The *Conquest* File System: Better Performance Through a Disk/Persistent-RAM Hybrid Design

AN-I ANDY WANG
Florida State University
GEOFF KUENNING
Harvey Mudd College
and
PETER REIHER and GERALD POPEK
University of California, Los Angeles

Modern file systems assume the use of disk, a system-wide performance bottleneck for over a decade. Current disk caching and RAM file systems either impose high overhead to access memory content or fail to provide mechanisms to achieve data persistence across reboots.

The *Conquest* file system is based on the observation that memory is becoming inexpensive, which enables all file system services to be delivered from memory, except for providing large storage capacity. Unlike caching, *Conquest* uses memory with battery backup as persistent storage, and provides specialized and separate data paths to memory and disk. Therefore, the memory data path contains no disk-related complexity. The disk data path consists of optimizations only for the specialized disk usage pattern.

Compared to a memory-based file system, *Conquest* incurs little performance overhead. Compared to several disk-based file systems, *Conquest* achieves 1.3x to 19x faster memory performance, and 1.4x to 2.0x faster performance when exercising both memory and disk.

Conquest realizes most of the benefits of persistent RAM at a fraction of the cost of a RAM-only solution. It also demonstrates that disk-related optimizations impose high overheads for accessing memory content in a memory-rich environment.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Storage hierarchies*; D.4.3 [**Operating Systems**]: File System Management—*Access methods; directory structures*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms: Design, Experimentation, Measurement, Performance

This work was supported by the National Science Foundation under grant no. CCR-0098363.
Authors' addresses: A.-I. A. Wang, Department of Computer Science, Florida State University, Tallahassee, FL 32306; email: awang@cs.fsu.edu; G. Kuenning, Computer Science Department, Harvey Mudd College, CA 91711; email: geoff@cs.hmc.edu; P. Reiher and G. Popek, Computer Science Department, University of California, Los Angeles, CA 90095; email: {reiher, popek}@cs.ucla.edu. G. Popek is also associated with United On-Line.

Additional Key Words and Phrases: Persistent RAM, file systems, storage management, performance measurement

1. INTRODUCTION

For over 25 years, disk has been the dominant storage medium for most file systems. Although disk storage capacity is advancing at a rapid rate, the mechanical latency of disk has improved only at 15% per year compared to the 50% per year speed improvements of memory and CPU. Within the past ten years, these differences in access rates have widened the performance gap between disk and CPU from five to six orders of magnitude.

The *Conquest* disk/persistent-RAM hybrid file system addresses the performance problem of disk. The key observation is that the cost of persistent RAM (e.g., battery-backed DRAM) is declining rapidly, and the assumption of RAM as a scarce resource is becoming less true for average users. *Conquest* explores these emerging memory-rich environments and their effects on file system architecture and better performance.

Compared to disk-based file systems, *Conquest* achieves 1.3× to 19× faster memory performance, and 1.4× to 2.0× faster performance when exercising both memory and disk. The *Conquest* experience also teaches the following lessons: (1) Current operating systems have a deep-rooted assumption of high-latency storage throughout the computing stack, which is difficult to bypass or remove; (2) File systems designed for disks fail to exploit the full potential of memory performance in a memory-rich environment. (3) separating data paths into low and high-latency storage and matching workload characteristics to appropriate storage, media can yield significant performance gains and data path simplifications.

1.1 The Emergence of Persistent RAM

Researchers have long been seeking alternative storage media to overcome the deficiencies of disks [Baker et al. 1992; Douglis et al. 1994; Miller et al. 2001]. Recently, persistent RAM has emerged as a good candidate.

Typically, persistent RAM can be classified into flash RAM and battery-backed DRAM (BB-DRAM). Both forms of persistent RAM can deliver two to six orders of magnitude faster access times than disks. Flash RAM is mostly used for mobile devices because of its ability to retain information with very low power. However, flash memory has a number of limitations: (1) Each memory location of a flash RAM is limited in terms of the number of times it can be written and erased, so that flash is not suitable for update-intensive loads; (2) the erasure time is in the range of seconds [Cáceres et al. 1993]; and (3) the density of flash memory storage is low compared to DRAM, and its physical size imposes limitations for deployment on general-purpose machines.

BB-DRAM can operate at the speed of DRAM for all operations and general workloads, but it requires a constant supply of power for persistent storage. Fortunately, this power can easily be supplied by an uninterruptible power supply (UPS) or on-board rechargeable batteries [PC World 2005].

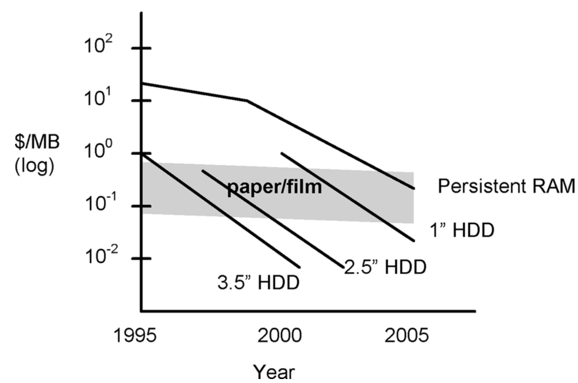


Fig. 1. Average price of storage. The shaded area shows the cost of paper and film storage as a comparison [Grochowski and Halem 2003].

Although a RAM-only storage solution can simplify the file system and provide improved performance, cost is still a concern. Figure 1 shows that even with their accelerated price decline after 1998, flash RAM and BB-DRAM are unlikely to match disks economically in the short run.

However, when the cost of various storage technologies is compared to that of paper and film, we can make an interesting observation: Historically, paper and film costs have represented an approximate barrier for market penetration. For example, disks with various geometries have gained wide acceptance as they cross this barrier [Grochowski and Halem 2003] (we do not attempt to explain the barrier in physical or economic terms, and are not necessarily convinced that there is any relationship between paper and memory costs; we simply observe that there is a price point below which technologies tend to gain acceptance). Currently, persistent RAM is crossing this barrier and thus becoming an affordable storage option. Also, we are seeing high-end machines equipped with 4 to 10GB of RAM that can potentially be converted into persistent storage. Therefore, a transitional approach to improving file systems combines the use of RAM and disk storage in an innovative way.

1.2 *Conquest* Approach

Conquest is a disk/persistent-RAM hybrid file system that delivers all file system services from persistent RAM, with the single exception that high-capacity storage is still provided by traditional disks. In essence, *Conquest* provides two specialized and simplified data paths to persistent-RAM and disk storage: (1) It stores all small files and metadata (e.g., directories and file attributes) in RAM; and (2) disk holds only the data content of the remaining large files (with their metadata stored in persistent RAM).

By partitioning data in this fashion, *Conquest* performs all file system management on memory-resident data structures, thereby minimizing disk accesses. Tailoring both file system data structures and management to the physical characteristics of memory significantly improves performance, as compared to disk-only designs. In addition, traversing the memory data path incurs no disk-related overhead, and the disk data path consists of only the processing costs for handling access patterns that are suitable for disk.

Another benefit of *Conquest* is its ability to provide a smooth and cost-effective transition from disk-based to persistent-RAM-based storage. Unlike other memory file systems [McKusick et al. 1990; Douglis et al. 1994; Wu and Zwaenepoel 1994], *Conquest* allows persistent RAM to assume more file system responsibility as memory prices decline. Thus, it can achieve most of the benefits of persistent RAM, without the high cost of RAM-only solutions.

2. COMMON ALTERNATIVES

The memory-rich environment significantly departs from the conventional mindset of RAM-scarce environments. Therefore, simple solutions derived from the old view fail to take complete advantage of new possibilities. In many cases, extensions of these simple methods can give results similar to that of the *Conquest* approach, but these extensions add so much complexity that they cease to be attractive alternatives.

This section discusses these approaches and their limitations. Some do not provide the expected performance gains, while others do not completely solve the problem of storing arbitrary amounts of data persistently, reliably, and conveniently. Rather than adding the complications necessary to fix these approaches, a better approach is to begin the design with a clean slate.

Caching. The most popular and effective approach to improving disk access speed is the disk buffer cache, which fulfills disk requests from memory whenever possible. As a common approach, replacing the least recently used (LRU) memory blocks with newly read blocks from disk can keep the disk cache populated with recently and frequently referenced disk content. Variants of this approach would seem to be an attractive alternative to *Conquest*, since adding memory requires no changes to the operating system. With a few changes to the cache replacement policy, it might even be possible for the existing disk buffer cache to behave like *Conquest*, leaving only the data content of large files on disk.

However, as memory access is becoming the common case in memory-rich environments, the data path of caching is still tailored for eventual disk accesses. Our results show that even when all accesses are fulfilled by cache, the performance penalty can be up to a factor of 3.5 (Section 7.2). Therefore, altering modern cache policy to either cache only small files and metadata or buffer writes indefinitely will not exploit the full performance potential of memory.

RAM Drives and RAM File Systems. *RAM drives* are reserved memory that is accessed through the device interface. They are attractive because they follow existing file system semantics and interfaces, and a RAM device is formatted and mounted just as if it were a disk. However, a RAM drive is still subject to all disk-related management—such as caching. Therefore, a piece of data can be both stored in the RAM drive and cached in the disk buffer cache, doubling memory consumption and halving the access bandwidth.

A *RAM file system* can be implemented as a kernel module, without the need for an associated device. RAM file systems under Linux and BSD [McKusick et al. 1990] are built on top of various temporary caches in the virtual file system interface (VFS) [Kleiman 1986] so that memory consumed by the RAM file system can be dynamically allocated as needed. By saving both data and metadata in various caches, RAM file systems avoid the duplicate memory consumption of RAM drives and can be expected to perform at the speed of disk caching. In practice, RAM file systems tend to perform even faster, due to the lack of metadata and data commits to disk (Section 7.1.2).

At first glance, RAM drives or RAM file systems appear to be ideal ready alternatives to disks in memory-rich environments. However, neither provides persistence of data across reboots. Although persistent RAM provides non-volatility of memory content, persistence also requires a protocol for storing and retrieving the information from the persistent medium so that both the file system and the memory manager know how to resurrect information from the storage medium across reboots.

For RAM drives, a protocol exists for storing and retrieving the in-memory information, but there is no protocol to ensure that states within the memory manager will survive reboots. Isolating these states is nontrivial, given that the existing memory manager makes no distinctions between persistent and temporary states. For RAM file systems, since the memory manager is unaware of the data content stored under various VFS caches, neither the file system nor memory states can survive reboots without significant modifications to the system.

Both RAM drives and RAM file systems also incur unnecessary disk-related overhead. On RAM drives, existing file systems (tuned for disk) are installed on the emulated drive, despite the absence of the mechanical limitations of disks. For example, the access to RAM drives is made in blocks, and the file system will attempt to place files in "cylinder groups," even though cylinders and block boundaries no longer exist.

Although RAM file systems have eliminated some of these disk-related complexities, they rely on VFS and its generic storage access routines wherein built-in mechanisms such as readahead and buffer cache reflect the assumption that the underlying storage medium is slower than memory, leading to lower performance.

In addition, both RAM drives and RAM file systems limit the size of the files to that of main memory. These restrictions have limited the use of RAM drives and RAM file systems to caching and temporary file systems. To move to a general-purpose persistent-RAM file system, we need a substantially new design.

Disk Emulators. To speed up deployment without kernel modifications, some manufacturers offer RAM-based disk emulators [BitMicro 2005]. These emulators generally plug into a standard SCSI or similar I/O port and look exactly like a disk drive to the CPU. Although they provide a convenient solution to those who need an instant speedup, and they do not suffer the persistence problem of RAM drives, they are again only an interim solution that neither addresses the underlying problem and nor takes advantage of the unique benefits of RAM. All of the drawbacks of RAM drives apply, and in addition, the use of standardized I/O interfaces forces emulators to employ inadequate access methods and low-bandwidth cables, greatly limiting their utility as anything other than a stopgap measure.

Customized Memory Filing Services Within Applications. Some storage-intensive applications (e.g., databases) use their own in-memory filing services to avoid accessing disks. By accessing files within a process's address space, this approach can avoid the performance penalties of kernel crossings and system calls, as well as expensive disk accesses. Also, since this approach involves no changes to the underlying kernel, modified applications are more portable to other operating system platforms.

This approach has two major drawbacks. First, application designers need to construct their own filing and memory services, which already offered at the operating system level, not to mention the possible redundant efforts among similar applications. Second, it requires access to the source code for modifications, which is not practical for legacy applications and programs whose source is unavailable.

Ad-Hoc Approaches. There are also less structured approaches to using existing tools for exploiting the abundance of RAM. *Ad hoc* approaches are attractive because they can often avoid modifying the operating system. Also, the design, development, and deployment cycles for *ad hoc* solutions can be significantly shorter than an approach that involves complete redesign. For example, we could achieve persistence by manually transferring files into a RAM file system at boot time, preserving them again before shutdown. However, this method would require an end user to identify the set of files that are active and small enough to fit into the memory. The user also needs to be aware of whether doing so will yield enough benefit for the particular set of files.

Another option is to manage RAM space by using a background daemon to stage files to a disk partition. However, this approach would require significant complexity to maintain a single name space (as does *Conquest*), and to preserve the semantics of links when moving files between storage media. Also, since RAM and disk are two separate devices, the design must handle the semantics where one of the devices is not mounted. A simple approach is to manage both RAM and disk file systems with the semantics of a single file system; however, a single caching policy specified at mount time cannot satisfy both RAM and disk file systems, since caching contributes little towards accelerating RAM accesses and wastes memory resources. Separate caching policies demand that a file be able to change caching status dynamically. As the details of an *ad hoc* approach grow, the resulting complexity is likely to match or exceed that of *Conquest*, without achieving *Conquest* performance.

3. *CONQUEST* FILE SYSTEM DESIGN

Conquest's design assumes the popular single-user desktop hardware environment enhanced with 1 to 4GB of persistent RAM. As of July 2005, we can add 2GB of battery-backed RAM to our desktop computers and deploy *Conquest* for around $300 [PC World 2005; Price Watch 2005]. Extending the *Conquest* design to other environments, such as laptops and distributed systems, will be future work. This section first presents the design overview of *Conquest*

(Section 3.1), followed by a discussion of various major design decisions (Section 3.2 and Sections 4 to 6).

3.1 File System Design

Conquest stores small files and metadata in persistent RAM; disk holds only the data content of large files. Section 3.2 will further discuss this storage delegation strategy.

An in-memory file is logically stored contiguously in persistent RAM. Disks store the data content of large files with coarse granularity, thereby reducing management overhead. For each large file, *Conquest* maintains a segment table in persistent RAM that tracks segments of data on disk. The on-disk allocation is done contiguously whenever possible, and the data layout is similar to a variant of the Berkeley fast file system (FFS) [McKusick et al. 1984; Peacock et al. 1998].

For each directory, *Conquest* uses a dynamically allocated extensible hash table [Fagin et al. 1979] to maintain metadata entries and retain the directory file pointer semantics (Section 4.2.2). Hard links are trivially supported by hashing multiple names to the same file metadata entry.

RAM storage allocation reuses the existing memory manager [Peterson and Norman 1977; Bonwick 1994] to avoid duplicate functionality. However, *Conquest* has its own dedicated instances of the manager, each governing its own memory region and residing persistently inside *Conquest*. Paging and swapping are disabled for *Conquest* memory, but enabled for the non-*Conquest* memory region for backward compatibility.

Unlike caching, RAM drives, and RAM file systems, *Conquest* memory is the final storage destination for small files and all metadata. A storage request can traverse the critical path of *Conquest*'s main store without such disk-related complexity as data duplication, migration, translation, synchronization, and associated management. *Conquest* also supports files and file systems that exceed the size of physical RAM.

Since *Conquest* follows the VFS interface, it has not changed the model of access controls, memory protection mechanisms, or the resulting reliability model. However, it applies the technique of soft updates [McKusick and Ganger 1999] and takes advantage of atomic memory operations to ensure the consistency of metadata. Updates to data structures are ordered in such a way that in the worst case, an interrupted file system update degenerates into a memory leak, which can be garbage collected periodically (note that the garbage collection process is not required for the correctness of file system operations). *Conquest* can either rely on backups or combine with Rio-like memory dump mechanisms [Ng and Chen 2001] to protect against battery failures.

3.2 Strategy for Delegating Storage Media

How to delegate the use of memory and disk is fundamental to *Conquest* design, and this decision has contributed most of the performance gain of *Conquest*. *Conquest*'s strategy for using storage media is based on a variety of studies of user access patterns and file size distributions. Recent studies [Douceur and

Bolosky 1999; Vogels 1999; Roselli et al. 2000; Evans and Kuenning 2002] independently confirm earlier observations [Ousterhout et al. 1985; Baker et al. 1991; Bozman et al. 1991; Irlam 1993]:

—Most files are small, and they consume a small fraction of the total disk storage.
—Most accesses are to small files.
—Most accesses are sequential.
—Most storage is consumed by large files, which are read most of the time. Large files are also becoming larger over time, as the popularity of multimedia files grows.

Although we could imagine many complex data placement algorithms (including an LRU-style migration of unused files to the disk), *Conquest* has taken advantage of the aforementioned characteristics by using a size threshold to choose which files are candidates for disk storage. Only the data content of files above the threshold is stored on disk; smaller files are stored entirely in RAM. Metadata entries for both large and small files are always stored in memory, even if the size of the directory exceeds the threshold. We have experimented with 0KB, 8KB, 64KB, 256KB, and 1MB thresholds, many of which work well for various benchmark workloads (Section 7). By varying the threshold from 64KB to 1MB, 96% to 99% of all files can be kept in RAM [Irlam 1993; Roselli et al. 2000]. By increasing this threshold, *Conquest* can use more RAM storage as its price declines. The current threshold was chosen somewhat arbitrarily; the future plan is to either leave this to system administrators or to dynamically control it with a user-level process.

The decision to use a threshold simplifies the code, yet does not waste an unreasonable amount of memory, since small files do not consume a large amount of the total space. An additional advantage of the size-based threshold is that all on-disk files are large, which allows us to achieve significant simplifications in disk management. For example, we can avoid adding complexity to handling fragmentation by using "large" and "small" disk blocks, as in FFS [McKusick et al. 1984]. Since we assume inexpensive and abundant RAM, the advantages of using a threshold far outweigh the small amount of space lost by storing rarely used small files in RAM.

The media delegation strategy for *Conquest* is not favorable for random seeks within large files; random seeks on disk are two orders of magnitude slower than sequential disk accesses. *Conquest* is not optimized for random seeks in the large files that are frequently observed in database applications. Rather, it covers common loads, such as sequential accesses to large multimedia, archives, compressed data objects, and associative accesses among small data objects (e.g., hypermedia and dynamic linked libraries).

3.2.1 *Files Stored in Persistent RAM.* Small files and metadata benefit the most from being stored in persistent RAM, given that they are more affected by disk latency. Figure 2 shows the data path for a conventional disk-based file system. A typical storage request begins by going through the I/O buffer

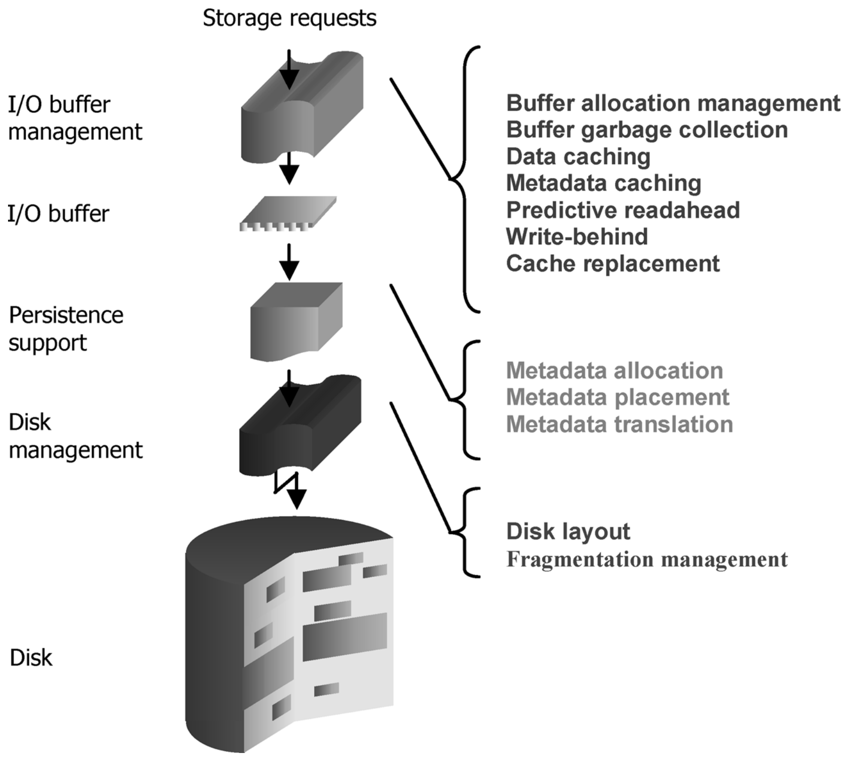


Fig. 2. The conventional data path for disk-based file systems.

management, which includes mechanisms to allocate, deallocate, and garbage-collect I/O buffers. The I/O buffer management also abstracts away file system functions such as speculative memory-management logic (e.g., predictive readahead) and the caching of data and metadata.

If the I/O buffer in the physical memory cannot fulfill the storage request, a file system has to locate the requested content by going through the persistence support component, which keeps track of the metadata and data locations on disk. For a typical disk-based file system, the persistence support needs to handle the allocation, deallocation, and placement of metadata on disk so as to translate the metadata between the run-time memory format and the on-disk, block-oriented, serialized format. The storage request must then locate the data by following the disk layout and consulting with the fragmentation manager to see if the data content is stored in a subblock. Finally, the disk scheduling system executes the request, possibly after delaying it to optimize head motion.

Figure 3 shows how our use of persistent RAM shortens the data path. For *Conquest*, memory accesses interact with the memory allocation manager directly and bypass the I/O buffer management (issues of reliability due to using memory for storage will be addressed in Section 6.1). Also, *Conquest* goes through a persistence support that requires less processing than that in a conventional file system, since metadata is stored in the run-time representation without need for translation into the disk representation. This simplification

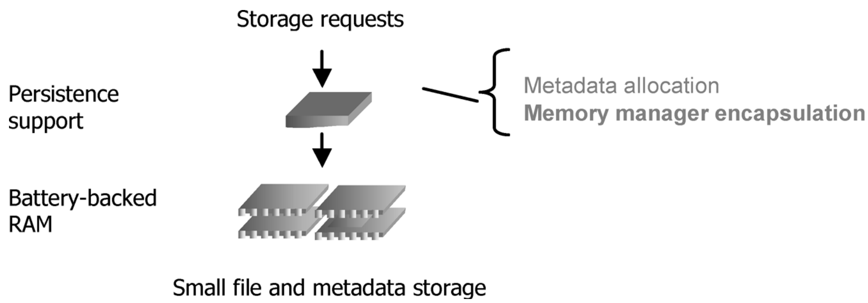


Fig. 3. The *Conquest* memory data path. *Conquest* has bypassed the I/O buffer and disk management. The persistence support under *Conquest* consists of a simplified metadata allocation component and mechanisms to encapsulate the memory manager.

also removes the mechanisms needed to propagate the metadata changes to disk [McKusick et al. 1984; Ganger et al. 2000; Seltzer et al. 2000; McKusick 2002].

3.2.2 *Large-File-Only Disk Storage.* Since small files are stored in persistent RAM, the disk data path can avoid small-file-related mechanisms such as storing the content of small files in the metadata directly, designing tailored trees to reduce the number of disk accesses before locating a small file, reducing disk fragmentations, and applying other seek time and rotational latency reduction methods [McKusick et al. 1984; Card et al. 1994; Namesys 2005].

With large-file-only disk storage, *Conquest* can use a coarser access granularity. Sequential-access-mostly large files exhibit well-defined readahead semantics. Large files are also read-mostly and incur little synchronization-related overhead. Combined with large data transfers and the lack of disk arm movements, disks can deliver near-raw bandwidth when accessing such files.

Figure 4 shows the disk data path of *Conquest*. We did not alter the VFS API for reasons that will be addressed in Section 6.4. Therefore, a typical disk request still goes through the I/O buffer management code under the VFS, with metadata caching still in place. However, given that the speculative memory-management logic is controlled at the file-system level, *Conquest* can exploit the access characteristics of sequentially accessed large files and reduce the complexity of predictive readahead, write-behind, and cache replacement policies. Since *Conquest*'s metadata is stored in memory, the disk data path can bypass the persistence support found in conventional file systems. In addition, *Conquest* stores only the data blocks of large files on disk. Disk management does not need to handle fragmentation or optimize layouts for small files.

For randomly accessed large files, the commonly used term "random" deserves reexamination. In the literature, an access is commonly defined as random if it is not sequential [Baker et al. 1991; Vogels 1999; Roselli et al. 2000]. This definition of random access may be misleading. For example, in the MP3 format, the title of a file is stored at the end of the file, and is usually accessed when the file is opened. But most other references to the file are sequential from the beginning. For video files, there is a relatively small number of scene changes that an end-user is likely to access, but within each scene, video frames

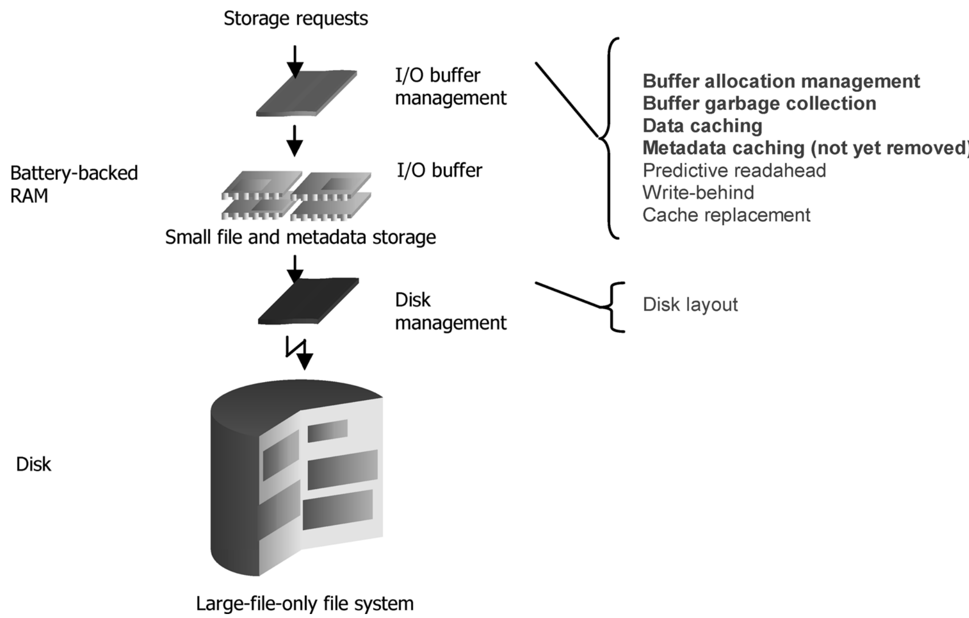


Fig. 4. The *Conquest* disk data path. *Conquest* has removed persistence support and disk fragmentation management. It has also simplified many disk-related components (not in bold type font).

will be viewed in sequential order. For such files, access is not truly random, but rather *near sequential*. With this observation, the metadata representation for large files can be greatly simplified, as described in the next section.

4. METADATA REPRESENTATION

The handling of file system metadata is critical, since this information is in the path of all file accesses. This section first describes how metadata are typically represented in legacy UNIX file systems, and then details how *Conquest* manages its metadata and data.

4.1 UNIX Metadata Representation

4.1.1 *Metadata Representation of a UNIX File.* The metadata for a UNIX file is represented with the legacy i-node data structure, whose design reflects the deep-rooted assumption of disk storage [Thompson 1978]. Although file systems have evolved through many generations, the original i-node design has changed little in the past 30 years [McKusick et al. 1984; Card et al. 1994].

The i-node under *ext2* contains 15 pointers used to track the locations of data blocks on disk. The first 12 pointer point to the first 12 data blocks. After consuming all 12 pointers, the 13th pointer points to a *single indirect block*, which in turn contains pointers to data blocks. The 14th pointer points to a *double indirect block*, which contains pointers to single indirect blocks. The 15th pointer points to a *triple indirect block*, which contains pointers to double indirect blocks.

This design allows small files to have fast access to data blocks, while infrequently accessed large files use the slower mechanism of traversing the nested indirect blocks. Block-based allocation avoids the need to manage external fragmentation, which can prevent the contiguous allocation of files that can occur even when space is available.

However, this design is limiting in several ways. First, optimizations for small file accesses complicate the data path for accessing large files. Second, although block-based allocation prevents external fragmentation, this design still needs to manage internal fragmentation (with inexpensive disk storage, managing internal fragmentation is done not so much to reduce the wasted storage as to improve the disk access bandwidth. Accessing loosely packed data blocks from small files can reduce disk bandwidth significantly, since disk transfers are at block granularity, even for a partially used block). Third, a data structure with full support for random accesses imposes unnecessary overhead and complexity for the common case of sequential large-file accesses. Finally, the total number of pointers provided by this data structure limits the size of the largest file.

4.1.2 *Metadata Representation of a UNIX Directory.* UNIX directories are represented as files whose data blocks contain a list of *directory entries* (names and i-node identification numbers) for the files and directories residing underneath. In most implementations, directory entries are stored in a variable length format so that files with shorter names consume less storage. For *ext2*, the size of a directory shrinks only when the entire directory is removed. If a file is removed, its directory entry is simply merged with the previous one by increasing its length.

The major advantage of this design is the reuse of the file abstraction to manipulate directories. However, the frequent operation of file lookup (e.g., `ls` and `dir`) within a directory generally requires linear searches.

4.2 *Conquest* Data and Metadata Representation

4.2.1 *Metadata and Data Representations for In-Memory Files.* *Conquest* removes nested indirect blocks from the commonly used i-node design. For in memory files, the data blocks are accessed through a uniform, single-level dynamically allocated index array in which each pointer points to one block to achieve logical contiguity.

Conquest does not use the v-node data structure provided by VFS to store metadata because the v-node is designed to accommodate different file systems with a wide variety of attributes, and *Conquest* does not need many of the mechanisms, such as metadata caching. *Conquest's* file metadata consists of only the fields (53 bytes) needed to conform to POSIX specifications.

To avoid extra metadata management, *Conquest* uses the memory addresses of metadata as unique IDs. When allocating metadata, the existing memory manager is invoked to allocate a memory region with the size of a file's metadata. Since no two *Conquest* i-nodes can have the same physical address, using the physical address as the metadata ID assures unique IDs. In addition, an ID allows the corresponding metadata be quickly located. The use of physical addresses is not that different from the way in which a disk refers to its physical block location, and virtual memory can still remap *Conquest*'s memory content to convenient locations. The downside of this design is that we need to modify the memory manager to anticipate that certain allocations will be relatively permanent.

For small in-memory write requests where the total allocation is unknown in advance, *Conquest* allocates data blocks incrementally. The current implementation does not return unused memory in the last block of a file, though we plan to add automatic truncation as a future optimization. *Conquest* also supports "holes" within a file, since they are commonly seen during compilation and other activities.

4.2.2 *Directory Representation.* The data structure design for directories needs to meet the following requirements: (1) preserving the legacy semantics of directories as files with file position pointers; (2) fast sequential retrieval of directory entries for common traversal operations (e.g., `ls` or `dir`); (3) fast random lookup (e.g., locating a file); and (4) management of hard links. To meet these requirements, we used a variant of extensible hashing [Fagin et al. 1979] for our directory representation. The directory structure is built with a hierarchy of hash tables, using file names as keys. Collisions are resolved by splitting (or doubling) hash indices and unmasking an additional hash bit for each key. A *path* (e.g., `/usr/bin`) is resolved by recursively hashing each name component of the path at each level of the hash table.

Extendible hashing preserves the ordering of hashed items when changing the table size, and this property allows `readdir()` to walk through a directory correctly while resizing a hash table (e.g., recursive deletions). Also, the use of hashing easily supports hard links by allowing multiple names to hash to the same file metadata entry. In addition, compared to *ext2*'s approach, hashing removes the need to compact directories that live in multiple (possibly indirect) blocks.

One concern with using extensible hashing is the wasted space due to unused hash entries. However, we found that alternative compact hashing schemes would consume a similar amount of space to preserve ordering during a resize operation.

4.2.3 *Metadata and Data Representation for On-Disk files.* For the metadata of on-disk files, the contiguous block segments of a file are tracked by a dynamically allocated segment table stored in persistent RAM, so that the maximum file size is limited only by physical storage. Locating a block involves sequentially finding a segment that contains the target block and adding an offset to the segment's starting block number. Disk storage is allocated contiguously whenever possible, in temporal order, similar to the hot-spot allocater used in a variant of the FFS [Peacock et al. 1998]. Temporal order is chosen since it correlates with spatial locality in many workloads.

Although *Conquest* currently has a linear search structure for disk storage, its simplicity and memory speed outweigh its algorithmic inefficiency, as demonstrated in our performance evaluation (Section 7). Given that *Conquest* has coarse disk allocation granularity, the segment table is likely to be small. As long as a file is not severely segmented, this in-memory search is sufficiently fast compared to the cost of disk access.

Conquest's design does not depend on any particular disk layout, so we can also borrow approaches from both video-on-demand (VoD) servers and traditional file systems research. For example, given its sequential access nature, a large media file can be striped across concentric disk zones so that disk scanning can serve concurrent accesses more effectively [Chen and Thapar 1997]. Frequently accessed large files can be stored near outer zones on disk platters for higher bandwidth. Spatial and temporal ordering can be applied within each disk zone at the granularity of an enlarged disk block.

Another example is to align allocated large-file data segments at disk track boundaries [Schindler et al. 2002]. Empirical measurements of this approach have reported improved disk access efficiency of up to 50% for mid-sized requests (100 to 500KB), a management granularity that matches *Conquest*'s large-file storage well.

With a variety of options available, the presumption is that after enlarging the disk access granularity for large-file accesses, disk transfer time will dominate access times. Since most large files are accessed sequentially, I/O buffering and simple predictive prefetching methods should still be able to deliver good read bandwidth.

5. *CONQUEST* PERSISTENCE SUPPORT

Since the metadata allocation of *Conquest* depends on a persistent association between the metadata ID and its physical memory address, *Conquest* needs additional mechanisms for the underlying memory manager to retain persistent states across reboots. Otherwise, at boot time, the operating system will reinitialize the memory manager and erase all information pertaining to prior allocations of *Conquest* metadata. While retaining information in persistent RAM seems simple, the design for *Conquest* persistence also needs to retain the legacy semantics of booting, where the content of volatile memory (or the content not pertaining to *Conquest*) must be properly reset. In this section, we discuss the existing memory manager under Linux 2.4.2 and describe the *Conquest* design.

5.1 Memory Manager under Linux 2.4.2

The memory manager under Linux 2.4.2 is structured in three layers (see Figure 5). The layer closest to the memory hardware is the page allocator, which tracks the allocation and memory attributes of individual pages. Locating a free memory region involves linear traversal of allocation bitmaps, structured in two levels. Therefore, as memory size increases, a linear-search-based page allocator becomes prohibitive.

The next layer is the zone allocator, which allocates memory zones for uses such as direct memory access, high memory, and I/O buffering. Within each zone, the zone allocator uses buddy allocation [Peterson and Norman 1977] to accelerate the allocation of memory pages in powers-of-two blocks. One problem

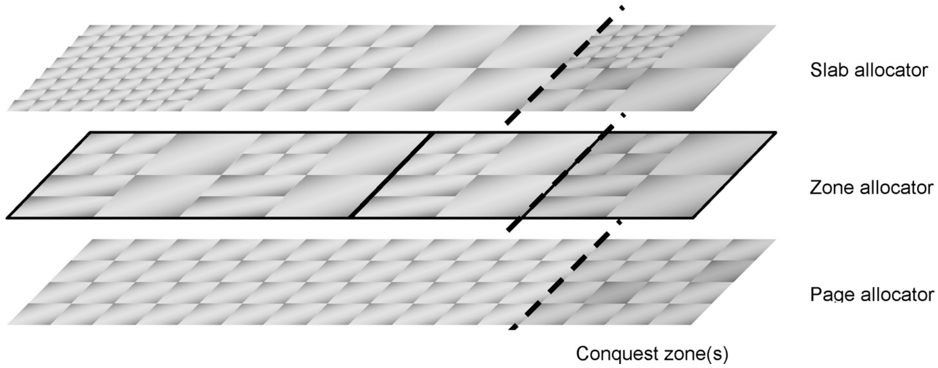


Fig. 5. *Conquest* memory manager.

naturally arising from the buddy allocation scheme is internal fragmentation within allocated pages, which leads to the need for a higher-level slab allocator [Bonwick 1994].

The slab allocator provides an efficient means of allocating memory at sub-page granularities and reducing internal memory fragmentation by allocating one page at a time and filing it with objects of a single type, initialized in bulk. For example, when *Conquest* allocates an *i*-node for the first time, the slab allocator will allocate a full page filled with initialized *Conquest* *i*-nodes, amortizing the allocation overhead.

5.2 *Conquest*'s Persistence Support

With the existing memory manager architecture, preserving persistent states across reboots is difficult for two reasons: (1) *Conquest*'s persistence support needs to preserve the mapping among three layers of memory management. For example, pages allocated for the slab allocator may belong to different instances of a buddy allocator under different zones. (2) Existing memory manager layers have no notion of persistence, hence temporary and persistent allocations are intermingled within three layers of complex data structures. Therefore, the zone manager may decide to "borrow" persistent memory for temporary uses and vice versa.

Conquest allocates its own memory zones, each containing an instantiation of the Linux memory manager (see Figure 5). Swapping and paging are disabled for *Conquest* zones to reduce their associated overheads, but enabled for non-*Conquest* zones for backward compatibility (i.e., for memory-intensive applications). *Conquest* zones also map well to separate uses of BB-DRAM and volatile RAM.

By reusing the existing Linux code base, *Conquest* can avoid building additional components. Concerns regarding memory bugs and reliability are addressed in Section 6.1. The reuse of the existing memory manager also implies compliance with the existing memory interface and leverage of all existing memory services. *Conquest* memory regions can be allocated through `kmalloc()` calls with additional *Conquest* flags. Memory fragmentation management is handled by the existing slab allocator design.

By having a separate instance of the memory manager residing within the *Conquest* memory zone that it governs, all pointers within these manager layers can be preserved across reboots due to the following invariants: (1) These pointers use physical memory addresses that are unchanged across reboots; and (2) they only point to physical addresses within the governed memory zone. Both invariants can be trivially satisfied and verified. The resulting encapsulation avoids complex serialization and logging code, and the runtime data structures of the *Conquest* memory manager can survive across reboots. Zone-based isolation also simplifies the semantics when *Conquest* is not mounted. A system can use the remaining memory resources to operate, without *Conquest*-related overhead.

6. *CONQUEST* DEPLOYMENT CONSIDERATIONS

Although *Conquest*'s design overcomes the cost constraints of persistent RAM, its practical deployment also relies on the proper handling of storage reliability (Section 6.1), memory depletion (Section 6.2), and system-wide data migration (Section 6.3). In addition, *Conquest* implementation needs to minimize changes to the existing kernel structure so that the resulting code is maintainable as the underlying kernel evolves. Finally, Section 6.4 discusses the current implementation status.

6.1 Reliability

Storing persistent data in memory inevitably raises concerns of reliability and data integrity. In general, disk storage is less vulnerable to corruption by software failures because it is less likely to perform illegal operations through the rigid disk interface. Main memory has a very simple load/store interface that allows a greater risk of corruption. A single wild kernel pointer could destroy many important files. However, we have found that memory can be reliable enough as a persistent storage medium in the sense that the use of common disk reliability techniques (e.g., backup) can provide a similar level of protection against data loss and failures.

6.1.1 *Comparison to Disk-Based File Systems.* Although conventional file systems use disk as the primary persistent storage medium, a significant portion of memory is used for caching data and metadata. The integrity of the memory content of disk-based file systems is protected at multiple fronts: the VFS interface, access control mechanisms within the VFS, and the underlying memory protection. For example, a misbehaving application owned by `root` can trespass the VFS interface and the access control mechanisms, but will be trapped when making an illegal access to a memory block. From this perspective, *Conquest* offers the same reliability model as disk-based file systems, since it does not alter any VFS mechanisms that ensure the integrity of memory content.

At the kernel level, operating system crashes raise another threat to reliability. However, the Rio file cache [Chen et al. 1996; Ng and Chen 2001] has demonstrated that memory can serve as reliable storage by examining 650 induced operating system crashes ranging from bit errors in the kernel stack to deleting branch instructions and C-level allocation management errors. The researchers discovered that 1.1% of the crashes corrupted the data on disk, compared to 1.5% for memory corruptions. Assuming one system crash every two months, we can expect to lose in-memory data (to the extent allowed by the aforementioned memory protection mechanisms) about once a decade [Ng et al. 1996]. Rio can be used as a reliability measure to complement *Conquest*'s streamlining of the memory data path, since the UPS or on-board memory provides enough power (5 minutes to 12 hours [APC 2005; PC World 2005]) to stage the memory content to disk.

At the hardware level, modern disks have a mean time between failures (MTBF) of 1 million hours [Seagate 2003]. Two hardware components, the RAM and battery backup system, cause *Conquest*'s MTBF to differ from that of a disk. Currently, *Conquest* uses a UPS as the battery backup. The MTBF of a UPS is lower than those of disks, but remains around 170,000 hours [Gibson and Patterson 1993; Liebert Cooperation 2005]. The MTBF of the RAM is comparable to disk [Micron 1997]. However, the MTBF of *Conquest* is dominated by the characteristics of the complete computer system; modern machines again have an MTBF of over 20,000 to 87,000 hours [Miles 2000; Dell 2002]. Thus, it can be seen that, at most, a machine using *Conquest* should lose data due to hardware failures only once every few years. For common users, this level of reliability is well within the acceptable range when standard backup procedures are used. Also, for high-end servers, dual power supplies and UPS units are readily available. When interconnected properly, the redundancy of power supplies and UPS units can further reduce the chance of single-point failures.

6.1.2 *Soft Updates and Pointer-Switch Commits.* In addition to a low memory corruption rate, *Conquest* also relies on other techniques to enhance reliability. For example, it applies the rules of soft updates [McKusick and Ganger 1999] and uses pointer assignment to atomically commit updates as follows: (1) Never point to a structure before it has been initialized (e.g., an *i*-node must be initialized before a directory entry references it); (2) never reuse a resource before nullifying all previous pointers to it (e.g., an *i*-node's pointer to a data block must be nullified before this block can be reallocated for a new *i*-node); and (3) never reset the old pointer to a live resource before the new pointer has been set (e.g., when renaming a file, do not remove the old name for an *i*-node until after the new name has been written).

At worst, poorly timed failures cause memory leaks which can be garbage-collected. Since the memory-leaked objects are either updates that are not yet reflected in the file system or removed objects that are not yet deallocated, the remaining file system is still consistent, and its correctness unaffected. Unlike *fsck*, the garbage collection can be performed as needed for reclaiming storage, since it is not required to correct inconsistent file system states. Also, this type of garbage collection process is reported to be 15x faster than *fsck,* as reported in Ganger and Patt [1994].

Although *Conquest* can alternatively implement transactional semantics, such as journaling file systems, its soft updates and pointer commits are significantly more lightweight, as observed in Section 7 and in the Solaris file system [Peacock et al. 1998], while meeting consistency and correctness requirements.

6.2 Memory Depletion

Memory depletion occurs when the allocation for persistent memory storage fails, which is equivalent to disk depletion in a conventional file system. Memory depletion can cause programs or operating systems to fail, and graceful recoveries are not always possible. In conventional systems, memory depletion is commonly handled by the virtual memory subsystem; idle processes are temporarily swapped out to disk to free up memory. Disk depletion is commonly handled by reserving extra disk storage so that the processes of reclaiming or rearranging disk storage can still create temporary files.

Under *Conquest*, the design space for handling memory depletion ranges from sophisticated data migration (e.g., LRU management) to simply complaining to the user (as in PDAs). Data migration is appealing in the sense that memory can provide the illusion of storage bounded only by the disk size. However, from the *Conquest* perspective, memory is abundant; migrating individual memory blocks introduces high complexity and overhead, as demonstrated in our performance (Section 7).

A coarser-grained approach would allow *Conquest* to adjust the large-file threshold dynamically so that smaller files migrate to disk when the memory is nearly depleted. Although this approach allows *Conquest* to operate in a more memory-constrained environment (e.g., laptops), a dynamic threshold can complicate the system in two ways. First, large-file-only disk storage would have to handle files that are smaller than expected and associated performance degradation. Bulk migration of data that is based on file size might capture neither spatial nor temporal localities. One remedy would be to design a new disk layout so that different disk zones, storing files that are accessed or created at similar time frames, improve temporal locality. However, this design would reintroduce disk-related complexity to the memory data path. Second, changing the large-file threshold would lead to a sudden migration of files. While the intended effect of migration is to free up memory space, it might need to be scheduled offline and designed as an infrequent or incremental event to avoid visible performance degradation by end-users.

Currently, *Conquest* uses the simplest strategy of just reporting the depletion of memory. This approach creates a firm boundary to prevent the disk code from handling unreasonably small files. *Conquest*'s design also handles the more familiar disk-depletion case, where freeing up storage requires an end user to archive infrequently used data content (usually at the granularity of directories) to removable media.

With an abundance of memory, memory depletion can be handled in a similar way, except that the same archiving operation has different semantics under *Conquest*. After archiving a sufficiently large directory (that contains small-file data content greater than the large-file threshold), the newly created archived file would be automatically transferred from memory to disk through *Conquest*'s media usage strategy, thereby freeing up memory storage (of course, *Conquest* would need to reserve enough memory to handle the scenario of memory depletion). As future work, we could automate this migration process at the user level, with the management at the coarse granularity of directories, as opposed to memory blocks. Also, an attempt to reference a migrated directory could automatically cause the entire tree to be restored.

6.3 System-Wide Data Migration

Another deployment consideration is that of migrating *Conquest* across machines. System-wide data migration is an infrequent event which happens mostly during system upgrades and system failure recovery. For system upgrades, the storage content under *Conquest* can be recursively copied through either the network or direct connections, which is not different from disk-based file systems. Changes in storage interface standards, rapid growth in memory and storage capacity, and warrantee issues often discourage end-users from reusing the old hardware. Should a user wish to reuse old hardware, a trivial utility program can be used to dump *Conquest's* memory region to disk and restore it on the new machine. *Conquest*'s dependency on physical addressing is not dissimilar to disks' physical addressing, which can be mapped.

If the system-wide data migration is triggered by system failures, we have two options. One is to physically move the on-board battery-backed memory and hardware drive to a new machine or power source within 12 hours. The second is to use the system backup to recover the data on a working machine, so long as the persistent RAM is based on a UPS. Again, this is no different from recovery methods for hard-disk-based systems.

6.4 *Conquest* Implementation Status

Conquest is fully operational as a loadable kernel module under Linux 2.4.2, with minor modifications to the Linux memory manager to support persistence. The current implementation follows the VFS API, but *Conquest* needs to override generic file access routines at times to provide both memory and on-disk accesses. For example, inside the read routine, *Conquest* assumes that accessing memory is the common case, and disk access is forwarded through a secondary data path.

Conquest does not disable the caching of metadata under the VFS due to difficulties in removing the deep-rooted assumption of high-latency storage and pervasive use of data and metadata caching. Routines within the VFS often use caching data structures as the standard internal representation for lookups and function parameters. Tailoring a memory data path through the VFS is likely to involve redesigning the data representation, data paths, and interfaces of internal calls within the VFS. *Conquest* currently passes its metadata structure through VFS calls such as `mknod`, `unlink`, and `lookup`. However, we altered the VFS so that *Conquest* metadata, particularly the file system root, is not destroyed at `umount` times.

Conquest is POSIX-compliant and supports both memory and on-disk storage. It currently uses a 1MB static dividing line to separate small from large files, although other thresholds are also possible (Section 3.2). Large files are stored on disk in 4KB blocks so as to reuse existing paging and protection code, without alterations. An optimization would be to enlarge the block size to 64KB or 256KB for better performance. The *Conquest* 2.4.2 source code consists of ~6,000 lines of kernel code, with garbage collection not yet implemented. *Conquest* has been used on a daily basis for the development of *Conquest* itself.

7. *CONQUEST* PERFORMANCE

We compared the performance of *Conquest* (under 0KB, 64KB, 256KB, and 1MB thresholds, dividing files stored in memory and on disk) with that of both disk- and memory-based file systems: *ext*2 [Card et al. 1994], *reiserfs* [Namesys 2003], *SGI XFS* [Sweeney et al. 1996], and *ramfs* by Transmeta [Shankland 2001]. The choice of disk-based file systems (*ext2*, *reiserfs*, and *SGI XFS*) is largely based on their common use for various performance comparisons. Since both *Conquest* and *ramfs* are under VFS API and various OS legacy constraints, the use of *ramfs* was aimed at approximating the practical achievable bound for *Conquest* performance. Note that *ramfs* uses the page cache and *v*-nodes of the VFS to store file system content and metadata directly, and provides no means of achieving data persistence after a system reboot.

We did not compare Conquest with flash-based file systems, since flash-specific management and the performance differences between flash and DRAM are orthogonal to the design principles of *Conquest*. Also, most flash-based file systems are designed for embedded devices rather than general desktop uses, and the performance of these systems is bounded by the *ramfs* case. As a reference point, the performance of JFFS2, a flash-based file system, is comparable to that of XFS and ReiserFS [Edel et al. 2004].

To conserve space, we omit the numbers for comparing *Conquest* with RAM drives using disk-based file systems. The VFS caching generally matches the *ext2* cached performance for reads, but the write performance is halved while running on RAM devices because of the copying through an additional layer of cache.

Table I describes the experimental platform. Since it has 2GB of physical RAM, all disk-based file systems use caching extensively, and the performance numbers presented reflect how well the various file systems can exploit memory hardware. In the experiments, all file systems have the same amount of memory available as *Conquest*, thus, we are comparing at constant cost.

Table II lists various file system settings. The file systems *reiserfs* and *SGI XFS* were not created and mounted with default settings because their default settings assume memory as a scarce resource. Section 7.1.1 will detail these nondefault options.

Although the most widely used benchmark in the file system literature is the Andrew file system benchmark [Howard et al. 1988], this benchmark no longer stresses modern file systems because its dataset is too small. The chosen benchmarks for this study are the Sprite LFS microbenchmarks [Rosenblum and Ousterhout 1991] with slight modifications (Section 7.1.2), the PostMark macrobenchmark[1] [Katcher 1997], and a revised version of PostMark that will

[1]As downloaded, Postmark v1.5 reported times only to a 1 second resolution. The benchmark was altered to report timing data at the resolution of the system clock.

Table I. Experimental Platform

	Experimental Platform
Manufacturer model	Dell PowerEdge 4400
Processor	1 GHz 32-bit Xeon Pentium
Processor bus	133 MHz
Memory	4×512 MB, Micron MT18LSDT6472G, SYNCH, 133 MHz, CL3, ECC
L2 cache	256KB Advanced
Disk	73.4GB, 10,000 RPM, Seagate ST173404LC
Disk partition for testing	6.1GB partition starting at cylinder 7197 (8783 cylinders total)
I/O adaptor	Adaptec AIC-7899 Ultra 160/m SCSI host Adaptor, BIOS v25306
UPS	APC Smart-UPS 700
OS	Linux 2.4.2

Table II. File System Settings

	File System Settings
Conquest	creation: default; mount: default
ext2fs (0.5b)	creation: default; mount: default
tramsmeta ramfs	creation: default; mount: default
reiserfs (3.6.25)	creation: default; mount: −o notail
SGI XFS (1.0)	creation: −l size=32768b
	mount: −o logbufs=8, logbsize32768

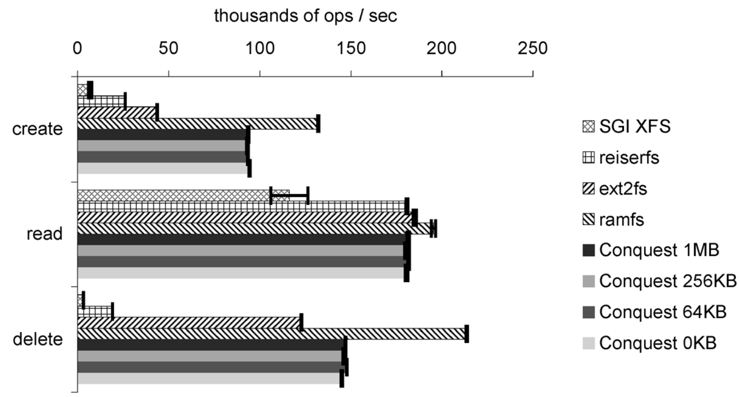
be described in Section 7.3. All results are presented at a 90% confidence level. The details of individual benchmark experiments will be discussed in corresponding subsections.

7.1 Sprite LFS Microbenchmarks

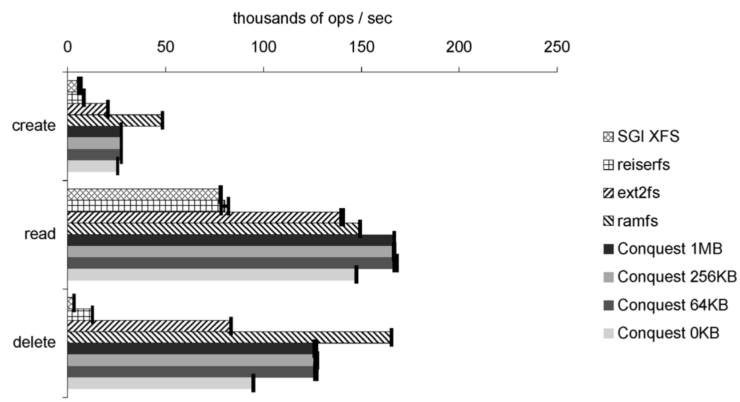
The Sprite LFS microbenchmarks measure the latency and throughput of various file operations. The benchmark consists of two separate suites for small and large files.

7.1.1 *Small-File Benchmark.* The small-file benchmark measures the latency of file operations. Each run consists of three separate phases—creating, reading, and unlinking—operating on 10,000 small files (see Figure 6). We tested three file sizes—0B, 1B, and 1KB. The 0B experiment compares the metadata performance of various file systems, since it does not exercise the code path for shipping data; the 1B experiment compares the overhead costs of data paths for various file systems to access a single byte; and the 1KB experiment compares the combined performance (both metadata and data path) of various operations on average-sized small files. For each file system, the performance numbers were collected over six runs, but averaged over only the last five to avoid warm-up effects.

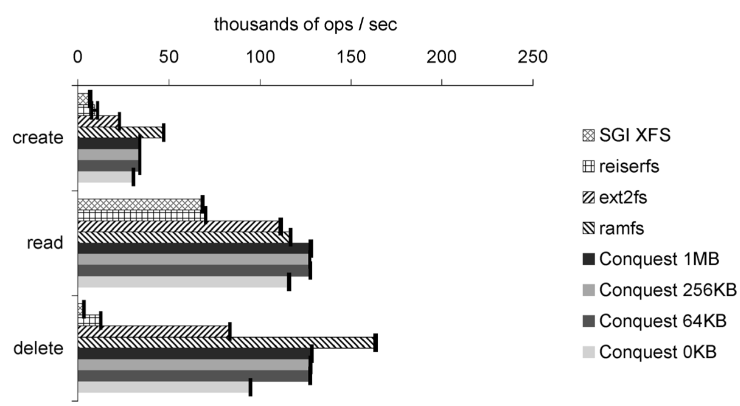
Conquest Compared to ramfs. From the 0B experiment (Figure 6a), we can see that *Conquest*'s metadata paths incur a 29% overhead in file creation, 8% in reading, and 31% in deletion. This discrepancy is caused by *Conquest*'s need to maintain its own metadata hashing data structures to support persistence, which is not provided by *ramfs*. Also, *Conquest* has not removed or disabled VFS caching for metadata, for the reasons mentioned in Section 6.4. Therefore,



(a) 10,000 0B files.



(b) 10,000 1B files.



(c) 10,000 1KB Files.

the VFS needs to go through an extra level of indirection to access *Conquest* metadata at times, while *ramfs* can avoid this overhead because its metadata is stored in the VFS cache directly.

Nevertheless, from the 1B experiment (Figure 6b), we can see that as soon as the data path is exercised, the *Conquest* memory data path (with thresholds greater than 0KB) starts to show an 11% faster read transaction rate than *ramfs*. Even though *ramfs* was considered to be a practical bound for the memory performance of file systems, *Conquest* is able to improve the read performance because the critical path to memory data contains no generic disk-related code, such as readahead or checking for cache status.

Also, given that *Conquest*'s metadata handling is slower than that of *ramfs*, the benefit of *Conquest*'s memory data path is actually greater than 11%. As we move from 0B to 1B files, *Conquest* has comparatively better read performance than *ramfs*. With a *Conquest* threshold of 0B, no small files are stored in memory, so the 1B and 1KB files exercise *Conquest*'s disk data path (Figures 6b and 6c), resulting in a noticeable performance hit, as described next.

Conquest Compared to Disk-Based File Systems. In the 0B experiment (Figure 6a), *Conquest* demonstrates 53% and 16% speed improvements over *ext2* for creation and deletion, respectively, which is mostly attributable to not needing to commit metadata synchronously to disk. For reads, *Conquest* and cached *ext2* have similar performance because both file systems have their own metadata management in addition to the VFS metadata caching.

For the 1B and 1KB experiments (Figures 6b and 6c), the *Conquest* memory data path with nonzero thresholds demonstrates 15% faster read performance than cached *ext2*, which uses the same generic disk access routines provided by VFS as *ramfs*.

For the 0KB threshold, *Conquest* uses the disk data path for all files. Leaving aside the duplicate efforts of managing metadata by *Conquest* and VFS, the in-memory metadata storage causes *Conquest* to be only marginally faster than *ext2* because metadata is heavily cached. Thus, we can conclude that the performance benefit of *Conquest* with nonzero thresholds comes mostly from its streamlined memory data path.

Journaling File Systems. The performance of SGI XFS and *reiserfs* is slower than *ext2* because of journaling overheads and their memory behaviors. The system *reiserfs* achieved even poorer performance with its default settings. Interestingly, *reiserfs* performs better with the *notail* option, which disables certain disk optimizations for small files.

SGI XFS's original default settings also produced poorer performance, since journaling consumes the log buffer quite rapidly. With a larger buffer size for logging, SGI XFS's performance improved. The numbers for both *reiserfs* and SGI XFS suggest that the overhead of journaling is very high.

←

Fig. 6. Transaction rate for the different phases of the Sprite LFS small-file benchmark, run on SGI XFS, *reiserfs*, *ext2*, *ramfs,* and *Conquest* with different large-file thresholds. Each run of the benchmark creates, reads, and unlinks 10,000 small files in separate phases. Each data point is averaged over five runs. In this and subsequent figures, the 90% confidence bars are nearly invisible due to the narrow confidence intervals.

Conquest with Different File-Size Thresholds. As long as the tested file size is smaller than or equal to a threshold, the performance reflects the memory data path of *Conquest*. As long as the tested file size is greater than a threshold, the performance reflects the disk data path of *Conquest*. The next subsection will further examine the effects of crossing various threshold boundaries.

7.1.2 *Modified Large-File Benchmark.* Each run of the original large-file benchmark writes a large file sequentially, reads from it sequentially, and then writes a new large file randomly, reads it randomly, and finally reads it sequentially. Data is `fsync`ed to disk at the end of each write phase. The final read phase was designed to measure the sequential read performance after randomly appended writes in a log-structured file system [Rosenblum and Ousterhout 1991].

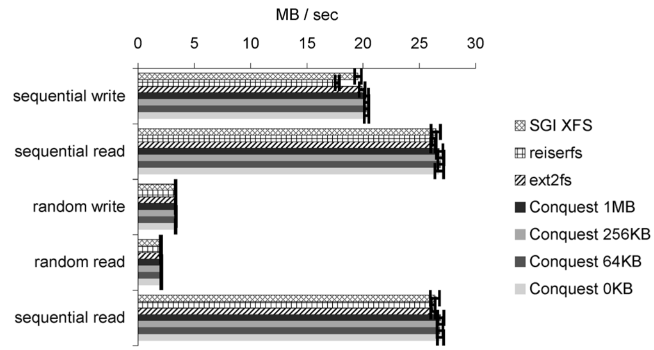
The benchmark was intended to measure disk performance. When directly applied to measuring the memory performance of file systems, the benchmark revealed a number of anomalies due to effects of memory and L2 caching. Therefore, the benchmark was modified, for the purpose of measuring *Conquest*, so that each phase of the benchmark operates on a set of equally sized files before executing the next phase. In addition, all random accesses are block-aligned to reflect common application usage patterns. Detailed reasons for these modifications and an explanation of the interactions between memory and L2 caching are explained in Wang et al. [2003].

To test the effect of a large-file threshold, we conducted two sets of experiments. For the first set, each phase of the benchmark operated on 41 large files with size equal to a given threshold, stored in memory. Results were averaged over the numbers collected from the last 40 files to avoid warm-up effects. To reset the memory states, the machine was rebooted when switching file systems.

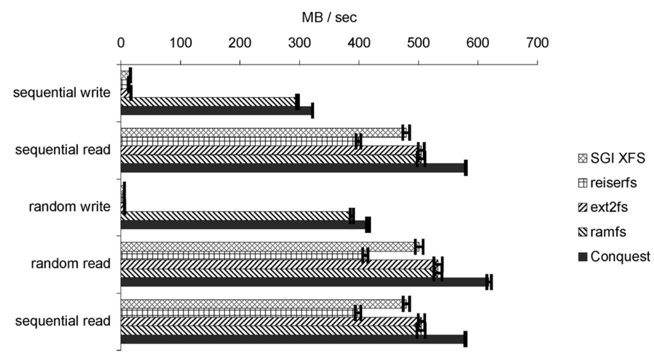
For each experiment, the file size was also increased by one block to see the performance difference once the files were switched from being stored in memory to stored on disk. To measure the performance of large on-disk files, each phase of the benchmark was also performed on 41 100MB files, with the first run discarded to avoid warm-up effects.

The 100MB Large-File Benchmark. The 100MB large-file benchmark measures the throughput of *Conquest* on-disk files (Figure 7a). This experiment only compared *Conquest* against disk-based file systems because the total size exercised by the benchmark exceeds the capacity of *ramfs*. All file systems demonstrate similar performance, showing that the additional memory data path of *Conquest* does not add noticeable overhead when accessing the disk.

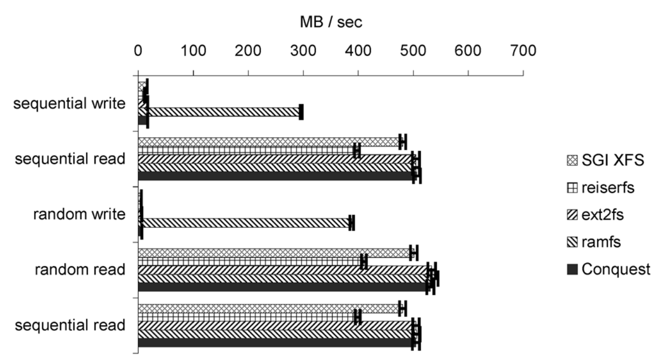
The 1MB Large-File Bbenchmark. The 1MB large-file benchmark measures the throughput of *Conquest*'s memory files (Figure 7b). Compared to *ramfs*, *Conquest* achieves a 7% higher bandwidth in both random and sequential writes and 14% higher bandwidth in random and sequential reads. The *ramfs* comparison also demonstrates the best achievable bounds for disk-based file systems. In other words, all requests are served from memory; all updates are either delayed indefinitely or committed to memory; and disk caching is bypassed to avoid extra copying and memory consumption. Compared to disk-based file systems, *Conquest* demonstrates speed improvements of 19× in sequential writes



(a) modified Sprite LFS large-file benchmark for 40 100MB files (stored on disk under Conquest)



(b) modified Sprite LFS large-file benchmark for 40 1MB files. The Conquest threshold is 1MB, and these files are stored in memory



(c) modified Sprite LFS large-file benchmark for 40 1.01MB files. The Conquest large-file threshold is 1MB, and these files are stored on disk

Fig. 7. Bandwidth for the different phases of the modified Sprite LFS large-file benchmark, run over SGI XFS, *reiserfs*, *ext2*, *ramfs*, and *Conquest* with different large-file thresholds. These experiments compare the large-file performance of memory and on-disk files under *Conquest*.

1.2× in sequential reads, 67× in random writes, and 1.2× in random reads over *ext2*. SGI XFS and *reiserfs* perform either comparably to or slower than *ext2*.

These numbers lead to several interesting observations. First, once the disk content is cached, the read performance differs little between existing disk-based file systems and *ramfs*. Since the read performance for 1MB files is dominated by transferring bytes between the VFS data cache and user-level buffers, *Conquest*'s faster read performance is mostly attributable to bypassing the VFS I/O buffer management, as opposed to having in-memory metadata management.

Second, random memory writes and reads are faster than corresponding sequential accesses. The cause is cache hits: For 1MB memory accesses with a 256KB L2 cache size, random accesses could have up to a 25% chance of reusing the L2 cache content, disregarding the additional uses of L2 caching by the operating system and microbenchmark itself. However, the sequential accesses are guaranteed to miss in such a small cache [Wang et al. 2003].

Third, the performance difference between random and sequential writes is larger than the performance difference for corresponding reads. The cause can be traced to the management semantics of L2 caching. Wang et al. [2003] explains the memory behavior of the Sprite LFS large-file benchmark, as well as subtleties in these benchmark numbers.

The 1.01MB Large-File Benchmark. For a 1MB threshold, the 1.01MB large-file benchmark shows the performance effects of switching a file from storage in memory to storage on-disk under *Conquest* (Figure 7b). For large files, since the overheads of both metadata and fragmentation management are negligible compared to the time to transfer bytes, *Conquest* benefits mostly in data path simplification, only matching the performance of cached *ext2* on this test. However, *Conquest* does show that the use of disjoint data paths for memory and disk imposes little or no extra overhead for disk accesses.

Other Large-File Benchmarks and Conquest Thresholds. Other benchmark numbers with different large file sizes and thresholds show similar trends and are omitted to save space. For a 256KB threshold, *Conquest* shows a 6% to 13% faster performance over *ramfs* for various reads and writes. As the threshold decreases to 64KB, *Conquest* offers only 3% to 9% performance benefits over *ramfs*, since the data path advantage of *Conquest* over *ramfs* decreases as the cost of metadata manipulation starts to dominate. Also, as soon as the file size exceeds the threshold, the performance of *Conquest* matches that of a disk-based system with caching, regardless of the threshold setting. Thus, a user can incrementally add more RAM under *Conquest*'s control, increasing the file size threshold and taking advantage of *Conquest*'s speed as the price of memory decreases.

7.2 PostMark Macrobenchmark

The PostMark benchmark was designed to model the workload seen by Internet service providers [Katcher 1997], simulating a combination of electronic mail, Usenet, and web-based commerce transactions. PostMark creates a set of files whose sizes are chosen at random and uniformly distributed over a file size range. These files are then subjected to transactions consisting of a pairing of file creation or deletion with file read or append. Each pair of transactions is chosen randomly, with a bias controlled by parameter settings. A deletion operation removes a file from the active set. A read operation reads a randomly selected file in its entirety. An append operation opens a random file, seeks to the end of this file, and writes a random amount of data, without exceeding the maximum file size.

Early PostMark experiments used 10,000 files with a size range of 512 bytes to 16KB. One run of this configuration performs 200,000 transactions with an equal probability of creates and deletes, and a 4× higher probability of performing reads than appends. The transaction block size is 512 bytes. However, since this workload is far smaller than the workload observed at any ISP today, we conducted experiments varying the number of files from 5,000 to 25,000 to see the effects of scaling.

Since all files within the size range will be stored in memory under *Conquest*, this benchmark did not exercise *Conquest*'s disk aspect. Also, since this configuration specifies an average file set of only 250MB, which fits in 2GB of memory, this benchmark compared the performance of *Conquest* against those of existing cache and I/O buffering mechanisms under a realistic mix of file operations. Since existing experimental settings of *Conquest* use thresholds either above or below the file size range of the PostMark workload, we also inserted a threshold setting of 8KB to see the effect of exercising both the memory and disk components of *Conquest*. However, this 8KB threshold is by no means a practical setting, since the disk component is designed to store much larger files so as to avoid the complexity and overhead of handling small files.

The measurement machine was rebooted when switching file systems. A shell first repeated six runs of the modified PostMark with the 5,000-file configuration, with the numbers collected from the first run dropped to reduce warm-up effects. Then, the script removed these 5,000 files and proceeded to the configuration of 10,000 files, and so on.

Figure 8 compares *Conquest*'s transaction rates with those of other file systems as the number of files varied from 5,000 to 25,000. For thresholds above the file size range of this workload, *Conquest* is marginally faster than *ramfs* because the data path dominates performance characteristics. For the same thresholds, *Conquest* achieves a 1.3× to 3.5× performance speedup compared to *ext2*. The systems SGI XFS and *reiserfs* perform much slower than *ext2* due to journaling overheads.

To see the performance contribution of storing metadata in memory, *Conquest* with a 0KB threshold matches the performance of *ext2* for 5,000 files due to the predominant use of the disk data path. However, as the number of files increases to 25,000, the in-memory *Conquest* metadata structure allows *Conquest* to outperform cached *ext2* by as much as 79%. Although storing metadata in memory already produces a significant performance improvement over disk caching, the data path benefit of *Conquest* actually pushes the performance boundary 3× as far, which even outperforms *ramfs*.

The 8KB threshold setting of *Conquest* only shows comparable performance to cache *ext2* for 5,000 files. However, as the number of files increases to 25,000,

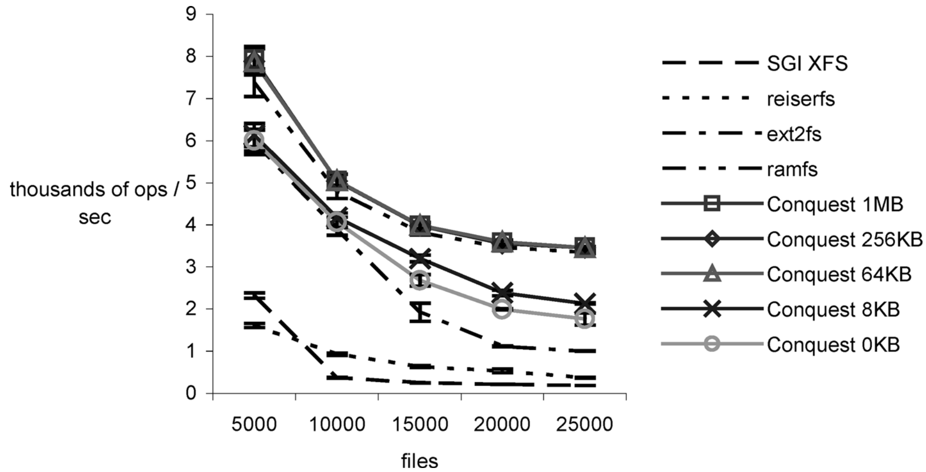


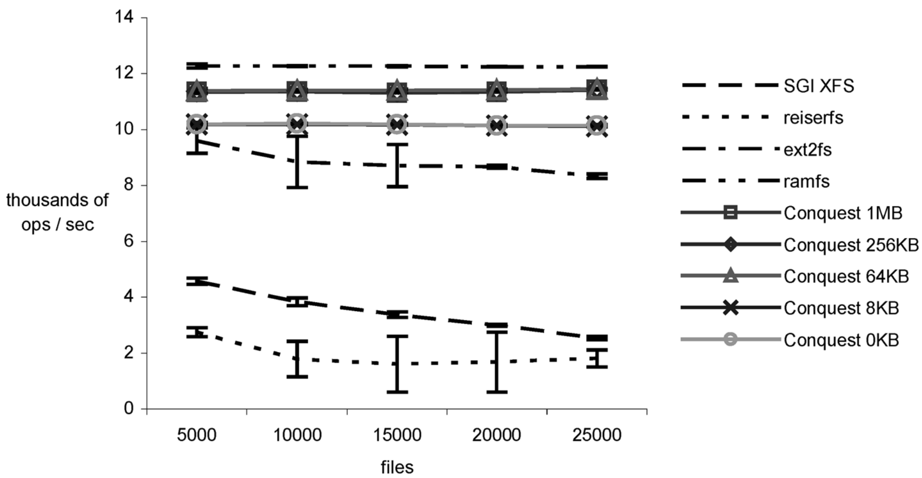
Fig. 8. PostMark transaction rate for SGI XFS, *reiserfs*, *ext2, ramfs*, and *Conquest* with different large-file thresholds varying from 5,000 to 25,000 files. The results are averaged over five runs.

this threshold still allows *Conquest* achieve a 2.1× speedup compared to *ext2*. Although the 8KB threshold allows *Conquest* to store 50% of the files on disk, *Conquest* does not achieve a performance at the midway between the 64KB+ thresholds and the 0KB threshold, since these 50% on-disk files are larger and constitute 75% of the bytes.

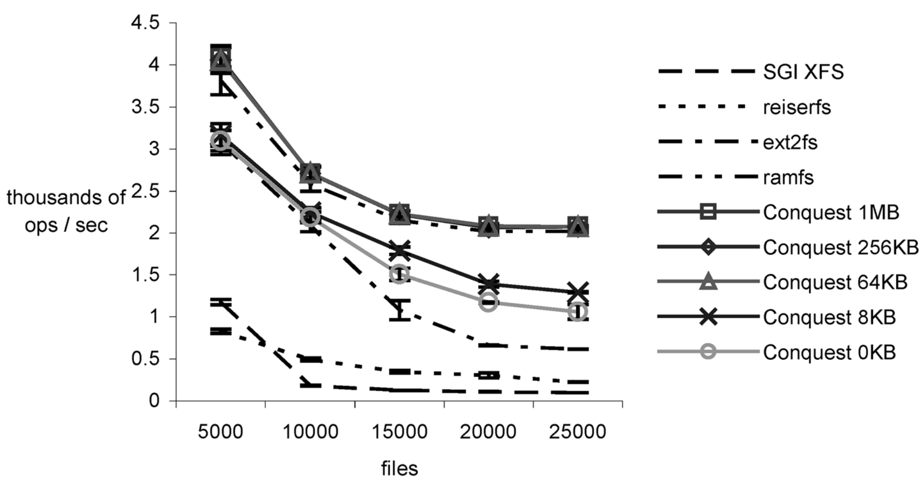
Conquest's ability to outperform disk caching in this memory-resident workload leads to two conclusions. First, while a memory-only workload may seem to be an unfair comparison for disk-based file systems, the trend of increasing memory size actually makes it a common case. Second, since file system designers do not emphasize the memory performance aspect of a file system as much as disk performance, disk caching designs fail to take full advantage of memory bandwidth. The overhead due to caching-related management can lead to an overall performance that is several times worse than *Conquest*.

To examine the effects of individual operations in the benchmark, Figure 9 presents the performance for only file creation operations. Similar lessons can be drawn for file deletion operations; therefore, its graph is omitted. Figure 9 shows the performance of the various file systems, both when each operation is conducted without interference, and when the tested operation is mixed with others.

As we can see, when measured without other types of operations, the file creation rate shows little degradation for all systems as the number of files increases (Figure 9a). When mixed with other types of file transactions (Figure 9b), the file creation rate degrades drastically for all file systems tested. Since the performance trend for mixed file creation (Figure 9b) is similar to the trend for throughput (Figure 10) and transaction rate (Figure 8), we can again see that the effect of *Conquest*'s streamlined data paths (or read and write operations) has contributed more to outperforming disk caching than has in-memory metadata handling. Even though PostMark is known to be metadata-intensive, the read and write throughput still dominates performance behavior.



(a) PostMark file creation rate, without the interference of other operations



(b) PostMark file creation rate, mixed with other operations

Fig. 9. PostMark file creation performance for SGI XFS, *reiserfs*, *ext2*, *ramfs*, and *Conquest* with different large-file thresholds, varying from 5,000 to 25,000 files. The results are averaged over five runs.

For the pure file creation numbers (Figure 9a), *Conquest* with 64KB, 256KB, and 1MB thresholds is about 7% slower than *ramfs* because *Conquest* has not disabled VFS metadata caching for the reasons mentioned in Section 6.4. With 8KB and 0KB thresholds where it uses the disk data path, *Conquest* is about 17% slower than *ramfs*, but still 6% to 21% faster than cached *ext2*. When mixed with other types of file transactions, various systems start creating files

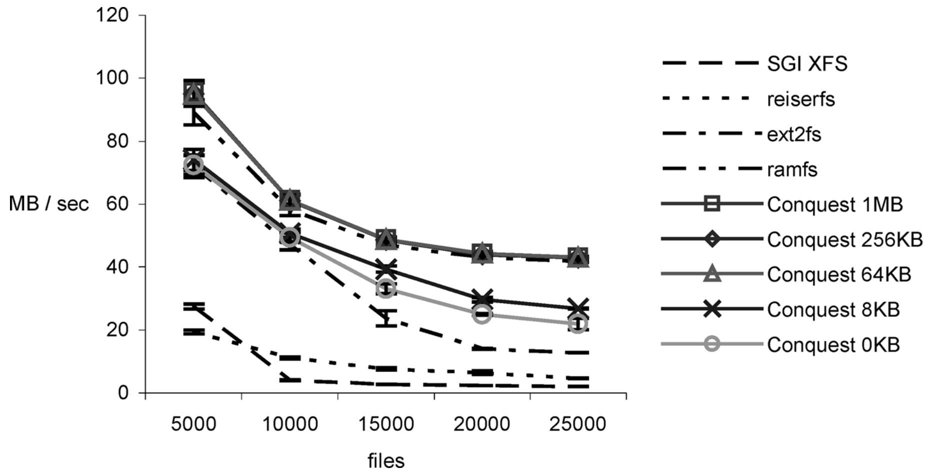


Fig. 10. PostMark throughput for SGI XFS, *reiserfs*, *ext2*, *ramfs*, and *Conquest* with different large-file thresholds, varying from 5,000 to 25,000 files. The results are averaged over five runs.

at rates according to the trend of transaction rates, since the PostMark numbers are largely dominated by the performance of reads and writes (Figure 9b).

It is interesting to see that SGI XFS has a faster file creation rate than *reiserfs* without mixed traffic, but a slower rate with mixed traffic (except in the 5000-file test). This result demonstrates that optimizing individual operations in isolation does not necessarily yield better performance when they are mixed, especially when other operations dominate the performance characteristics.

Confidence intervals tend to be larger with fewer files (Figure 9b) due to lower accuracy when averaging over shorter elapsed times. Also, the confidence intervals are large at times for disk-based file systems. For SGI XFS, file creations may interact with the journaling mechanisms that use the disk heavily (Figure 9a). For *ext2*, file deletion interacts with recent asynchronously buffered writes.

In terms of the throughput (Figure 10), although the overall performance trend matches that of the transaction rate (Figure 8), Figure 10 shows a significant throughput loss when compared to microbenchmark numbers (Figure 7b). This discrepancy could be caused by a combination of factors, including the mixture of creation and deletion operations, a larger number of files and total file sizes, and the interaction of the benchmark footprint with L2 caching management. Since this performance loss is not specific to *Conquest*, a detailed investigation will be conducted in future work.

7.3 Modified Postmark Benchmark

We also modified the PostMark benchmark to exercise both the memory and disk components of *Conquest*. The modified PostMark benchmark generates a percentage of files in a large file category, with file sizes uniformly distributed between 2MB and 5MB. The mean of this size range is twice the mean video request size observed in a proxy traffic workload study [Mahanti et al. 2000], giving a more conservative picture of *Conquest*'s performance when both memory and disk components are exercised. This setting is reflective of the growing number of multimedia files and their emerging file size distribution [Evans and Kuenning 2002]. The setting also anticipates continuance of the trend for large files to grow over time [Douceur and Bolosky 1999; Vogels 1999; Roselli et al. 2000; Evans and Kuenning 2002]. The remaining files were uniformly distributed between 512 bytes and 16KB. The total number of files was fixed at 10,000 and the percentage of large files varied from 0.0 to 10.0 (0GB to 3.5GB). Since the file set exceeds the storage capacity of *ramfs*, *ramfs* could not be included in the results.

Figure 11 compares the transaction rates of SGI XFS, *reiserfs*, *ext2*, and *Conquest* with various large-file thresholds. Figure 11a shows how the measured transaction rates of the four file systems vary as the percentage of large files increases. Because the scale of this graph obscures important details at the righthand-side, Figure 11b shows the performance ratio of *Conquest* with a 1MB threshold compared to various file systems and *Conquest* threshold settings.

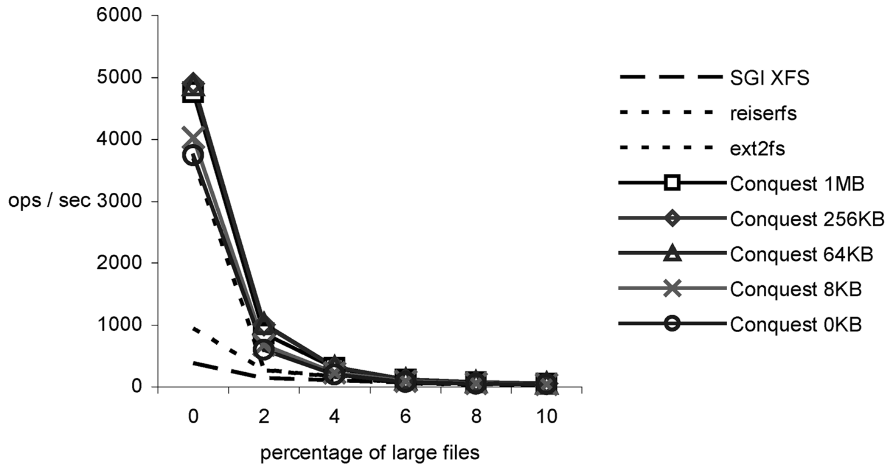
Conquest demonstrates 1.3× to 3.1× faster transfer rates than *ext2* (Figure 11b). The shape of the *Conquest* speedup curve over *ext2* reflects the rapid degradation of *ext2* performance with the injection of disk traffic. As more disk traffic is injected, we start to see a relatively steady performance ratio as the data path bandwidth effect dominates the performance effect. At a steady state, *Conquest* shows transaction rates 1.8× faster than *ext2*, 1.4× faster than SGI XFS, and 2.0× faster than *reiserfs*.

In terms of various large-file thresholds, it is interesting to note that the performance ratios of *Conquest* with different thresholds are relatively constant, suggesting that (1) the data path bandwidth effect dominates the *Conquest* performance landscape, even when there are no large files, and (2) the handling of small files does not penalize the performance of large files. At the steady state, *Conquest*'s 1MB threshold is 29% faster than the 0KB threshold, 18% faster than the 8KB threshold, and comparable to the 64KB and 256KB thresholds because the file size ranges of modified PostMark do not have files between 16KB and 2MB.

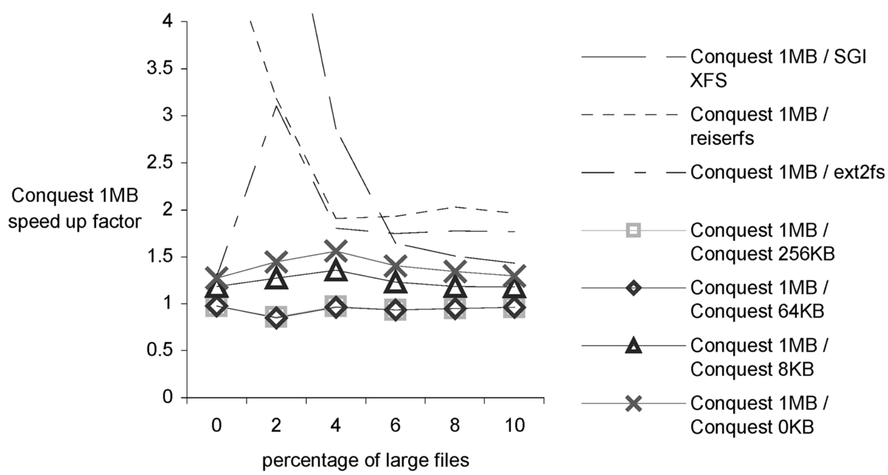
Both SGI XFS and *reiserfs* show significantly slower memory performance (lefthand-side of Figure 11a). However, as the file set exceeds the memory size, SGI XFS starts to outperform *ext2* and *reiserfs* (Figure 11b). Clearly, different file systems are optimized for different workloads.

8. RELATED WORK

Caching in volatile RAM has inspired *Conquest* to a large extent, and the relative strengths and weaknesses of caching have already been discussed in Section 2. The idea of combining memory- and disk-based storage service can be traced back to main-memory databases. Researchers then began to use persistent RAM in the file systems arena. PDA operating systems represented a major step, providing both memory and file system services in persistent RAM; however, *Conquest* assumes an abundance of persistent RAM, which is scarce on these handheld devices. The design philosophy of *Conquest* is also supported by existing systems that share similar characteristics.



(a) the full scale graph



(b) Conquest speedup curves for the full graph

Fig. 11. Modified PostMark transaction rate for SGI XFS, *reiserfs*, *ext2*, and *Conquest* with different large-file thresholds, with varying percentages of large (on-disk *Conquest*) files ranging from 0.0 to 10.0 percent.

Main-Memory Databases. The database community has a long-established history of main-memory database systems (MMDBs). Early survey papers reveal key architectural implications of abundant RAM [DeWitt et al. 1984; Garcia-Molina and Salem 1992]. One file-system-related observation is that since memory is much faster than disk, each transaction is completed within a shorter time; therefore, the probability of locking contention is smaller. A larger locking granularity in an MMDB can reduce the locking overhead [Lehman and

Carey 1987] and complexity of the system. A reduced probability of waiting for locks also translates into fewer context switches and resulting cache flushes, further improving overall system performance.

Garcia-Molina and Salem [1987] also discussed several early main-memory databases. For example, IMS/VS Fast Path [Gawlick and Kinkade 1985] delivers frequently used database items from an MMDB and infrequently used ones from a disk-resident database (DRDB). The two databases are designed with separate access mechanisms. Instead of making duplicate copies of data, as in multilevel caching, IMS/VS Fast Path occasionally migrates data from one persistent medium to another on the basis of access patterns. Similar to *Conquest*, IMS/VS Fast Path relies on battery backing, frequent system backups, and uninterruptible power supplies for reliability. MARS [Eich 1987], HALO [Garcia-Molina and Salem 1987], TPK [Li and Naughton 1988], and other early MMDBs rely on data mirroring, background logging, or dual processing to achieve reliability.

Unfortunately, the techniques developed by main-memory databases are not directly applicable to operating systems. Databases are optimized for database access patterns, rather than generalized file system access patterns. Their storage system is commonly accessed through the query interface, as opposed to the VFS interface. The *Conquest* design is unique in offering a transition for delivering file system services from main memory in a practical and cost-effective way.

File-System Applications of Persistent RAM. One early use proposed for persistent RAM was to hold write buffers [Baker et al. 1992]. Since data would be buffered in persistent memory, the intervals between synchronizations to the disk could be lengthened. Although disk activity would be reduced significantly, this approach does not eliminate data duplication, migration, synchronization, and fresh loading of the buffer when the disk content is first accessed.

Write anywhere data layout (WAFL) [Hitz et al. 1994] shares a certain similarity with *Conquest*, since both WAFL and *Conquest* achieve file system consistency at all times. WAFL advocates the collocation of metadata and data on disk, so metadata writes are piggybacked with data writes to avoid disk accesses. Written data is not used until the system advances atomically to the next snapshot. Atomicity is achieved through journaling, with logs stored in persistent RAM. WAFL can recover from a system failure by replaying the logs. If the persistent RAM fails, the disk still retains a consistent snapshot, which is taken every 10 seconds. Since WAFL runs only in an NFS appliance, it is difficult to compare its performance to *Conquest*, although since WAFL does not focus on streamlining the memory data path, it is unlikely to outperform *Conquest*.

A plethora of flash-memory-based file systems has emerged to replace disks on small mobile computing devices [Wu and Zwaenepoel 1994; Kawaguichi et al. 1995; Niijima 1995; Torelli 1995; Woodhouse 2001; Gal and Toledo 2005]. The low-power requirements of these devices make flash memory an attractive choice; however, flash memory has a limited number of erase-write cycles and slow (second-range) time for storage reclamation. These characteristics cause new kinds of performance problems.

Quantum has proposed using battery-backed DRAM for logging, metadata updates, transactions, and caching for disk array, database, and multimedia servers [Quantum 2003]. They do not plan to use BB-DRAM as the primary storage device, however. Instead, they access BB-DRAM devices as if they were mechanical disks via the SCSI interface, with the same software infrastructures to handle disks. From a marketing point of view, Quantum has shortened the development process by making BB-DRAM plug-and-play, but from an engineering viewpoint, these BB-DRAM devices are far from achieving their full performance potential.

The Rio file cache [Chen et al. 1996; Ng and Chen 2001] combines UPS, volatile memory, and a modified write-back scheme to achieve the reliability of a write-through file cache and the performance of a pure write-back file cache (with no reliability-induced writes to disk). The resiliency offered by Rio complements *Conquest* well. While *Conquest* uses the main store as the final storage destination, Rio's BIOS *safe sync* mechanism provides high assurance for a system to: (1) transfer control to the `sync` routine during a crash; and (2) write the data content of memory to disk. Rio's reliability mechanisms try to avoid any dependency on underlying kernel mechanisms to minimize the effect of kernel crashes on the proper operations of Rio. For example, Rio removes dependencies on the virtual memory system by switching the processor to use physical addresses. Also, Rio removes dependencies on kernel device drivers by using the BIOS interface to disk.

The HeRMES project [Miller et al. 2001] takes advantage of a form of persistent RAM that is still under development as of this writing, namely, magnetic RAM (MRAM) [Boeve et al. 1999]. HeRMES uses MRAM primarily to store the file metadata so as to reduce a large component of existing disk traffic, and also to buffer writes so as to lengthen the time frame for committing modified data. HeRMES also assumes that persistent RAM will remain a relatively scarce resource for the foreseeable future, especially for large file systems. As our performance results show, many significant *Conquest* performance gains are not due to improvements in metadata handling, and HeRMES limits its use of persistent memory to metadata.

PDA Operating Systems. The two leading PDAs on the market are PalmOS and Windows CE devices. Both systems deliver memory and file system services via BB-DRAM, but their designs are more concerned with fitting the operating systems into memory-constrained environments than with exploiting an abundance of persistent RAM.

To use limited BB-DRAM on handhelds, both PalmPilot and Windows CE make many design simplifications. Palm has designed a commodity system with a few essential core services. For Palm OS 5 [Palm 2000], the execution environment does not provide a process abstraction; instead, a single program executes at a time. Palm OS hardwires the partitions of memory space for purposes such as low memory globals, the dynamic heap, and storage heap. The low memory is used for various OS subsystems. The dynamic heap stores global data for the operating system, applications, and various other purposes, and the storage heap provides persistent storage. The data storage granularity is a contiguous *chunk* of memory of 1KB to 64KB, managed by an internal database engine. For large files, an alternative API can support files of arbitrary sizes, and file-based operations are buffered. The reliability model of PalmPilot mainly relies on battery backing and data synchronization (i.e., backup) to a desktop machine.

Unlike Palm OS, Windows CE tries to miniaturize the full operating system environment to the scale of a PDA, with full support for multiple processes and threads for execution. Windows CE 3.0 [Microsoft 2003] uses a variety of techniques to simplify memory management and reduce memory overhead. File operations are provided, and memory-mapping mechanisms are available to avoid copying. Windows CE can mount external file systems, but they are limited to the size of the memory on the handheld device, and the maximum file size is limited to 32MB.

Neither PDA design is suitable for general deployment on desktop computers. The PalmPilot lacks a full-featured execution model, and the number of efficient methods for accessing large data objects is limited. Windows CE is not designed for desktop-scale deployment, and many management functions are simplified by requiring end users to specify them explicitly (e.g. the boundary between the program memory and permanent storage).

IBM AS/400. IBM AS/400 servers provide the appearance of storing all files in memory. This uniform view of storage access is accomplished by the extensive use of virtual memory. However, underneath the hood of AS/400, the conventional role of memory as the cache for disk content still applies, and disks are still the persistent storage medium for files [IBM 2003].

Slice. *Conquest*'s approach of separating data paths based on file sizes and metadata can also be found in distributed storage systems. The Slice file service [Anderson et al. 2000] is an example. Each client's request stream is partitioned into three functional request classes: (1) high-volume I/O to large files; (2) I/O on small files; and (3) operations on the name space or file attributes. Based on the request type and arguments, a front-end μproxy switches to redirect requests to a selected server that is responsible for handling a given class of requests. Directory servers provide naming services through distributed hashing and load balancing. Small-file servers are specialized for fragmentation management so that they can provide both efficient storage and high bandwidth. Bulk I/O operations route directly to an array of storage nodes which provides block-level access to raw storage objects.

9. FUTURE WORK

Conquest is now operational, but we can further improve its performance and usability in a number of ways. One previously mentioned area is that of finding a better disk layout for large data blocks (Section 4.2.3).

High-speed in-memory storage also opens up additional possibilities for operating systems. *Conquest* provides a simple and efficient way for kernel-level code to access a general storage service, which has conventionally either been avoided entirely or achieved through the use of more limited buffering mechanisms. One major area of application for this capability would be system monitoring and lightweight logging, but there are numerous other possibilities, as well.

In terms of our research so far, we have aggressively removed many disk-related complexities from the in-memory critical path, without questioning exactly how much each disk optimization adversely affects file system performance. One area of research is the question of how to break down these performance costs so that designers can improve the memory performance of disk-based file systems.

Memory under *Conquest* is a shared resource among execution, storage, and buffering for disk access. Finding the "sweet spot" for optimal system performance will require both modeling and empirical investigation. In addition, after reducing the roles of disk storage, *Conquest* exhibits different system-wide performance characteristics, and the implications can be subtle. For example, the conventional wisdom of mixing CPU- and IO-bound jobs may no longer be a suitable scheduling policy. We are currently experimenting with a wider variation of workloads to investigate a larger range of *Conquest* behavior.

10. CONCLUSIONS

This article presents *Conquest*, a fully operational file system that integrates persistent RAM with disk storage to provide significantly improved performance as compared to other approaches, such as RAM disks or enlarged buffer caches. *Conquest* demonstrates a $1.4\times$ to $2.0\times$ speedup over popular disk-based file systems for both in-memory workloads and workloads that must exercise the disk.

The benefits of *Conquest* arise from rethinking basic file system design assumptions. *Conquest* explores the implications of a memory-rich environment, challenges the commonly perceived performance bound of LRU disk caching, and questions layer-based optimizations by proposing separate cut-through data paths. By revisiting individual decision points in a new context and designing from the ground up, we evolved *Conquest* into two simpler data paths that surpass the performance accomplished by using many layers of legacy optimizations.

The experience of designing and implementing *Conquest* offers several major lessons:

—*The handling of disk characteristics permeates file system design, even at levels above the device layer.* For example, the default VFS routines contain readahead and buffer cache mechanisms that add high and unnecessary overheads to low-latency main store. Because of the need to bypass these mechanisms, building *Conquest* was much more difficult than we initially expected. For example, certain internal storage routines anticipate the data structures associated with disk handling. Reusing these routines either involves constructing memory-specific access routines from scratch or finding ways to invoke them with memory-based data structures.

—*File systems that are optimized for disk are not suitable for an environment where memory is abundant.* For example, *ext2*, *reiserfs*, and SGI XFS do not exploit the speed of RAM as well as had been anticipated. Disk-related optimizations impose high overheads on in-memory accesses.

—*Matching the physical characteristics of media to storage objects provides opportunities for faster performance and considerable simplification for each medium-specific data path. Conquest* applies this principle of specialization: Leaving only the data content of large files on disk leads to simpler and cleaner management for both memory and disk storage. This observation may seem obvious, but good results are not achieved automatically. For example, should the L2 cache footprint of two specialized data paths exceed the size of a single generic data path, the resulting performance can go in either direction, depending on the size of the physical cache.

—*Access to cached data in traditional file systems incurs performance costs due to commingled disk-related code.* Removing disk-related complexity for in-memory storage under *Conquest* therefore yields unexpected benefits, even for cache accesses. In particular, one surprising result was *Conquest*'s ability to outperform *ramfs* by 7% to 14% in bandwidth under microbenchmarks, despite the fact that the storage data paths in *ramfs* are already heavily optimized.

—*It is much more difficult to use RAM to improve disk performance than might appear at first.* Simple approaches such as increasing the buffer cache size or installing simple RAM-disk drivers do not generate a full featured, high-performance solution.

Conquest demonstrates how this process of rethinking underlying assumptions can lead to significant performance benefits and architectural simplifications. This experience suggests that radical changes in hardware, applications, and user expectations of the past decade should also lead us to reflect on future file system design and other aspects of operating system design.

ACKNOWLEDGMENTS

We would like to thank Michael Gorlick and Richard Guy for reviewing an early presentation of the *Conquest* performance results and offering useful insights. In addition, we want to thank Mark Yarvis, Scott Michel, and Janice Wheeler for commenting on earlier drafts of this article.

REFERENCES

APC. 2005. SMART-UPS. http://www.apc.com.

ANDERSON, D., CHASE, J., AND VAHDAT, A. 2000. Interposed request routing for scalable network storage. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*. San Diego, CA.

BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. 1991. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*. Pacific Grove, CA.

BAKER, M., ASAMI, S., DEPRIT, E., OUSTERHOUT, J., AND SELTZER, M. 1992. Non-volatile memory for fast, reliable file systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. Boston, MA.

BITMICRO. 2005. High-End solid state disk. http://www.bitmicro.com/products_edisk_25_scsin.php.

BOEVE, H., BRUYNSERAEDE, C., DAS, J., DESSEIN, K., BORGHS, G., DE BOECK, J., SOUSA, R., MELO, L., AND FREITAS, P. 1999. Technology assessment for the implementation of magnetoresistive elements with semiconductor components in magnetic random access memory (MRAM) architectures. *IEEE Trans. Magnet. 35*, 5, 2820–2825.

BOLOSKY, W. J., FITZGERALD, R. P., AND DOUCEUR, J. R. 1997. Distributed schedule management in the Tiger video fileserver. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France.

BONWICK, J. 1994. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer Technical Conference*. Boston, MA.

BOZMAN, G. P., GHANNAD, H. H., AND WEINBERGER, E. D. 1991. A trace-driven study of CMS file references. *IBM J. Res. Dev. 35*, 5–6, 815–828.

CÁCERES, R., DOUGLIS, F., LI, K., AND MARSH, B. 1993. Operating system implications of solid-state mobile computers. *Tech. rep.* MITL-TR-56-93, Matsushita Information Technology Laboratory, United States.

CARD, R., TS'O, T., AND TWEEDIE, S. 1994. Design and implementation of the second extended filesystem. In *Proceedings of the 1st Dutch International Symposium on Linux*. ISBN 90-367-0385-9.

CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. 1996. The Rio file cache: Surviving operating system crashes. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA.

CHEN, S. AND THAPAR, M. 1997. A novel video layout strategy for near-video-on-demand servers. *Tech. rep.* HPL-97-52. Hewlett-Packard Laboratories.

DELL. 2002. Determining the availability and reliability of storage configurations. http://www1.us.dell.com/content/topics/global.aspx/power/en/ps3q02_shetty?c=us&l=en&s=corp. Google keywords: Dell, reliability, MTBF, hours.

DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M., AND WOOD, D. A. 1984. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

DOUCEUR, J. R. AND BOLOSKY, W. J. 1999. A large-scale study of file-system contents. In *Proceedings of the ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*. Atlanta, GA.

DOUGLIS, F., CÁCERES, R., KAASHOEK, F., LI, K., MARSH, B., AND TAUBER, J. A. 1994. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*. Monterey, CA.

EDEL, N. K., TUTEJA, D., MILLER, M. L., AND BRANDT, S. A. 2004. MRAMFS: A compressing file system for non-volatile RAM. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Volendam, the Netherlands.

EICH, M. H. 1987. A classification and comparison of main memory database recovery techniques. In *Proceedings of the 3rd International Conference on Data Engineering*. Los Angeles, CA.

EVANS, K. M. AND KUENNING, G. K. 2002. A study of irregularities in file-size distributions. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*. San Diego, CA.

FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. R. 1979. Extensible hashing—A fast access method for dynamic files. *ACM Trans. Datab. Syst. 4*, 3, 315–344.

GAL, E. AND TOLEDO, S. 2005. A transactional flash file system for microcontrollers. In *Proceedings of the USENIX Annual Technical Conference*. Anaheim, CA.

GANGER, G. R. AND PATT, Y. N. 1994. Metadata update performance in file systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

GANGER, G. R., MCKUSICK, M. K., SOULES, C. A. N., AND PATT, Y. N. 2000. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst. 18*, 2, 127–153.

GARCIA-MOLINA, H. AND SALEM, K. 1987. High performance transaction processing with memory resident data. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*. Pacific Grove, CA.

GARCIA-MOLINA, H. AND SALEM, K. 1992. Main memory database systems: An overview. *IEEE Trans. Know. Data Eng. 4*, 6, 509–516.

GAWLICK, D. AND KINKADE, D. 1985. Varieties of concurrency control in MIS/VS fast path. *IEEE Datab. Eng. 8*, 2, 3–10.

GIBSON, G. A. AND PATTERSON, D. A. 1993. Designing disk arrays for high data reliability. *J. Parallel. Distribut. Comput. 17*, 1–2, 4–27.

GROCHOWSKI, E. AND HALEM, R. D. 2003. Technological impact of magnetic hard disk drives on storage systems. *IBM Syst. J. 42*, 2. http://www.research.ibm.com/journal/sj/422/grochowski.html.

HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter Technical Conference*. San Francisco, CA.

HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1, 51–81.

IBM. 2003. IBM iSeries storage overview. http://www-1.ibm.com/servers/eserver/iseries/hardware/storage/overview.html.

IRLAM, G. 1993. UNIX file size survey—1993. http://www.base.com/gordoni/ufs93.html.

KATCHER, J. 1997. PostMark: A new file system benchmark. *Tech. Rep.* TR3022. Network Appliance, Inc.

KAWAGUICHI, A., NISHIOKA, S., AND MOTODA, H. 1995. A flash-memory-based file system. In *Proceedings of the USENIX Winter Technical Conference*. New Orleans, LA.

KEREKES, Z. 2005. Charting the rise of the solid state disk market. http://www.storagesearch.com/chartingtheriseofssds.html.

KLEIMAN, S. R. 1986. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer USENIX Conference*. Atlanta, GA.

LEHMAN, T. J. AND CAREY, M. J. 1987. A recovery algorithm for a high-performance memory-resident database system. In *Proceedings of the ACM SIGMOD Conference*. San Francisco, CA.

LI, K. AND NAUGHTON, J. F. 1988. Multiprocessor main memory transaction processing. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*. Austin, TX.

LIEBERT COOPERATION. 2005. Field MTBF numbers: What do they really mean? http://www.liebert.com/support/whitepapers/documents/techmtbf.asp.

MAHANTI, A., WILLIAMSON, C., AND EAGER, D. 2000. Traffic analysis of a web proxy caching hierarchy. *IEEE Netw. Magazine: Special Issue on Web Performance 14*, 3, 16–23.

MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. 1984. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2*, 3, 181–197.

MCKUSICK, M. K., KARELS, M. J., AND BOSTIC, K. 1990. A pageable memory based filesystem. In *Proceedings of the Summer USENIX Conference*. Anaheim, CA.

MCKUSICK, M. K. AND GANGER, G. R. 1991. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the USENIX Annual Technical Conference*.

MCKUSICK, M. K. 2002. Running "fsck" in the background. In *Proceedings of the BSDCon Conference*. San Francisco, CA.

MICRON. 1997. Module mean time between failures (MTBF). Tech. Note TN-04-45. http://download.micron.com/pdf/technotes/DT45.pdf.

MICROSOFT. 2003. Microsoft Windows CE 3.0: Files, databases, and persistent storage. *MSDN Online Library*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncenet/html/systemmemorymgmtwince.asp.

MILES, J. B. 2000. Thin clients. *Government Comput. News 6*, 11. http://appserv.gcn.com/state/vol6_no11/guide/893-1.html.

MILLER, E. L., BRANDT, S. A., AND LONG, D. D. E. 2001. HerMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*. Schloss Elmau, Germany.

NAMESYS. 2005. http://www.namesys.com.

NG, N. T., AYCOCK, C. M., RAJAMANI, G., AND CHEN, P. M. 1996. Comparing disk and memory's resistance to operating system crashes. In *Proceedings of the International Symposium on Software Reliability Engineering*. Hong Kong, China.

NG, N. T. AND CHEN, P. M. 2001. The design and verification of the Rio file cache. *IEEE Trans. Comput. 50*, 4, 322–337.

NIIJIMA, H. 1995. Design of a solid-state file using flash EEPROM. *IBM J. Res. Dev. 39*, 5, 531–546.

OUSTERHOUT, J. K., DA COSTA, H., HARRISON, D., KUNZE, A., KUPFER, M., AND THOMPSON, J. G. 1985. A trace driven analysis of the UNIX 4.2 BSD file systems. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*. Orcas Island, WA, 15–24.

PALM. 2004. Introduction to palm OS memory use. *Palm OS Programmer's Companion, Vol. I*. http://www.palmos.com/dev/support/docs/palmos/PalmOSCompanion/Memory.html.

PC WORLD. 2005. IRam speeds Windows XP startup. *PC World*. http://www.pcworld.com/news/article/0,aid,121105,00.asp.

PEACOCK, J. K., KAMARAJU, A. AND AGRAWAL, S. 1998. Fast consistency checking for the solaris file system. In *Proceedings of the USENIX Annual Technical Conference*. New Orleans, LA.

PETERSON, J. L. AND NORMAN, T. A. 1997. Buddy systems. *Commun. ACM 20*, 6, 421–431.

PRICE WATCH. 2005. Memory—System. http://www.pricewatch.com.

QUANTUM. 2003. Achieving real-time multimedia performance with multistream solid-state disk. http://uk.builder.com/whitepapers/0,39026692,60018746p-39000844q,00.htm.

RIEDEL, E. 1998. A performance study of sequential I/O on Windows NT 4. In *Proceedings of the 2nd USENIX Windows NT Symposium*. Seattle, WA.

ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. 2000. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference*. San Diego, CA.

ROSENBLUM, M. AND OUSTERHOUT, J. 1991. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. Pacific Grove, CA.

SCHINDLER, J., GRIFFIN, J. L., LUMB, C. R., AND GANGER, G. R. 2002. Track-aligned extents: Matching access patterns to disk drive characteristics. In *Proceedings of the USENIX File and Storage Technologies Conference*. Monterey, CA.

SEAGATE. 2003. Cheetah 10K.6 reliability, performance, and low ownership cost. http://www.seagate.com.

SELTZER, M. I., GANGER, G. R., MCKUSICK, M. K., SMITH, K. A., SOULES, C. A. N., AND STEIN, C. A. 2000. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the USENIX Annual Technical Conference*. San Diego, CA.

SHANKLAND, S. 2001. Transmeta taking Linux gadgets mobile. *CNET News.com* http://news.com.com/2100-1001-254020.html?legacy=cnet.

SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. 1996. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference*. San Digeo, CA.

THOMPSON, K. 1978. UNIX implementation. *Bell Syst. Tech. J. 57*, 6, 1931–1946.

TORELLI, P. 1995. The Microsoft flash file system. *Dr. Dobb's J. Feb*, 63–70.

VOGELS, W. 1999. File system usage in Windows NT 4.0. In *Proceedings of the 17th Symposium on Operating Systems Principles*. Kiawah Island, SC.

WANG, A. I. A., KUENNING, G. H., REIHER P., AND POPEK, G. 2003. The effects of memory-rich environments on file system microbenchmarks. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*. Montreal, Canada.

WOODHOUSE, D. 2001. JFFS: The journaling flash file system. http://sources.redhat.com/jffs2/jffs2-html/.

WU, M. AND ZWAENEPOEL, W. 1994. eNVy: A non-volatile, main memory storage system. In *Proceedings of the 6th Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA.

Received November 2005; revised April 2006; accepted April 2006