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 363–378.

[25] Rosenblum, M., Bugnion, E., Herrod, S. A., Witchel, E., and Gupta, A. The impact of architectural trends on operating system performance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 285–298.

[26] Schmuck, F., and Wyllie, J. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 239–53.

[27] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and Noveck, D. Network File System (NFS) version 4 Protocol. Tech. Rep. RFC 3530, IETF, April 2003.

[28] Spector, A. Z., Daniels, D., Duchamp, D., Eppinger, J. L., and Pauch, R. Distributed transactions for reliable systems. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA, December 1985), pp. 127–146.

[29] Srinivasan, S., Andrews, C., Kandula, S., and Zhou, Y. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference* (Boston, MA, June 2004).

[30] Srinivasan, V., and Mogul, J. Spritely NFS: Experiments with cache consistency protocols. In *Proceedings of the 12th ACM Symposium on Operating System Principles* (December 1989), pp. 45–57.

[31] Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)* (Vancouver, Canada, June 2000), pp. 1–24.

[32] Weinstein, M. J., Thomas W. Page, J., Livezey, B. K., and Popek, G. J. Transactions and synchronization in a distributed operating system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA, December 1985), pp. 115–126.

[33] Zhang, Y., Rauchwerger, L., and Torrellas, J. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proc. of the 5th Intl. Symposium on High Performance Computer Architecture* (Orlando, FL, January 1999), p. 135.

[34] Zhu, N., and Chiueh, T. Design, implementation and evaluation of the Repairable File Service. In *Proceedings of the International Conference on Dependable Systems and Networks* (San Francisco, CA, June 2003).