The Kudos File System Architecture

Andrew de los Reyes, Chris Frost, Eddie Kohler, Mike Mammarella, and Lei Zhang
University of California, Los Angeles
{adlr,frost,kohler,mikem,leiz}@cs.ucla.edu

ABSTRACT

We propose a file system implementation architecture, called *Kudos*, where *change descriptor* structures represent any and all changes to stable storage. Kudos is decomposed into fine-grained modules which generate, consume, forward, and manipulate change descriptors. The uniform change descriptor abstraction allows modules to impose and follow arbitrary file system consistency policies: a collection of loosely-coupled modules cooperates to implement strong and possibly complex guarantees, even though each individual module does a relatively small part of the work. For example, by observing and modifying change descriptor constraints, our journaling module can automatically add journaling to any file system. Additionally, a new system call interface gives applications some direct control over change descriptors. We have used this interface to improve the UW IMAP server, removing inefficient and unnecessary calls to `fsync()` while preserving the integrity of mail messages. We have implemented Kudos as both a Linux kernel module and a FUSE [2] userspace file system server. Our current untuned implementation is competitive with FreeBSD soft updates for number of blocks written.

1 INTRODUCTION

Ongoing massive growth in both disk capacities and user and application storage requirements, combined with the increasing disparity between disk and processor speeds, have led to ongoing innovation in file system design. File systems are expected to do more things: they can automatically encrypt stored data [31], support extensible metadata, such as content indexes [7], automatically version [5, 19, 20, 27], and even detect viruses [18]. Improved disk interfaces can make use of file system usage patterns to improve performance [23, 25].

As file system functionality increases, file system *correctness* is also increasingly a focus of research [8, 24]. File system extensions and disk-layer interventions risk violating the dependency relationships between written blocks on which file system correctness relies [10]. Stackable file systems [12, 21, 34, 35] avoid some of these issues by placing extensions over an existing file system, which is responsible for maintaining consistency. Unfortunately, not all extensions are easily implemented entirely above an unmodified file system. Furthermore, a stackable file system may implement a single system call with multiple operations; for instance, imagine creating a file whose additional metadata information is stored in another file with a related name. Since current VFS (virtual file system) implementations can't express dependencies among operations, the stackable file system must must either impose an ordering using `fsync`, which is expensive, or accept and account for the possibility of inconsistency. This problem exists for applications as well. The expensive `fsync` and `sync` system calls are the only tools available for enforcing file consistency. Robust applications, including databases, mail servers, and source code management tools, either accept the performance penalty of these system calls or rely on specialized raw-disk interfaces.

Proposed systems for improving file system integrity and consistency differ mainly in the kind of consistency they aim to impose, ranging from metadata consistency to full data journaling [29] and even full ACID transactions [9, 16]. However, different extensions within a file system, or different applications over the file system, may require different types of consistency semantics, and performance suffers when lower layers are unnecessarily denied the opportunity to reorder writes [10]. A better choice might be an abstraction that could express many consistency models.

This work develops a modular file system architecture called Kudos that facilitates both file system extension and consistency maintenance. We decompose the entire file system layer, from system calls to block devices, into pluggable modules, hopefully making the system as a whole more configurable, extensible, and easier to understand. Kudos modules are built around a new fundamental data type called a *change descriptor*. A change descriptor represents both a change to disk data and any *dependencies* between that change and other changes. Change descriptors were inspired by the dependency abstraction from BSD's soft updates [10], but whereas the soft updates implementation is specific to the UFS file system, change descriptors are fully general. Any file system can generate change descriptors, and those change descriptors are obeyed by all other file system layers. Change descriptors may be examined and modified by other modules as well, and even passed through layers such as loop-back file systems. As a result, the

loosely-coupled modules that make up a file system implementation can cooperate to implement strong and often complex consistency guarantees, even though each individual module does a relatively small part of the work.

Change descriptors can implement many consistency mechanisms, including soft updates and journaling. Our file system modules impose soft updates-style change descriptor requirements by default. A single module placed downstream of any file system can analyze these change descriptor layouts and transform them into dependencies that implement a journal. Additionally, we give user-level applications controlled access to change descriptors, allowing applications to impose their own limited consistency policies. Modifying an IMAP mail server to use this interface requires only localized changes; the resulting server follows IMAP's consistency requirements while writing fewer blocks to disk. Although implementation experience is preliminary, and change descriptors do not yet support all consistency policies (for instance, not ACID transactions: transactions should be independent, but any file system client can observe all active change descriptors), we believe the Kudos architecture will allow the construction of consistent, modular, extensible file systems that are much easier to understand.

Our contributions include the generalized change descriptor design, Kudos's module interfaces (a particular innovation is the separation of the low-level specification of on-disk layout from higher-level file-system-independent code), particular modules including the journal module and the write-back cache, and the changegroup interface that exports change descriptors to userspace.

2 Related Work

Stackable File Systems Stackable module software for file systems continues to attract active research [11, 12, 21, 26, 32, 33, 34, 35]. Previous systems like FiST [34] or GEOM [3] generally focus on an individual portion of the system and thus restrict both what a module can do and how modules can be arranged. FiST, for instance, does not provide a way to deal with structures on the disk directly – it provides only "wrapper" functionality around existing file systems. GEOM, on the other hand, deals only with the block device layer, and has no way to work with the file systems stored on those block devices. Neither has a formal way of specifying or honoring complex write-ordering information, which is what change descriptors in Kudos provide. We imagine that systems like these could be adapted to work with change descriptors, giving the benefits of both ideas.

Consistency Soft updates [10] significantly lowers the overhead required to provide file system consistency. By carefully ordering writes to disk, soft updates avoids the need for synchronous writes to disk or duplicate writes to a journal. Soft updates also guarantees a strong level of consistency after a crash, enough so that the system can avoid time-consuming file system consistency checks using a utility like *fsck*.

Another approach to protecting the integrity of the file system is to write upcoming operations to a journal first. The content and the layout of the journal varies in each implementation, but in all cases, the system can use the journal to play out or roll back the operations that did not complete as a result of a crash. Thus, *fsck* can be avoided by consulting the journal when recovering from a crash. Section 5.2 explains journaling with change descriptors. [22] compares journaling and soft updates in practice.

[14]

Customizable application-level consistency protocols have previously been considered in the context of distributed, parallel file systems by CAPFS [30]. In such a system, enforcing an unnecessary consistency protocol can be extremely expensive, and not providing the right consistency protocol can cause unpredictable failures. However, this is also true with a local file system – and as a result, applications must use expensive interfaces like `fsync()` when they require specific consistency guarantees. Kudos brings this sort of customizable consistency to all applications, not just those using specialized distributed file systems.

Applications A variety of extensions to file systems have been proposed in recent work, like the FS2 Free Space File System [13] and encrypting file systems like NCryptfs [32]. Although we have not implemented Kudos modules for these extensions, we believe that they would be relatively easy to implement.

3 Design

We first describe the design of Kudos's basic abstractions: block descriptors, change descriptors, and file system modules.

3.1 Block Descriptors

Disk blocks are represented by objects called *block descriptors*. Kudos maintains at least one block descriptor object for each cached block; in certain cases, such as encrypted file systems, where the on-disk representation of a block differs fundamentally from the memory representation, a block may have more than one block descriptor.

Kudos expects to be able to write blocks atomically. Thus, the modules that interface with the disk return blocks that are 512 bytes long (the sector size). A block resizer module joins sector-sized blocks into a more convenient size for the relevant file system, such as 2048 or

```
struct chdesc {
    BD_t *device;
    bdesc_t *block;
    enum {BIT, BYTE, NOOP} type;
    union {
        struct {
            uint16_t offset;
            uint32_t xor;
        } bit;
        struct {
            uint16_t offset, length;
            uint8_t *data;
        } byte;
    };
    struct chdesc_queue *dependencies;
/* ... */ };
```

Figure 1: Partial change descriptor structure.

4096 bytes, and splits these large blocks into sector-sized chunks as they are written.

3.2 Change Descriptors

Each in-memory modification to a cached disk block has an associated change descriptor. Each change descriptor applies to data on exactly one block, and each block links to all associated change descriptors. Different change types correspond to different forms of change descriptors; the change descriptor for a flipped bit – such as in a free-block bitmap – contains an offset and mask, while larger changes contain an offset, a length, and the new data. The change descriptor's *dependencies* point to other change descriptors that must precede it to stable storage. A change descriptor can be applied or reverted to switch the cached block's state between old and new.

Figure 1 gives a simplified version of the structure, and Figure 2 shows most of the API for working with them. The ability to revert and re-apply change descriptors is inspired by soft updates dependencies, but generalized so that it is not specific to any particular file system. Change descriptor dependencies can create cyclic dependencies among blocks, although the change descriptors themselves must never form a cycle. To handle this case, some change descriptors may need to be rolled back, or reverted, in order to write the others, allowing such cycles to be broken.

Kudos modules change blocks by attaching change descriptors to them, using functions such as `chdesc_create_bit`. Most file system modules initially generate change descriptors whose dependencies impose soft-update-like ordering requirements (see §5.1). These change descriptors are then passed down, through other modules, in the general direction of the disk. The intervening modules can inspect, delay, and even modify them before passing them on further. For instance, the write-back cache module (§4.1), which is essentially a buffer cache, holds on to blocks and their change descriptors instead of forwarding them immediately. When evicting a block and associated change descriptors, the write-back cache enforces an order consistent with the change descriptor dependency information.

```
chdesc_t *chdesc_create_noop(
    bdesc_t *block, BD_t *owner);
chdesc_t *chdesc_create_bit(
    bdesc_t *block, BD_t *owner,
    uint16_t offset, uint32_t xor);
int chdesc_create_byte(
    bdesc_t *block, BD_t *owner,
    uint16_t offset, uint16_t length,
    const void *data, chdesc_t **head);
int chdesc_create_diff(
    bdesc_t *block, BD_t *owner,
    uint16_t offset, uint16_t length,
    const void *newdata, chdesc_t **head);
int chdesc_add_depend(
    chdesc_t *depender, chdesc_t *dependee);
void chdesc_remove_depend(
    chdesc_t *depender, chdesc_t *dependee);
int chdesc_apply(chdesc_t *chdesc);
int chdesc_rollback(chdesc_t *chdesc);
int chdesc_satisfy(chdesc_t *chdesc);
int chdesc_push_down(
    BD_t *current_bd, bdesc_t *current_block,
    BD_t *target_bd, bdesc_t *target_block);
```

Figure 2: Partial change descriptor API.

A change descriptor is *satisfied* when the change to which it corresponds has been resolved—for instance, written to disk. If one change descriptor depends on another, then the depender (the change descriptor depended on) must be satisfied before the dependee (the change descriptor that depends) can be written to disk itself.

Each change descriptor on a block may or may not be visible to a given module. For example, modules that respond to user requests generally view the most current state of every block – the block with all change descriptors applied. However, a write-back cache may choose to write some change descriptors on a block while reverting others, since those others currently have unsatisfiable dependencies. In this case, modules below the write-back cache should view the unsatisfied change descriptors in the reverted state. Kudos provides a block revisioning library function that automatically rolls back those change descriptors that should not be visible at a particular module, and then rolls them forward again after that module is done with the block.

A collection of simple operations, such as shifting a change descriptor from one block to another, checking whether a change descriptor has unsatisfiable dependencies, and copying a change descriptor (useful for modules such as RAID), are handled by common library functions available to any module.

No-op Change Descriptors The prototypical change descriptor corresponds to some change on disk, but Kudos also supports a *no-op* change descriptor type, which doesn't change the disk at all. No-op change descriptors can have dependencies, like any other change descriptor, but they don't need to be written to disk: they are trivially satisfied when all of their dependencies are

satisfied. Thus, they can be used to "stand for" entire sets of other changes. This capability is extremely useful, and is used by most operations on disk structures so that a single change descriptor can be returned that depends on the whole change. Likewise, a no-op change descriptor can be passed in as a parameter to a disk operation to make the whole operation depend on a set of other changes. No-op change descriptors allow dependencies between sets without a quadratic number of dependency edges in the change descriptor graph, and without having to pass around arrays of change descriptors. The cost is that some functions may have to traverse trees of no-op change descriptors to determine true dependencies.

Modules can also use no-op change descriptors to *prevent* changes from being written. A *managed* no-op must be explicitly satisfied; any changes that depend on that no-op are delayed until the owning module explicitly satisfies it. This is used, for instance, by the journal module (§5.2) to prevent a transaction's change descriptors from being written before the journal commits.

3.3 Module Interfaces

A complete Kudos configuration is composed of many modules. New modules are simple to write, and by changing the module arrangement, a broad range of behaviors can be implemented. It's also easy to tell what behavior a given arrangement will give just by looking at the connections between the modules.

Kudos has three major types of modules. Closest to the disk are block device (BD) modules, which have a fairly conventional block device interface with interfaces such as "read block" and "flush". Closest to the system call interface are *common file system* (CFS) modules, which have an interface similar to VFS [15]. In traditional systems, a CFS-like module would be connected directly to a BD-like module in order to set up a file system for mounting. In Kudos, however, there is an intermediate interface which interposes between block devices and the high-level CFS interface. This *low-level file system* (LFS) interface helps divide file system implementations into common (reusable) code and file system specific code. A Kudos file system designer combines modules with all three interfaces in many ways – a departure from stackable file systems, which act only at the VFS/CFS layer. Kudos modules are implemented in C using structures of function pointers to achieve object oriented behavior.

The LFS interface (Figure 3) has functions to allocate blocks, add blocks to files, allocate file names, and other file system micro-ops. A module implementing the LFS interface should define how bits are laid out on the disk, but doesn't have to know how to combine the micro-ops into larger, more familiar file system operations. A generic CFS-to-LFS module decomposes the larger file write, read, append, and other standard operations into

```
int (*get_root)(inode_t *inode);
uint32_t (*allocate_block)(
    fdesc_t *file, int purpose,
    chdesc_t **head);
int (*free_block)(
    fdesc_t *file, uint32_t block,
    chdesc_t **head);
bdesc_t *(*lookup_block)(uint32_t number);
int (*write_block)(
    bdesc_t *block, chdesc_t **head);
int (*lookup_name)(
    inode_t parent, const char *name,
    inode_t *inode);
fdesc_t *(*lookup_inode)(inode_t inode);
void (*free_fdesc)(fdesc_t *fdesc);
uint32_t (*get_file_numblocks)(fdesc_t *file);
uint32_t (*get_file_block)(
    fdesc_t *file, uint32_t offset);
int (*get_dirent)(
    fdesc_t *file, struct dirent *entry,
    uint16_t size, uint32_t *basep);
int (*append_file_block)(
    fdesc_t *file, uint32_t block,
    chdesc_t **head);
uint32_t (*truncate_file_block)(
    fdesc_t *file, chdesc_t **head);
fdesc_t *(*allocate_name)(
    inode_t parent, const char *name,
    uint8_t type, fdesc_t *link,
    inode_t *newinode, chdesc_t **head);
int (*remove_name)(
    inode_t parent, const char *name,
    chdesc_t **head);
int (*rename)(
    inode_t oldparent, const char *oldname,
    inode_t newparent, const char *newname,
    chdesc_t **head);
int (*get_metadata)(
    inode_t inode, uint32_t id,
    size_t size, void *data);
int (*set_metadata)(
    inode_t inode, uint32_t id,
    size_t size, const void *data,
    chdesc_t **head);
```

Figure 3: Important functions in the LFS interface.

LFS micro-ops. This one module, called the "universal high-level file system" or UHFS, can be used with many different LFS modules implementing different file systems. (There is also a generic VFS-to-CFS layer, which has two implementations: one for the Linux kernel module, and one for the FUSE [2] version. See §7 for details.)

The UHFS module encompasses all logic for decomposing higher level CFS calls into lower level LFS calls, and for connecting the resulting change descriptors together. The finer granularity of LFS calls divides the problem space into smaller chunks. Since the issue of how to tie the LFS micro-ops together has already been solved, file system module developers can give more attention to the particulars of the file system, such as how to allocate a new filename or how to look up the Nth data block for a file. To see how this simplifies the development process, consider the VFS `write()` call, which has the task of writing some amount of data to a file at a given offset. In Kudos, the logic to determine the correct offsets within blocks and whether new blocks must be allocated is built into UHFS. A file system module need only implement four LFS

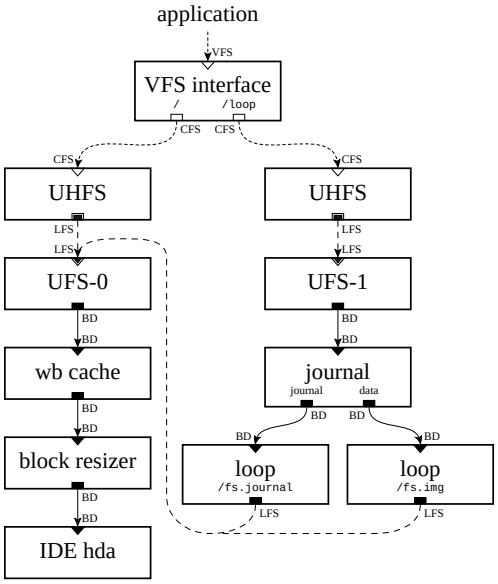

Figure 4: A running Kudos configuration. */* is a soft updated file system on an IDE drive; */loop* is an externally journaled file system on loop devices.

calls: `get_file_block()`, `allocate_block()`, `append_block()`, and `write_block()`. Additionally, the granularity of calls at the LFS layer makes it an appropriate layer for inserting test harnesses and developing file system unit tests.

Figure 4 shows a somewhat contrived example taking advantage of the LFS interface and change descriptors. A file system image is mounted with an external journal, both of which are loop devices on the root file system, which uses soft updates. The journaled file system's ordering requirements are sent through the loop device as change descriptors, allowing dependency information to be maintained across boundaries that might otherwise lose that information. In contrast, without change descriptors and the ability to forward change descriptors through loop devices, BSD cannot express soft updates' consistency requirements through loop-back file systems. The modules in Figure 4 are a complete and working Kudos configuration, and although the use of a loop device is somewhat contrived in the example, they are increasingly being used in conventional operating systems. For instance, Mac OS X uses them in order to allow users to encrypt their home directories.

4 MODULES

This section describes some Kudos modules which contribute most to the change descriptor related functionality of the system. There are several other block device modules, like the write-through cache, partitioner, memory block device, and loop-back block device, which perform more traditional functions, and a handful of CFS and LFS modules like the block device file system module, file hiding module, and whole disk access module, which demonstrate various other abilities of the module system.

4.1 Write-Back Cache

Write-back cache modules act as the system's primary cache, and are responsible for holding on to the change descriptors sent to them until they can be safely sent on towards the disk. There can be many write-back caches in a configuration at once (for instance, one for each block device). To a write-back cache, the complex consistency protocols that other modules want to enforce are nothing more than sets of dependencies among change descriptors – it has no idea what consistency protocols it is implementing, if any at all. Yet it is the module that ends up doing most of the work to make sure that change descriptors are written in an acceptable order.

Despite this, write-back caches are relatively simple – only about 550 lines of code, including comments. The current write-back cache uses a simple LRU policy, except that blocks can't be evicted unless all the change descriptors on them are ready to be sent to the next module (for example, the disk). A change descriptor is ready if all of its dependencies are at least as close to the disk as it is. So, when looking for a block to evict, the cache may not be able to evict any block it chooses – but it evicts the least recently used block that it can.

The write-back cache may need to write a non-evictable block anyway, however. For instance, change descriptors may be arranged in a way that creates a dependency cycle at the level of blocks, even though the change descriptor dependencies themselves are acyclic. The same solution as soft updates uses is employed to break the cycle: the write-back cache just holds on to the change descriptors that cannot yet be written, and forwards the others on to the next module as it writes the block. It cannot evict the block yet, since it is still "dirty," but progress has been made that can make other blocks evictable.

The write-back cache has one other property that makes it useful in Kudos: it respects dependencies between one cache and another, so that (for instance) dependencies between the changes on a file system and its external journal are properly respected. This is actually not a special case, nor does it require any extra code – it is a property that just falls out of the way change descriptor graphs are processed in order to determine which change descriptors are ready to move towards the disk. This property also extends to changegroups, which are explained in §6.

4.2 Elevator Scheduler

The elevator scheduler is like a simpler version of the write-back cache. Unlike the write-back cache, it will not

hold on to change descriptors indefinitely until their dependencies have been satisfied. Its purpose is instead to function like a "peephole optimizer" on the block writes done by some other module, while respecting any dependencies among the change descriptors on the blocks being written. Effectively, a sliding window of writes to the disk (or some other module) is rearranged subject to the constraints mandated by the change descriptor dependencies. Having a separate module to do this job, rather than the write-back cache itself, allows us to separate the policy the elevator scheduler implements (in this case, elevator scheduling) from the main write-back cache code.

4.3 UFS

Kudos supports two base file system types, a simple inode-less implementation called JOSFS and a version of UFS (Unix File System), the modern incarnation of the Fast File System [17] used in 4.2 BSD. We describe UFS as an example of a complete and relatively complex file system.

The general concepts for UFS, such as inodes, cylinder groups, and indirect blocks, are well understood. In Kudos, we implemented the UFS1 file system as an LFS module. UFS is of particular interest because it is the only file system that has been extended with both soft updates and journaling. [22] We chose UFS1 over UFS2, as UFS1 is well established and more widely supported.

The UFS module is implemented at the LFS interface. This keeps properties specific to UFS (such as the UFS on-disk format and rules governing block allocation) hidden within the file system module. For instance, UFS uses 2KB *fragments* to store small files efficiently. Once a file gets big enough to require the use of indirect blocks, then UFS changes its allocation policy and starts allocating 16KB *blocks*, where a block is made up of 8 aligned and contiguous fragments. Our UFS module implement this by using fragments as the basic block size. For large files, our UFS module internally allocates a block, but returns only the first fragment in that block at the LFS level. The next 7 allocation calls will be no-ops that simply return the subsequent fragments in the allocated block. In this way, UFS can stay consistent internally without special support from other Kudos modules.

The UFS module creates change descriptors for all its changes to the disk, and connects them to form configurations that achieve soft updates consistency. Unlike FreeBSD's soft updates implementation, once the dependencies have been hooked up, UFS no longer needs to concern itself with the change descriptors it has created. The block device subsystem will track and enforce the change descriptor orderings.

In order to implement soft update ordering, we examined the places where UFS created change descriptors. With the consistency rules for soft update in mind, we carefully connected the change descriptorscreated within an LFS call. Assured that all LFS calls in the UFS module followed soft updates ordering, we then made sure that the combinations of LFS calls do not violate soft updates consistency.

4.3.1 Modularity

Although the UFS implementation must be somewhat monolithic in order to understand the interlinked UFS-specific disk structures, we would like to take advantage of modularity to allow for flexibility and encapsulation. UFS is divided into modules at boundaries that expose naturally replaceable policies, namely in resource allocation and directory entry traversal, as well as for objects such as the superblock and cylinder groups. We use specialized interfaces specific to UFS, because these interfaces do not apply to all file systems in general.

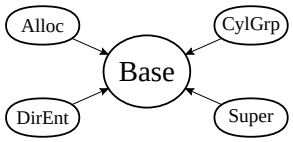

Figure 5: UFS and its submodules.

Within the UFS module, the internal organization is shown in Figure 5. The *base* module contains the core code for the file system. Using object-orientation techniques, we encapsulated the code for data structures in the file system, like the superblock and the cylinder groups. We also recognized two locations where we can apply different search algorithms. One is with respect to the allocation functions for blocks, fragments, and inodes. The other is for searching and writing directory entries (see Figure 6). Currently we have two allocator modules as a proof of concept for modularity.

Cylinder Group Interface:

```
const UFS_cg_t * read(int num);
int write_time(int num, int value, chdesc_t ** head);
int write_rotor(int num, int value, chdesc_t ** head);
int write...
int sync(int num, chdesc_t ** head);
```

Allocator Interface:

```
int find_free_block(fdesc_t * file, int purpose);
int find_free_frag(fdesc_t * file, int purpose);
int find_free_inode(fdesc_t * file, int purpose);
```

Figure 6: Example UFS Internal Module Interfaces

The UFS cylinder group module interface regulates access to the cylinder groups. While there is unrestricted read access to cylinder groups, the interface limits write access to certain fields within the *UFS_cg* struct. This is because, under normal operations, fields like the number of data blocks per cylinder group remain constant. The cylinder group module can also define the policy for when changes to the cylinder group are written to disk.

It can, for example, make the policy "write-through" and have all changes immediately go to disk. However, choosing this policy means that on every block allocation, UFS needs to write *cg_rotor*, the position of the last used block, to disk. To avoid performance hits like this, we implemented the "write-back" policy instead. To support flushing dirty data to disk, the cylinder group module interface also has a sync call. Similarly, the super block module interface controls access to the super block.

The UFS allocator module interface has methods for allocating blocks, fragments, and inodes. All three methods take in a *file descriptor* and a *purpose* variable. The UFS *file descriptor* contains all relevant information pertaining to a given file, including an in-memory copy of the file's inode. The intent of the *purpose* variable is to provide hints to the allocator about how the newly allocated resource will be used. Given these two pieces of information, as well as cylinder group information from the previously mentioned module, UFS allocator modules can make informed decisions to allocate resources in an efficient manner. For instance, given the status for all cylinder groups, a block allocator function can examine the blocks previously allocated to a given file and figure out what block to allocate next based on the strategies used by FreeBSD.

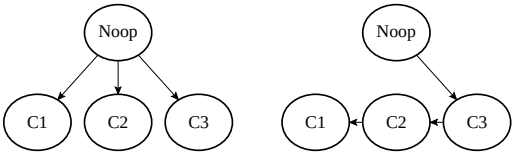

Figure 7: Change descriptor dependencies, when not strictly needed, restrict the possible choices for write ordering. This results in suboptimal write ordering and more scans through the change descriptors for Kudos. On the left, change descriptors C1, C2, and C3 can be written in any order. Only one ordering is possible on the right.

4.3.2 Change Descriptor Arrangements

While optimizing our UFS implementation, we gained some insight and found a couple practices critical to good performance. First, do not create unnecessary dependencies between change descriptors. Doing so artificially limits the commit order for change descriptors, which results in bad performance for several reasons. Not only will unneeded dependencies force the disk to do more writes and seeks, but Kudos will have to scan through the change descriptor graph multiple times, since the dependencies prevent the change descriptors from being flushed out to disk at once.

In Figure 7, we have two possible arrangements for three byte change descriptors. The noop change descriptor represents a root node that can reach all other change descriptors. In the parallel arrangement on the left, Kudos has the freedom to write change descriptors C_1, C_2, and C_3 to disk in any order. All three change descriptors can be marked writable with one graph traversal. In the serial arrangement on the right, there exists only one valid write ordering. For Kudos to write this arrangement out to disk, it will have to scan through the graph three times, since change descriptor C_n cannot be marked as writable until C_{n-1} has been written. An instance like this came up because our UFS implementation frequently writes the *cylinder group summaries* out to disk. By simply changing the arrangement between three change descriptors created in a single function, UFS got a 33% speed increase for several common file operations.

A corollary of this observation is to create change descriptors of the minimal size to avoid accidental overlaps, which in turn, create unnecessary dependencies. Change descriptors can represent changes to regions as small as one byte, and as large as an entire block. Many times, it can be tedious for developers to calculate exactly what parts in a large data structure have been modified and need to be written to disk. As such, laziness will result in the creation of change descriptors for the entire data structure. Doing this can be detrimental to performance, as shown in Figure 8.

An occurrence of this problem came up for inodes, where we make several independent modifications to different fields within an inode. In principle, the change descriptors created are conflict-free. Previously, because we did not take the time to calculate the exact offsets and lengths for the fields that changed, we just created a change descriptor for the entire inode every time we modified any part of the inode. Thus all changes to any particular inode would always overlap, causing unnecessary dependencies. Our solution to this problem is a utility function called `chdesc_create_diff()`, which compares a modified copy of a data structure to the original, and creates a minimal set of change descriptors accordingly. Due to the frequent use of inodes, one simple use of `chdesc_create_diff()` in the UFS inode functions reduced change descriptor graph traversal time significantly. Although this does introduce a slight overhead, due to the need to *diff* the two structures.

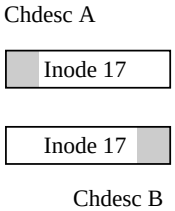

Figure 8: On inode 17, the gray regions represent modified fields that do not overlap. If change descriptor A and change descriptor B are exactly the size of the gray regions, then there is no implicit dependency. Making change descriptors for the entire inode data structure will, in turn, make one change descriptor depend on the other because they overlap.

4.4 RAID

We have implemented a RAID mirroring module. The code is actually very straightforward, and not very long. It attaches to two block devices and presents a single block device. Either of the two subordinate block devices can fail without the RAID module reporting error. Read requests are passed to both devices, alternating based on the stride size. Write and sync requests are passed ideally to both devices, but maybe to only one subordinate device if the other is down.

The RAID module also has the ability to attach a subordinate block device and sync its contents so that it mirrors the existing subordinate block device. In this way, RAID can be used to hot-swap disks: start out with a RAID module and only one subordinate block device, attach a second subordinate disk, wait for synchronization to complete, remove the original subordinate block device. A userspace tool provides the ability to force a subordinate device to go into failed mode.

Because RAID works simply by transforming change descriptors, the subordinate block devices do not need to be physical disks. They may just as well be loop-back devices, memory devices, network block devices, or any combination.

5 CONSISTENCY

Soft updates, journaling, and many application-specific consistency models all correspond to different change descriptor arrangements, so these features can be added to the system as modules which appropriately connect or reconnect the change descriptors. For soft updates, the change descriptors created by the file system module must be connected according to the rules in [10], and the write-back cache will take it from there. For journaling, a journal module is added above the write-back cache that rearranges the change descriptors to provide journaling. Other semantics could be supported by the addition of similar modules. In this section, we will discuss the soft updates and journaling support in Kudos.

5.1 Soft Updates

Generally, a file system image is consistent if a program like *fsck* would report no errors – that is, all the structures are in a completely correct organization. Soft updates aims, by enforcing orderings between write, to maintain a file system in a relatively consistent state at all times, speeding reboot by avoiding *fsck*. Not all invariants can be preserved, so soft updates relaxes the least critical: a crash or sudden reboot can leak blocks or other disk structures, but can never create more serious inconsistencies.

In the FFS implementation of soft updates, each file system operation is represented by a structure in memory encapsulating the structural changes to the disk image necessary to implement that operation. As a result, there exist many specialized data structures to represent the different possible file system operations. These structures, their relationships to one another, and their uses for tracking and enforcing dependencies are all complex. In Kudos, change descriptors are used to store all changes to the disk. Rather than a single structure that represents the whole file system operation, several change descriptors are created, one for each range of bytes on the disk which must be changed. These change descriptors are connected to specify the order in which the changes must be written to disk. For instance, when a block is removed from a file, we create (at least) two change descriptors in most file systems: one that clears out the reference to that block number in the file's list of blocks, and one that marks the block as free. By hooking up the second change descriptor to depend upon the first, we can implement the soft updates semantic straightforwardly. Another example is depicted in Figure 9.

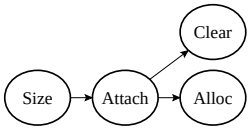

Figure 9: Soft updates change descriptor graph, including the dependencies for adding a newly allocated block to an inode. Writing the new block pointer to an inode (Attach) depends on initializing the block (Clear) and updating the free block map (Alloc). Updating the size of the inode (Size) depends on writing the block pointer.

The Kudos approach certainly takes more memory than the soft updates approach, which limits the state required by any individual file system operation; we do not yet address the issue of memory exhaustion. However, Kudos does separate dependency enforcement from dependency specification. This makes the actual implementation easier to read, and allows the dependency structure to be examined and modified by other modules of the system that may not have any idea what the changes are actually doing.

5.2 Journaling

Although change descriptors might initially seem to be specifically designed to implement soft updates-like consistency semantics, they are in fact much more flexible and can be used to implement journaling as well. What distinguishes journaling from soft updates (from the point of view of change descriptors and the write-back cache) is that with soft updates, change descriptors can always be written to the disk in order to empty the cache, while journals can "lock" changes into the cache when transactions are in progress.

To accomplish this, the journal module makes all change descriptors passing through it depend on a managed no-op change descriptor(§3.2), and makes copies of

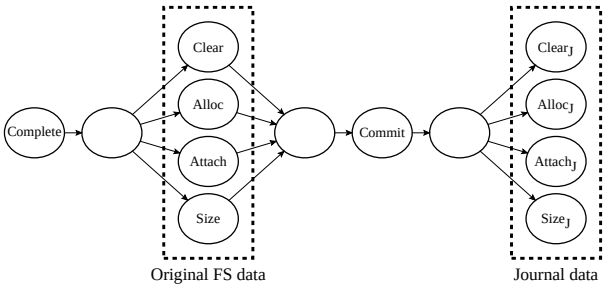

Figure 10: Journal change descriptor graph for the change in Figure 9. Empty circles are "no-op" change descriptors with no associated block data.

them to write to the journal. When the transaction is over, it creates a commit record, sets all the change descriptors in the transaction to depend on it, and satisfies the managed no-op change descriptor.

For example, the change descriptors in Figure 9 can be transformed to provide journaling semantics. The original four change descriptors are modified to depend on a journal commit record, which depends on blocks journaling the changes. This transformation is performed incrementally as change descriptors arrive. Once the actual changes commit, the journal record is marked as completed. Figure 10 shows these transformed change descriptors. The resulting journal on disk is similar in format to those generated by ext3 [29] – it has a list of block numbers, followed by the data which should be in those blocks. Finally, there is a commit record which applies to the whole set.

A particularly nice property of this arrangement is that the journal module is completely independent of any specific file system. It is a block device module that automatically journals whatever file system is stored on it. Further, by changing our journal module to journal only change descriptors that modify file system metadata – and by adding additional dependencies to prevent premature reuse of blocks – we could even obtain metadata-only journaling (as opposed to the full data journaling described here). The extra change descriptor dependencies would serve the same purpose as the special hooks and corner cases surrounding reuse of blocks discussed in [28]. The journal module can automatically identify metadata change descriptors because of the LFS interface described in §3.3. Other block device layering systems, like GEOM [3] or JBD in Linux, would or do need special hooks into file system code to determine what disk changes represent metadata in order to do metadata-only journaling. Change descriptors and the LFS interface allow us to do this automatically.

6 CHANGEGROUPS

Kudos modules use change descriptors to express internal file system consistency protocols, such as soft updates and journaling (§5). Applications also implement consistency protocols; for example, CVS clients synchronize local files with the server, and IMAP servers must commit mailbox changes to disk. In this section we discuss the extension of change descriptors from Kudos to user applications to support custom application-level consistency protocols. We discuss the changegroup interface and implementation and a case study of changegroups in the UW IMAP server.

6.1 Interface

Our current interface allows applications to specify orderings among file system operations. A changegroup corresponds to the group of change descriptors generated by a file system operation; applications can specify ordering requirements among these groups. Figure 11 shows the changegroup interface functions.

```
typedef int changegroup_id_t;
changegroup_id_t changegroup_create(int type);
int changegroup_sync(changegroup_id_t cg);
int changegroup_add_depend(
    changegroup_id_t depender,
    changegroup_id_t dependee);
int changegroup_engage(changegroup_id_t cg);
int changegroup_disengage(changegroup_id_t cg);
int changegroup_release(changegroup_id_t cg);
int changegroup_abandon(changegroup_id_t cg);
```

Figure 11: Changegroup Interface

`changegroup_add_depend()` makes one changegroup depend on another. Until `changegroup_release()` is called, a changegroup's change descriptors will not go to disk. `changegroup_sync()` synchronizes a changegroup by committing all its changes, preserving its dependencies. Each file system operation's change descriptors are added to the *engaged set* of changegroups. `changegroup_engage()` and `changegroup_disengage()` add and remove a changegroup from this set. Lastly, `changegroup_abandon()` invalidates a changegroup handle for the calling process.

The changegroup interface uses a set of engaged changegroups for each process rather than a changegroup parameter in file system calls to match common changegroup usage (an application operation composed of multiple file system operations), and to avoid changing existing file system calls. Each process is associated with a set of changegroups to support modular composition.

To allow changegroup interactions among processes, child processes copy their parent's engaged set and changegroup ID table. This behavior also allows a process to run a helper process whose file system changes will all be contained within a changegroup. For example, suppose a user wants to run in their shell:

```
sort < src > sorted && rm src
```

However, the user wants to ensure that `src` is not removed before `sorted` is committed to disk, so that a

crash part way through does not lose both files. Using changegroups, the user could make `rm`'s updates depend on `sort`'s. A program `depend`, that creates a changegroup for each argument, evaluates the argument as a shell command, and makes the changegroup depend on the previous argument, can do just this:

```
depend "sort < src > sorted" "rm src"
```

With an application's ability to create dependencies among changegroups, cyclic dependencies could arise. The changegroup interface ensures that cycles can only arise in changegroup operations, ensuring file system operations cannot fail due to cycles. The changegroup interface limits cycle creation opportunity through the allowed changegroup state transitions, shown in Figures 12 and 13, which ensure that dependees cannot be added when the changegroup contains any change descriptors or has any dependers.

State Requirement	State Change
none	released
none	abandoned (end)
no dependers and released	engaged
engaged	disengaged
no dependers and not released	has dependees
disengaged	has dependers

Figure 13: Changegroup State Transitions — alternative view of Figure 12. A fresh changegroup is not released, is not abandoned, is disengaged, and has no dependees or dependers.

"tail" change descriptor and a "head keep" change descriptor, while the tail change descriptor depends on a "tail keep" change descriptor. The purpose of the "keep" change descriptors is to prevent the head and tail change descriptors from being satisfied should all their other dependencies be satisfied (for instance, by being written to disk). An LFS module in the module graph is responsible for connecting all operations performed while a changegroup is engaged to its head and tail change descriptors, so that an entire changegroup can be made to depend on another changegroup by adding a dependency from the depender's tail to the dependee's head.

Changegroups are created and managed almost exactly like file descriptors. Handles to them are copied across `fork()`, preserved across `exec()`, and closed upon `exit()`. Whether a changegroup is engaged or not is a property local to the process, like the "close-on-exec" flag of file descriptors. These properties allow existing programs to use changegroups without knowing it, in the same way that they don't have to know anything about pipes for the shell to connect them in a pipeline – the parent process creates and engages the changegroups before calling `exec()`, and then the process does its file system operations without knowing or needing to know that they are being hooked up to a changegroup.

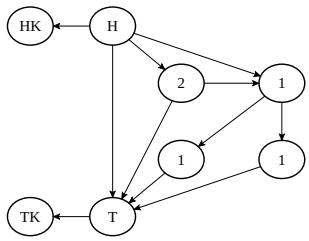

Figure 14: Changegroup Change Descriptors. H and T are the head and tail of the changegroup; HK and TK are the head keep and tail keep. The three change descriptors labeled 1 occurred during one file system operation, and the change descriptor labeled 2 occurred during a second operation that depended on the first.

Figure 12: Changegroup State Transitions. Edges represent changegroup operations: **A**dd depende**E**, **A**dd depende**R**, **R**elease, **E**ngage, and **D**isengage. Not drawn: every state connects to the end node via abandon.

6.2 Implementation

Changegroups are implemented using an arrangement of no-op change descriptors which allows the changegroup to be connected to other changegroups using ordinary change descriptor dependencies. Figure 14 shows this arrangement: a "head" change descriptor depends upon a

6.3 UW IMAP Case Study

To assess the utility of changegroups in a real world application we updated the University of Washington's IMAP mail server to take advantage of changegroups, to

ensure mail updates are committed to disk in safe and efficient orderings.

The Internet Message Access Protocol (IMAP) [6] provides remote access to a mail server's email message store. An IMAP client is in one of four states: not authenticated, authenticated, mailbox selected, and logged out. States define the set of valid commands a client may issue. For example, clients in the selected state may retrieve messages from the selected mailbox. In our case study we are interested in mailbox manipulation; the authenticated and selected states allow mailbox manipulation. The most relevant IMAP commands synchronize changes to the server's disk (CHECK), copy a message from the selected mailbox to another mailbox (COPY), and delete messages marked for deletion (EXPUNGE).

UW IMAP [4], authored by Mark Crispin of the IMAP RFC, includes an IMAP mail server imapd and mail client library c-client. We have updated imapd and the mbox mail storage driver to use changegroups to ensure all disk changes occur in a safe ordering, while performing a minimal number of disk writes. The original IMAP server conservatively preserved command ordering by syncing the mailbox file after each CHECK on it or COPY into it. With changegroups, each command's file system updates are executed under a distinct changegroupand, through the changegroup, made to depend on the previous command's updates. This is necessary, for example, so that moving a message to another folder (accomplished by copying to the destination file and then removing from the source file) can not lose the copied message should the server crash part way through the disk updates. The updated CHECK command uses `changegroup_sync()` to sync all preceding disk updates. This enhancement to CHECK removes the requirement that COPY sync its destination mailbox; the client's CHECK request will ensure changes are committed to disk; the changegroup dependencies will ensure changes are committed in a safe ordering.

These changes to UW IMAP simplify two aspects: ensuring change ordering correctness and making efficient usage of disk updates. As each command's changes now depend on the preceding command's changes, it is no longer required that all code be concerned with ensuring its changes are committed before any later, dependent command's changes. Without changegroups, modules like the mbox driver forced a conservative disk sync protocol because ensuring safety more efficiently required additional state information, adding further complexity. The Dovecot IMAP server's source code notes this exact difficulty [1, maildir-save.c]:

```
/* FIXME: when saving multiple messages, we could get
   better performance if we left the fd open and
   fsync()ed it later */
```

The performance of the changegroup-enabled UW IMAP mail server is evaluated in §8.3.

7 IMPLEMENTATION

Kudos was originally implemented as a stand-alone file system server daemon for a small operating system (called "KudOS"), but now runs as either a FUSE [2] file system server under Linux or BSD, or as a Linux kernel module. The former is particularly useful for profiling and debugging, while the latter is best for real-world performance testing and actual use.

The Kudos Linux kernel module looks like a VFS file system to Linux, but unlike traditional file system modules, it is merely a front end to a Kudos module graph. The Kudos server starts up a kernel thread which performs all the initialization to get the module graph set up, and modules then can make file systems available to mount though the VFS front end. Each file system can be mounted using a command like `mount -t kfs kfs:`*`name`*` /mnt/point`.

The Linux kernel must be modified in order to provide support for changegroups. As a kernel module, there is no way to get notifications about when processes are created, or when they terminate; this is necessary to clone the changegroup scope from the parent process when the process is created, and to abandon all the changegroups when it terminates. Ordinarily, this sort of state would be kept as part of the Linux `task_struct` (i.e. process) structure, but this is not a viable option for a dynamically loadable kernel module. Instead, the Kudos module keeps this state internally and uses new kernel hooks to be notified when processes fork or exit.

At the other end of the Kudos module graph, the terminal block device modules are wrappers for Linux's normal block devices. By using Linux's generic block layer, Kudos can interface with any existing block device (e.g. RAM disk, IDE disk, etc.). When the module receives a read or write call, it passes the request to Linux via `generic_make_request()`. Linux is configured to use a 'noop' elevator, meaning that the calls we make to `generic_make_request()` are sent to the physical disk in the exact order we make them. Kudos takes advantage of DMA capabilities offered by Linux to do disk reads and writes efficiently.

The module also implements rudimentary read-ahead. When a read call needs to be made to the physical device, the module checks to see if any of the following nine blocks are in memory already. If any of them are not, they are also requested at the same time. After all blocks are read in, the read call is complete.

Our interfaces bypass all Linux caches, because Kudos provides its own change descriptor-aware caches (§4.1) in order to provide its write-ordering guarantees.

Our current implementation uses Linux's queuing structures for queuing I/O requests for block devices. Although this is working, there are a couple subtle prob-

lems that we would like to solve in a future version. First of all, while we can be sure that writes to a single block device are delivered in our exact order, we cannot be sure that writes we make to multiple block devices are made in the correct order relative to each other. Also, we would like to be able to insert read and write requests to different sides of the I/O queue. Right now a read request will be satisfied only after all the pending writes are completed, but a better queue design would allow reads to "cut the line" since they are more urgent.

8 EVALUATION

We evaluate our prototype implementation of Kudos in two ways: first, we show that the overall performance is within reason, even though it is slower than Linux or BSD by a nontrivial amount. We justify this difference, and argue that it can be improved substantially. Second, we show specific block write orderings in a variety of situations, demonstrating the effect and utility of changegroups and the potential for them to improve the efficiency of applications using them.

8.1 Overall Performance

Our first benchmark test is to untar the Linux 2.6.15 source code, from `linux-2.6.15.tar` which is already decompressed and cached in RAM. We did this test on identical machines running Linux 2.6.12 (using ext3), FreeBSD 5.4 (using UFS, with and without soft updates), and Kudos running as a Linux 2.6.12 kernel module (using UFS, with and without soft updates). As a second test, we then delete the resulting source tree. The times below include time to fully sync the changes to disk.

System	Untar (sec)	Delete (sec)
Kudos (SU)	24.50	11.69
Kudos (async)	20.14	11.27
FreeBSD 5.4 (SU)	20.47	4.71
FreeBSD 5.4 (async)	9.63	3.72
Linux 2.6.12 (journaling)	12.90	4.90

Figure 15: Untar/delete times for Kudos, FreeBSD, and Linux, with soft updates (SU) and with async mode.

Kudos is about 20% slower than FreeBSD using soft updates for writes, and a little over a factor of two slower for deletions. This shows that, although it is slow, Kudos is close to running in acceptable time. While we do have CPU usage and I/O delay problems, we have only focused on performance for a few weeks, in which we were able to speed up the system enormously. There are more optimization opportunities that can likely bring us very close in performance to existing implementations of soft updates (see §9).

8.2 Block Writes

We have also looked at the number of block writes that Kudos makes relative to other systems for several operations. In one test, we create 100 small files in a directory and measure the number of blocks written to disk. We do this with and without soft updates. Kudos writes 122 blocks, while FreeBSD writes 135 blocks.

We also look at a very small benchmark: removing a file from a directory and adding a new file to it. We compare the number of blocks written by Kudos with that of FreeBSD using soft updates. Here Kudos writes 15 blocks, while FreeBSD writes 83 blocks.

We ran the two small benchmarks with the journal module, which does full data journaling. Creating 100 small files with the journal resulted in 245 blocks written to the disk, but only 126 blocks without the journal. Deleting and adding a file resulted in writing 31 with the journal module and 18 without it.

From this we gather that our UFS, soft updates, and journal implementations are correct, and this reinforces our case that we need to reduce CPU usage to be competitive.

8.3 UW IMAP Case Study

To assess the changegroup-enabled UW IMAP mail server (§6.3) we compare the number of block writes that Kudos makes relative to FreeBSD to move 100 emails. The test selects the source mailbox (with 100 messages, sized 2kB each), creates the new mailbox, copies each of the 100 messages to the new mailbox, marks each source message for deletion, expunges the marked messages, requests a check, and logs out. We compare using soft updates, Figure 16 shows the results.

System	# writes (blocks)
Kudos (SU)	919
FreeBSD 5.4 (SU)	2200

Figure 16: Number of block writes to move 100 IMAP messages.

9 DISCUSSION

There are several areas in which we would like to improve our work. The obvious first area we would like to work on is the performance of Kudos. We have already improved the performance by literally several orders of magnitude, since we only recently began examining performance in addition to correctness, but the system is not as fast as it could be – and not quite as fast as it needs to be to be a viable option for most computer systems.

We would also like to identify places where Kudos can be made more flexible, by adding more features to the system and changing the existing parts that make these features difficult to add. For instance, we would like to find more applications which can take advantage

of changegroups, and see what changes might need to be made to the changegroup interface in order to facilitate adapting those applications.

There are several ways in which the performance can be improved. For instance, we create a very large number of change descriptors, and the sheer number of them can cause problems for any algorithm which needs to traverse parts of the change descriptor dependency graph. We can attack this problem in two ways: first, we can reduce the number of change descriptors by intelligently merging them when we determine that having separate change descriptors is not necessary. For example, if the file system creates a file and immediately deletes it, we can take the two opposing change descriptors sets and coalesce them into a no-op.

As another strategy, we can improve the efficiency of the traversal algorithms to examine fewer change descriptors. Many of the operations that modules like the write-back cache need to do involve quickly discovering the set of change descriptors that satisfy some property, and often even some ordering of those change descriptors. In early prototypes, these sets were often calculated by doing large traversals of the change descriptor dependency graph. However, these traversals are in practice extremely expensive, and optimizing them has improved and will likely continue to improve the performance of Kudos.

We have worked on both of these approaches, especially the second, but we believe further improvements can be made. One simple change which helped enormously was hidden in the order that change descriptors were listed as dependencies of other change descriptors. We had originally used a simple linked list for this structure, and treated it like a stack for ease of insertion to the list. It turned out that a much better way to use the linked list was to treat the list like a queue, because almost always, elements being removed from the list were at the end of the list. (Since they had been added longest ago, and often changes are written in an order similar to the order in which they were made.) This converted a linear time search into one which would usually execute in constant time.

Another example is that when examining a block and deciding which change descriptors must be rolled back in order for the block's data to be safe to write to the disk, the dependency graph between the various changes on that block must be taken into account so that changes which depend on one another are rolled back in proper dependency order. Once again, we discovered that we were examining the change descriptors in almost exactly the opposite order as the order they should be rolled back in – for the same reason as the previous example. By simply changing the direction of the loop, we were able to improve the loop from quadratic time to linear time.

For the Kudos UFS module, we want to add more features to make the implementation complete. Currently, we do not support symbolic links, triple indirect blocks, or sparse files. The first two can easily be implemented given time. Sparse file support will require the LFS interface to be aware of files with holes. This also means the layer immediately above the LFS interface may need to adjust some of the assumptions it currently makes about files and block allocation.

We have not taken advantage of the UFS module's infrastructure for supporting more sophisticated allocation algorithms. There are auxiliary on-disk data structures that assist resource allocation algorithms. We currently ignore many of them, partially because our basic algorithms do not use them, and partially due to the lack of documentation to explain the purpose of some data structures. Over time, we hope to better understand UFS and improve our UFS implementation.

10 Conclusion

Kudos provides a new way for file system implementations to formalize write ordering requirements, allowing such software to be divided into modules which cooperate loosely to implement strong consistency guarantees. It simplifies many aspects of the design of consistency protocols like journaling and soft updates by separating the specification and enforcement of the write ordering requirements. Additionally, the new abstraction can be extended outside the file system implementation into userspace, allowing applications to specify in a limited way what their specific consistency requirements are, providing the file system implementation more freedom to reorder writes without violating the application's needs. We argue that this system can be made fast enough for general purpose use, and we demonstrate with a case study that the extension of the new Kudos abstraction into userspace can improves the efficiency of applications with specific write ordering requirements.

References

[1] *Dovecot.* Version 1.0 beta7, `http://www.dovecot.org/`.

[2] *FUSE – Filesystem in Userspace.* `http://fuse.sourceforge.net/`.

[3] *GEOM – modular disk I/O request transformation framework.* `http://www.freebsd.org/cgi/man.cgi?query=geom&sektion=4`.

[4] *UW IMAP toolkit.* `http://www.washington.edu/imap/`.

[5] B. CORNELL, P. DINDA, and F. AND. Wayback: A user-level versioning file system for linux, 2004.

[6] M. Crispin. Internet Message Access Protocol - version 4rev1. RFC 3501, IETF, March 2003.

[7] M. A. S. David K. Gifford, Pierre Jouvelot and J. W. O. Jr. Semantic file systems. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, October 1991.

[8] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the Fourth USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[9] E. Gal and S. Toledo. A transactional Flash file system for microcontrollers. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 89–104, Anaheim, California, Apr. 2005.

[10] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, May 2000.

[11] J. S. Heidemann and G. J. Popek. A Layered Approach to File System Development. Technical report, UCLA, Los Angeles, CA, March 1991.

[12] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1): 58–89, Feb. 1994.

[13] H. Huang, W. Hung, and K. G. Shin. FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption. In *Proceedings of 20th ACM Symposium on Operating Systems Principles*, pages 263–276, October 2005.

[14] E. D. I. in Storage: Techniques and Applications. Gopalan sivathanu and charles p. wright and erez zadok, 2005.

[15] S. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 USENIX Summer Technical Conference*, pages 238–47, Atlanta, Georgia, 1986.

[16] B. Liskov and R. Rodrigues. Transactional file systems can be fast. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

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

[18] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An on-access anti-virus file system. In *Proceedings of the 13th USENIX Security Symposium (Security '04)*, San Diego, California, Aug. 2004.

[19] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March/April 2004. USENIX Association.

[20] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey,CA, 2002.

[21] D. S. H. Rosenthal. Evolving the vnode interface. In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, California, Jan. 1990.

[22] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.

[23] M. Sivathanu, V. Prabhakaran, F. Popovici, T. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco, California, Mar. 2003.

[24] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha. A Logic of File Systems. In *Proceedings of the Fourth USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[25] M. Sivathanu, L. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco, California, Dec. 2005.

[26] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A Progress Report. In *Proceedings of the 1993 USENIX Summer Technical Conference*, pages 161–174, Cincinnati, OH, 1993.

[27] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco, California, Mar. 2003.

[28] S. Tweedie. Ext3, Journaling Filesystem, July 2000. `http://olstrans.sf.net/release/OLS2000-ext3/OLS2000-ext3.html`.

[29] S. Tweedie. Journaling the Linux ext2fs Filesystem. In *LinuxExpo '98*, 1998.

[30] M. Vilayannur, P. Nath, and A. Sivasubramaniam. Providing tunable consistency for a parallel file store. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco, California, Dec. 2005.

[31] C. Wright, M. Martino, and E. Zadok. Ncryptfs: A secure and convenient cryptographic file system, 2003.

[32] C. P. Wright, M. C. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, Texas, June 2003.

[33] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and Unix semantics in namespace unification. *ACM Transactions on Storage*, Mar. 2006.

[34] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 55–70, June 2000.

[35] E. Zadok, I. Badulescu, and A. Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 57–70, Monterey, California, June 1999.