First Appeared in the "USENIX Symposium on Operating Systems Design and Implementation", November, 1994, pp. 49-60.

Metadata Update Performance in File Systems

Gregory R. Ganger, Yale N. Patt
Department of EECS, University of Michigan
ganger@eecs.umich.edu

Abstract

Structural changes, such as file creation and block allocation, have consistently been identified as file system performance problems in many user environments. We compare several implementations that maintain metadata integrity in the event of a system failure but do not require changes to the on-disk structures. In one set of schemes, the file system uses asynchronous writes and passes ordering requirements to the disk scheduler. These **scheduler-enforced** ordering schemes outperform the conventional approach (synchronous writes) by more than 30 percent for metadata update intensive benchmarks, but are suboptimal mainly due to their inability to safely use delayed writes when ordering is required. We therefore introduce **soft updates**, an implementation that asymptotically approaches memory-based file system performance (within 5 percent) while providing stronger integrity and security guarantees than most UNIX file systems. For metadata update intensive benchmarks, this improves performance by more than a factor of two when compared to the conventional approach.

1 Introduction

File system metadata updates traditionally proceed at disk speeds rather than processor/memory speeds [Ousterhout90, McVoy91, Seltzer93], because synchronous writes are used to properly order stable storage changes. This update sequencing is needed to maintain integrity in the event of a system failure (e.g., power loss).[1] For example, the *rename* operation changes the name of a file by adding a link for the new name and removing the old link. If the system goes down after the old directory block has been written (with the link removed) but before the new one is written, neither name for the file will exist when the system is restarted. To protect metadata consistency, the new directory entry must reach stable storage before the old directory block. We refer to this ordering requirement as an update *dependency*, as writing the old directory block *depends* on first writing the new block. The ordering constraints essentially map onto three simple rules: (1) Never reset the old pointer to a resource before the new pointer has been set (when moving objects), (2) Never re-use a resource before nullifying all previous pointers to it, and (3) Never point to a structure before it has been initialized.

Synchronous[2] writes are used for metadata update ordering by many variants of both the original UNIX^{TM} file system [Ritchie78] and the Berkeley fast file system (FFS) [McKusick84]. The performance degradation can be so dramatic that many implementations choose to ignore certain update dependencies. For example, a pointer to a newly allocated block should not be added to a file's inode before the block is initialized on stable storage. If this ordering is not enforced, a system failure could result in the file containing data from some previously deleted file, presenting both an integrity weakness and a security hole. However, *allocation initialization* with synchronous writes can degrade performance significantly. As a result, most UNIX file system implementations, including FFS derivatives, either do not force initialization or force initialization only for newly allocated directory blocks in order to protect the integrity of the directory hierarchy. We investigate the performance cost of allocation initialization in our comparison of different ordering schemes.

Previous schemes that address the performance penalty of update ordering generally entail some form of logging (e.g., [Hagmann87, Chutani92, Journal92]) or shadow-paging (e.g., [Chamberlin81, Ston87, Chao92, Seltzer93]). While these approaches have been successfully applied,

[1] For complete integrity, each individual update must also be atomic (not partially written to disk). This can be achieved by forcing each critical structure to be fully contained by a single disk sector. Each disk sector is protected by error correcting codes that will almost always flag a partially written sector as unrecoverable. This may result in loss of structures, but not loss of integrity. In addition, many disks will not start laying down a sector unless there is sufficient power to finish it.

[2] There are three types of UNIX file system writes: *synchronous*, *asynchronous* and *delayed*. A write is synchronous if the process issues it (i.e., sends it to the device driver) immediately and waits for it to complete. A write is asynchronous if the process issues it immediately but does not wait for it to complete. A delayed write is not issued immediately; the affected buffer cache blocks are marked dirty and issued later by a background process (unless the cache runs out of clean blocks).

there is value in exploring implementations that do not require changes to the on-disk structures (which may have a large installed base).

The remainder of this paper is organized as follows. Section 2 describes our experimental setup, measurement tools and base operating system. Sections 3 and 4 describe several approaches to “safe" metadata updates, including our implementations of each. Section 3 describes schemes in which the file system uses asynchronous writes and passes any ordering restrictions to the disk scheduler with each request. Section 4 describes soft updates, a file system implementation that safely performs metadata updates with delayed writes. Section 5 compares the performance of the different schemes. Section 6 compares important non-performance characteristics, such as user-interface semantics and implementation complexity. Section 7 draws some conclusions and discusses avenues for future research. The appendix describes some low-level details of our soft updates implementation.

2 Experimental apparatus

All experiments were performed on an NCR 3433, a 33MHz Intel 80486 machine equipped with 48 MB of main memory (44 MB for system use and 4 MB for a trace buffer). The HP C2447 disk drive used in the experiments is a high performance, 3.5-inch, 1 GB SCSI storage device [HP92]. Our base operating system is UNIX SVR4 MP, AT&T/GIS's production operating system for symmetric multiprocessing. We use the *ufs* file system for our experiments, which is based on the Berkeley fast file system [McKusick84]. The virtual memory system is similar to that of SunOS [Gingell87, Moran87], and file system caching is well integrated with the virtual memory system. The scheduling code in the device driver concatenates sequential requests, and the disk prefetches sequentially into its on-board cache. Command queueing at the disk is not utilized.

One important aspect of the file system's reliability and performance is the *syncer daemon*. This background process executes at regular intervals, writing out dirty buffer cache blocks. The syncer daemon in UNIX SVR4 MP operates differently than the conventional “30 second sync"; it awakens once each second and sweeps through a fraction of the buffer cache, marking each dirty block encountered. An asynchronous write is initiated for each dirty block marked on the previous pass. This approach tends to reduce the burstiness associated with the conventional approach.

We run all experiments with the network disconnected and with no other non-essential activity. We obtain our measurements from two sources. The UNIX *time* utility provides total execution times and CPU times. We have also instrumented the device driver to collect I/O traces, including per-request queue and service delays. The traces are collected in the 4 MB trace buffer mentioned above and copied to a separate disk after each experiment. The timing resolution is approximately 840 nanoseconds, and the tracing alters performance by less than 0.01 percent (assuming that the trace buffer could not be otherwise used).

In section 5, we use several benchmarks to compare the performance of the metadata update schemes described in the next two sections. To concisely quantify the performance impacts of some of the implementation decisions, however, it will be useful to provide small amounts of measurement data with the descriptions. For this purpose, we use the results from two metadata update intensive benchmarks. In the N-user copy benchmark, each “user” concurrently performs a recursive copy of a separate directory tree (535 files totaling 14.3 MB of storage taken from the first author's home directory). In the N-user remove benchmark, each “user” deletes one newly copied directory tree. Each datum is an average of several independent executions, with coefficient of variation (standard deviation / mean) below 0.05.

3 Scheduler-enforced ordering

With scheduler-enforced ordering, the responsibility for properly sequencing disk writes is shifted to the disk scheduler (generally part of the device driver). The file system uses asynchronous writes and augments each request with any ordering requirements. We examine two levels of ordering information: a simple flag and a list of specific dependencies.

3.1 Ordering flag

A straight-forward implementation of scheduler-enforced ordering attaches a one-bit flag to each disk request (as suggested in [McVoy91]). Write requests that would previously have been synchronous for ordering purposes (i.e., writes that may require ordering with respect to subsequent updates) are issued asynchronously with their ordering flags set. Of course, the disk scheduler must also be modified to appropriately sequence flagged requests.

The most significant implementation issue is the semantic meaning of the flag, which represents a contract between the file system and the disk scheduler. The ordering semantics determine which subsequent requests can be scheduled before a flagged request and which previous requests can be scheduled after a flagged request. With the most restrictive semantics, a flagged request acts as a barrier. Less restrictive meanings offer the disk scheduler more freedom but may require the file system to set the flag more frequently, reducing scheduling flexibility. In general, we find that less restrictive flag semantics result in improved performance. In particular, allowing read requests to bypass the list of writes waiting on ordering constraints[3] can improve performance significantly without endangering metadata integrity.

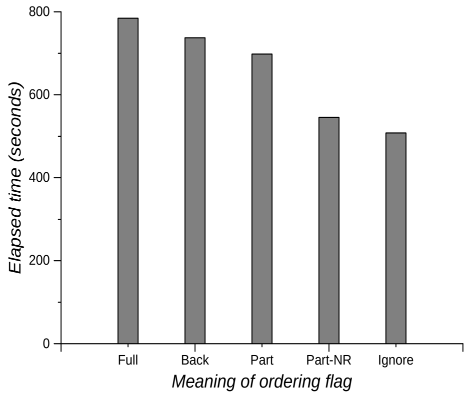
[3] Unless the read requests are for locations to be written, of course.

We compare four meanings for the ordering flag: *Full*, *Back*, *Part* and *Ignore*. In *Full*, a flagged request acts as a full barrier (i.e., all previous requests must complete before it is scheduled and no subsequent requests can bypass it). *Back* prevents requests issued after a flagged request from being scheduled before it or any previous request, but allows the flagged request to be re-ordered freely with previous non-flagged requests. This scheme is less restrictive than *Full* but still allows several requests to be ordered with respect to a later request with only the last such request issued with the flag set. *Part* further relaxes the constraints by requiring only that requests issued after a flagged request not be scheduled before it (i.e., previous non-flagged requests can be re-ordered freely with subsequent requests and the flagged request). With this flag meaning, all requests that require ordering with respect to any subsequent request must have the flag set. The addition of *-NR* to any scheme indicates that the disk scheduler allows non-conflicting read requests to bypass the list of writes waiting because of ordering restrictions.[4] Finally, *Ignore* re-orders requests freely, ignoring the flag. This scheme does not protect metadata integrity and we include it only for comparison.

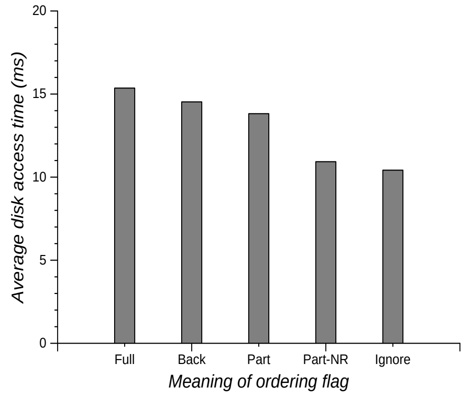
For the 4-user copy benchmark (figure 1), performance improves with each reduction in the flag's restrictiveness. Reducing the number of requests with the flag set should improve performance by increasing the disk scheduler's freedom to re-order. Such cases do occur, but too infrequently to counter the increased restrictiveness of the flag's meaning. We also found that allowing reads to bypass flag-pending writes improves performance significantly. The disk access times (figure 1b) directly display the impact of allowing the scheduler greater freedom. The elapsed times (figure 1a) show how this translates into overall performance. This trend, less restrictive flag semantics results in higher performance, holds for most of our benchmarks, and the comparisons in section 5 utilize *Part-NR*.

The 1-user remove benchmark (figure 2) is an exception to this rule. The elapsed times (figure 2a) are system response times observed by the benchmark "user." As the write requests that remove the directory tree are issued, a very large queue builds up in the scheduler. This effect is evident from the average driver response times (i.e., the times from when requests are issued to the device driver to when they complete, including both queue times and disk access times) of 5+ seconds shown in figure 2b. When read requests bypass this queue, the "user" process wastes very little time waiting for I/O, and the benchmark completes without waiting for the driver queue to empty (the writes fit into main memory). Given the *-NR* option, the **more** restrictive flag semantics result in lower user-observed response times, because fewer requests interfere with read requests

[4] Note that this could reasonably be viewed as an implementation decision rather than part of the flag semantics. The ordering required for metadata integrity pertains only to writes.



(a) Elapsed time (seconds)



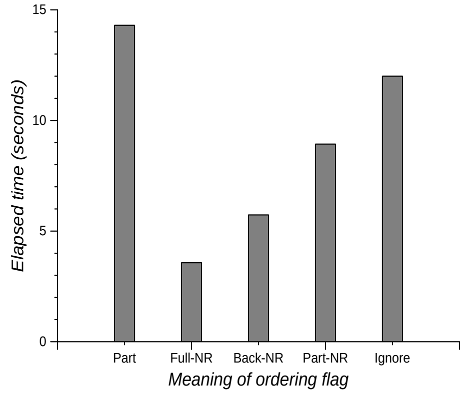
(b) Average disk access time (ms)

Figure 1: Performance impact of ordering flag semantics for the 4-user copy benchmark.

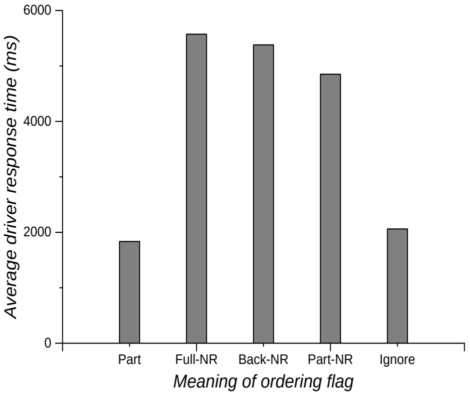
for service. This demonstrates the performance advantage, in sub-saturation bursts of activity, of giving preference to those requests which block processes [Ganger93]. However, tying the disk scheduler's hands is a poor way of realizing this performance improvement and does not behave well when system activity exceeds the available memory (as shown above).

3.2 Scheduler chains

Even with the least restrictive semantics for the ordering flag, requests are often constrained unnecessarily by flagged writes. By tagging each disk request with a unique identifier and a list of requests on which it depends (i.e., a list of requests that must complete before it can be scheduled), such "false" dependencies can be avoided. We refer to this approach as *scheduler chains*. Similar approaches exist in other systems. The MPE-XL^{TM} file system provides support for ordered sequences of user writes, although write-



(a) Elapsed time (seconds)



(b) Average driver response time (ms)

Figure 2: Performance impact of ordering flag semantics for the 1-user remove benchmark.

ahead logging protects the metadata [Busch85, Kondoff88]. [Cao93] describes a method for supporting request dependencies in an intelligent storage controller (or a device driver). As we are not dealing with multiple hosts and an interconnection network, our disk scheduler support can be less complete. In particular, we do not allow requests to depend on future requests; a new request can depend only on previously issued requests. In addition to the increased scheduler complexity, the file system must maintain information regarding which dirty blocks depend on which outstanding requests. In most cases, this is straight-forward because newly updated blocks depend on just-issued requests.

The exception to this rule is block de-allocation. A de-allocated block should not be re-used before the old pointer has been re-initialized on stable storage. Generally, the de-allocation is independent of subsequent re-use; at the least, they usually occur during separate system calls. We examined two approaches to maintaining the required ordering. The first falls back on the flag-based approach. The asynchronous write of the inode (or indirect block) is issued as a *Part-NR* barrier (i.e., no subsequent write request is scheduled before it completes). The second approach maintains information about recently freed blocks until the re-initialized pointer reaches the disk. The blocks can be re-allocated at any time and the new owner (inode or indirect block) becomes dependent on the write of the old owner. In fact, we make the newly allocated block itself dependent on the old owner. This prevents new data from being added to the old file due to untimely system failure. The barrier approach is obviously simpler to implement, but can cause unnecessary dependencies and thereby reduce performance. As with the comparison of flag meanings, we find that the less restrictive approach provides superior performance (e.g., 16 percent for the 4-user remove benchmark). Therefore, the second approach was used for the scheduler chains data reported in section 5.

3.3 Avoiding write locks

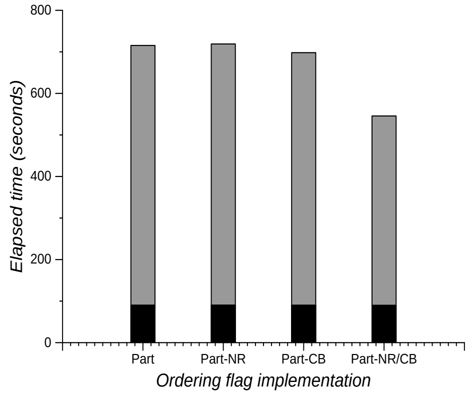
Our initial implementation of scheduler-enforced ordering revealed that, although metadata writes are no longer synchronous, processes still wait for them in many cases. This occurs when multiple updates to given file system metadata occur within a short period of time. When a write request is issued to the device driver, the source memory block(s) are write-locked until the request completes.[5] This prevents subsequent updates from occurring while the I/O subsystem hardware may be accessing the data. As a result, a second update must wait for the first to reach stable storage. One solution is to make a second (temporary) in-memory copy of the memory block(s) before issuing the first request. This copy becomes the source for the first write request, obviating the need to write-lock the original at the cost of a block copy operation.[6] To avoid unnecessary overhead for the special case of allocation initialization, we reserve a zero-filled memory block when the system is booted. This block becomes the source for initialization writes.[7]

Figure 3 compares four different implementations of the *Part* ordering flag scheme described above: with no options, with the *-NR* option, with the block copying (*-CB*) and with both. With both (*Part-NR/CB*), user processes spend the least time waiting for disk requests. Failing to include either enhancement greatly reduces the benefit. For this

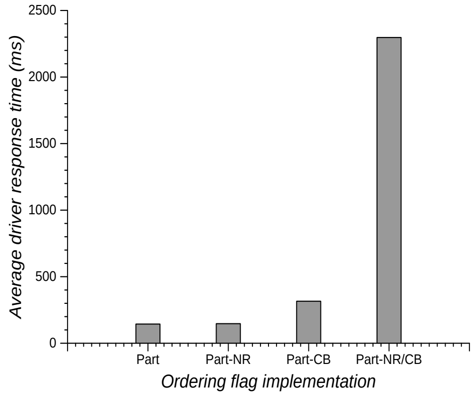
[5] "Less critical" source data, such as file blocks and virtual memory pages, often bypass this safety precaution. For file blocks in particular, this is a difficult performance vs. reliability trade-off/judgment call made by system implementors.

[6] Copy-on-write would clearly be a superior approach. We plan to investigate this alternative, but do not expect substantially improved throughput (the increase in CPU usage caused by the memory copying is a small fraction of the total time). However, the copy-on-write approach should be more "memory-friendly."

[7] A better approach would utilize an "erase" I/O operation (e.g., the *WRITE SAME* SCSI command [SCSI93]), initializing the disk sectors without wasting a block of memory or transferring a block of zeroes from host memory to disk.

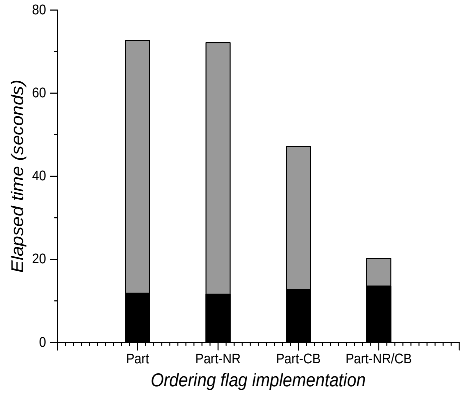


(a) Elapsed time (seconds)

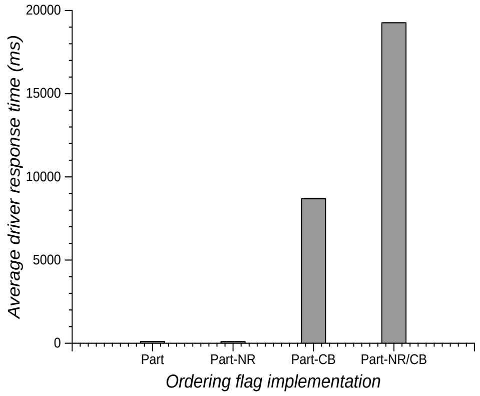


(b) Average driver response time (ms)

Figure 3: Implementation improvements for ordering flags for the 4-user copy benchmark. *Part-NR* allows reads to bypass writes waiting due to ordering restrictions. *Part-CB* uses the block copy scheme to avoid write locks. *Part-NR/CB* combines these two enhancements. The dark region of each elapsed time bar represents the total CPU time charged to the four benchmark processes.



(a) Elapsed time (seconds)



(b) Average driver response time (ms)

Figure 4: Implementation improvements for ordering flags for the 4-user remove benchmark. *Part-NR* allows reads to bypass writes waiting due to ordering restrictions. *Part-CB* uses the block copy scheme to avoid write locks. *Part-NR/CB* combines these two enhancements. The dark region of each elapsed time bar represents the total CPU time charged to the four benchmark processes.

reason, the performance comparisons in earlier subsections used the block copying implementation. The CPU time increases caused by the block copying are small and tend to use time that was otherwise idle (processes waiting for disk I/O). Figure 3b shows the driver response times for each of the implementations. Again we find that the queue grows very large as processes generate requests much more quickly than the disk can service them. The 4-user remove benchmark (figure 4) follows the same general trends, but the performance differences are more substantial. Also, the queueing delays are much larger (almost 20 seconds for *Part-NR/CB*).

We also observe the same general behavior with scheduler chains. The block copying (*-NR* holds no meaning with scheduler chains) reduces the elapsed time by 26 percent for the 4-user copy benchmark and 57 percent for the 4-user remove benchmark.

4 Delayed metadata writes

Delayed metadata writes associate additional information with the in-memory metadata, detailing any ordering constraints on stable storage updates. To complete a structural change, the file system modifies the in-memory copies of the affected metadata (via delayed writes) and updates the corresponding dependency information. The dirty metadata blocks are later flushed by the syncer daemon as described in section 2. Of course, the ordering constraints must be upheld in the process. Delayed metadata writes can substantially improve performance by combining multiple updates into a much smaller quantity of background disk writes. The savings come from two sources: (1) multiple updates to a given metadata component (e.g., removal of a recently added directory entry), and more significantly, (2) multiple independent updates to a given block of metadata (e.g., several files added to a single directory). The next subsection briefly describes our original approach (which is flawed), and the following subsection describes our current implementation.

4.1 Cycles and aging problems

When we began this work, we envisioned a dynamically managed DAG (Directed, Acyclic Graph) of dirty blocks for which write requests are issued only after all writes on which they depend complete. In practice, we found this to be a very difficult model to maintain, being susceptible to cyclic dependencies and aging problems (blocks could consistently have dependencies and never be written to stable storage). Most of the difficulties relate to the granularity of the dependency information. The blocks that are read from and written to disk often contain multiple metadata structures (e.g., inodes or directory fragments), each of which generally contains multiple dependency causing components (e.g., block pointers and directory entries). As a result, originally independent metadata changes can easily cause dependency cycles and excessive aging. Detecting and handling these problems increases complexity and reduces performance.

4.2 Soft updates

Having identified coarse-grain dependency information as the main source of cycles and aging, our most recent implementation (which we refer to as *soft updates*) maintains dependency information at a very fine granularity. Information is kept for each individual metadata update indicating the update(s) on which it depends. A block containing dirty metadata can be written at any time, so long as any updates within that block that have pending dependencies are first temporarily "undone" (rolled back). Thus, the block as written to disk is consistent with respect to the current on-disk state. When the disk write completes, any undone updates are re-established before the in-memory block can be accessed. With this approach, aging problems do not occur because new dependencies are not added to existing update sequences. Dependency cycles do not occur because no single sequence of dependent updates is cyclic. In fact, the sequences are the same as in the original synchronous write approach.

There are four main structural changes requiring sequenced metadata updates: (1) block allocation (direct and indirect), (2) block de-allocation, (3) link addition (e.g., file creation), and (4) link removal. We implement block allocation and link addition with the undo/redo approach outlined above. For block de-allocation and link removal, we defer the freeing of resources until after the newly reset pointer has been written to stable storage. In a sense, these deferred updates are undone until the disk writes on which they depend complete.

When a disk write completes, there is often some processing needed to update/remove dependency structures, redo undone changes, and deal with deferred work, such as block de-allocation and file removal. An implementation of soft updates requires some method of performing these small tasks in the background. Very simple changes can be made during the disk I/O completion interrupt service routine (ISR), which calls a pre-defined procedure in the higher-level module that issued the request. However, any task that can block and wait for a resource (e.g., a lock or, worse yet, an uncached disk block) cannot be handled in this way. Such a task must be handled outside of the ISR, preferably by a background process. We use the syncer daemon (described in section 2) for this purpose. Any tasks that require non-trivial processing are appended to a single *workitem* queue. When the syncer daemon next awakens (within one second), it services the workitem queue before its normal activities.

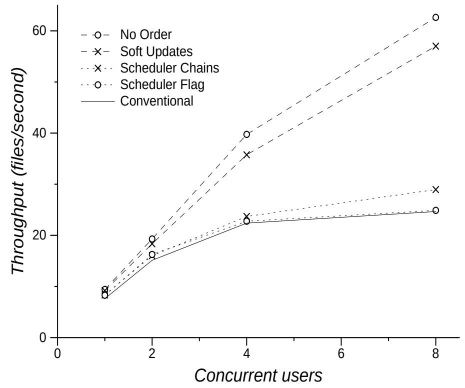
The appendix describes our soft updates implementation in detail, including how each of the four main structural changes is supported.

5 Performance comparison

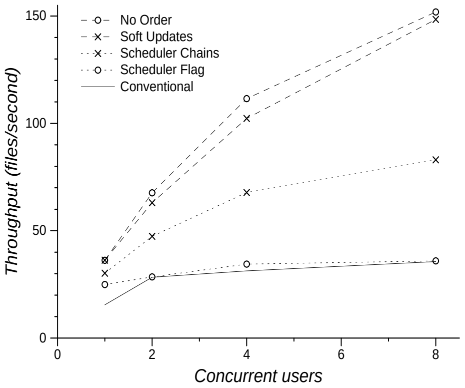
In the following subsections, we compare the performance of five different ordering schemes. As a baseline (and a goal), we ignore ordering constraints (*No Order*) and use delayed writes for all metadata updates. This baseline has the same performance and lack of reliability as the delayed mount option described in [Ohta90]. It is also very similar to the memory-based file system described in [McKusick90]. The *Conventional* scheme uses synchronous writes to sequence metadata updates. The *Scheduler Flag* data represent the *Part-NR/CB* scheduler-enforced ordering flag scheme. The *Scheduler Chains* data represent the best performing scheme described in section 3.2. Both scheduler-enforced ordering schemes use the block copying enhancement described in section 3.3. The *Soft Updates* data are from our current implementation.

5.1 Metadata throughput

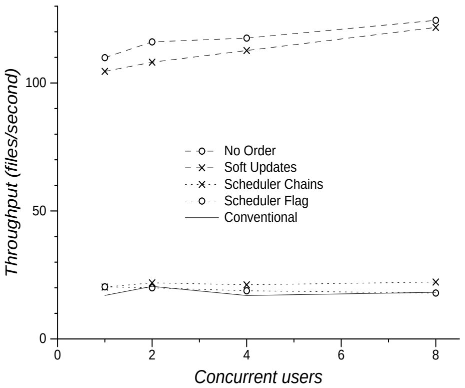
Figure 5 compares the metadata update throughput supported by the five implementations as a function of the number of concurrent "users." Each "user" works in a separate directory. As a result, create throughput improves with the number of "users" because less CPU time is spent checking the directory contents for name conflicts. *Scheduler*



(a) 1KB file creates



(b) 1KB file removes



(c) 1KB file create/removes

Figure 5: Metadata update throughput (files/second). Each data point (10,000 files split among the "users") is an average of several independent executions. All coefficients of variation are less than 0.05. Allocation initialization is enforced only for *Soft Updates*.

Flag reduces metadata update response times compared to *Conventional*, but does not substantially improve throughput. *Scheduler Chains* does better, more than doubling file removal throughput with 8 "users." *No Order* and *Soft Updates* both outperform the other schemes, and the differences increase with the level of concurrency. The power of delayed metadata writes can be seen in figure 5c, where each created file is immediately removed. *No Order* and *Soft Updates* proceed at memory speeds, achieving over 5 times the throughput of the other three schemes. In all cases, *Soft Updates* performance is within 5 percent of *No Order*.

5.2 Metadata intensive benchmarks

Table 1 shows performance data for the 4-user copy benchmark. *No Order* decreases elapsed times by 20 percent and the number of disk requests by 12 percent when compared to *Conventional* with no allocation initialization. *Scheduler Flag* and *Scheduler Chains* decrease the elapsed times by 2 and 4 percent, respectively, but do not affect the number of disk requests. Performance for *Soft Updates* is within a few percent of *No Order* in both elapsed time and number of disk requests. The performance cost of allocation initialization for this benchmark ranges from 3.8 percent for *Soft Updates* to 87 percent for *Conventional*.

The performance differences are more extreme for file removal (table 2), which consists almost entirely of metadata updates. Note that *Soft Updates* elapsed times are lower than *No Order* for this benchmark. This is due to the deferred removal approach used by *Soft Updates*. The order of magnitude decrease in disk activity (e.g., *Soft Updates* verses *Scheduler Chains*) demonstrates the power of delayed metadata writes. The lengthy response times for the scheduler-enforced ordering schemes are caused by the large queues of dependent background writes that form in the device driver.

5.3 Andrew benchmark

Table 3 compares the different schemes using the original Andrew file system benchmark [Howard88]. It consists of five phases: (1) create a directory tree, (2) copy the data files, (3) examine the status of every file, (4) read every byte of each file, (5) compile several of the files.

As expected, the most significant differences are in the metadata update intensive phases (1 and 2). The read-only phases (3 and 4) are practically indistinguishable. The compute intensive compile phase is marginally improved (5-7 percent) by the four non-*Conventional* schemes. The compile phase dominates the total benchmark time because of aggressive, time-consuming compilation techniques and a slow CPU, by 1994 standards.

Ordering Scheme	Alloc. Init.	Elapsed Time (seconds)	Percent of No Order	CPU Time (seconds)	Disk Requests	I/O Response Time Avg (ms)
Conventional	N	390.7	123.9	72.8	36075	293.3
	Y	732.3	232.3	82.4	51419	140.1
Scheduler Flag	N	381.3	120.9	72.8	36038	477.3
	Y	545.7	173.1	90.0	51028	2297
Scheduler Chains	N	375.1	119.0	76.0	36019	304.1
	Y	530.6	168.3	86.0	51248	423.8
Soft Updates	N	319.8	101.4	69.6	31840	368.7
	Y	330.9	104.9	80.0	31880	262.1
No Order	N	315.3	100.0	68.4	31574	304.1

Table 1: Scheme comparison using 4-user copy. Each datum is an average of several independent executions. The elapsed times are averages among the "users", with coefficients of variation less than 0.05. The CPU times are sums among the "users", with coefficients of variation less than 0.1. The disk system statistics are system-wide, with coefficients of variation less than 0.2.

Ordering Scheme	Elapsed Time (seconds)	Percent of No Order	CPU Time (seconds)	Disk Requests	I/O Response Time Avg (ms)
Conventional	80.24	1050	12.68	4600	68.02
Scheduler Flag	24.97	326.8	13.64	4631	22173
Scheduler Chains	31.03	406.2	14.80	4618	2495
Soft Updates	6.71	87.83	5.64	391	73.53
No Order	7.64	100.0	7.44	278	84.03

Table 2: Scheme comparison using 4-user remove. Each datum is an average of several independent executions. The elapsed times are averages among the "users", with coefficients of variation less than 0.05. The CPU times are sums among the "users", with coefficients of variation less than 0.1. The disk system statistics are system-wide, with coefficients of variation less than 0.2.

Ordering Scheme	(1) Create Directories	(2) Copy Files	(3) Read Inodes	(4) Read Files	(5) Compile	Total
Conventional	2.49 (0.50)	4.07 (0.71)	4.08 (0.27)	5.91 (0.31)	295.8 (1.53)	312.4 (1.98)
Scheduler Flag	0.54 (0.50)	4.45 (0.77)	4.09 (0.28)	5.91 (0.29)	279.1 (1.50)	294.1 (1.96)
Scheduler Chains	0.53 (0.50)	3.72 (0.74)	4.09 (0.27)	5.86 (0.35)	280.6 (0.78)	294.8 (1.36)
Soft Updates	0.34 (0.47)	2.77 (0.60)	4.25 (0.43)	5.84 (0.86)	276.3 (0.86)	289.5 (1.32)
No Order	0.37 (0.48)	2.74 (0.52)	4.14 (0.34)	5.84 (0.38)	276.6 (2.58)	289.7 (2.76)

Table 3: Scheme comparison using Andrew benchmark. Each value is in seconds and represents an average of 100 independent executions. The values in parens are the standard deviations. Allocation initialization was enforced only for *Soft Updates*.

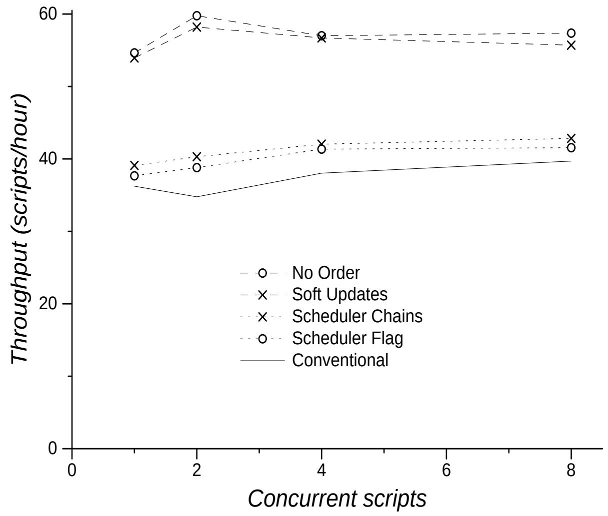


Figure 6: Scheme comparison using Sdet. Each data point is an average of 3 independent executions and all coefficients of variation are less than 0.02. Allocation initialization is enforced only for *Soft Updates*.

5.4 Sdet

Figure 6 compares the five schemes using Sdet, from the SPEC SDM suite of benchmarks. This benchmark [Gaede81, Gaede82] concurrently executes one or more *scripts* of user commands designed to emulate a typical software-development environment (e.g., editing, compiling, file creation and various UNIX utilities). The scripts are generated randomly from a predetermined mix of functions. The reported metric is scripts/hour as a function of the script concurrency.

Scheduler Flag outperforms *Conventional* by 3-5 percent. *Scheduler Chains* provides an additional one percent improvement. *No Order* outperforms *Conventional* by 50-70 percent, and *Soft Updates* throughput is within 2 percent of *No Order*.

6 Non-performance comparisons

6.1 File system semantics

The use of synchronous writes to sequence metadata updates does not imply synchronous file system semantics. In general, the last write in a series of metadata updates is asynchronous or delayed. In many cases, when a file system call returns control to the caller, there is no guarantee that the change is persistent. For link addition and block allocation, the last update adds the pointer to the directory block, inode or indirect block. So, the requested change is not permanent when the system call returns.[8] For link removal and block de-allocation, however, the last update modifies the free maps. When the system call returns, the link is permanently removed and/or the blocks have been freed and are available for re-use. With the scheduler-enforced ordering schemes, freed resources are immediately available for re-use, but links and pointers are not permanently removed when the system call returns. For soft updates, neither is true. In particular, freed resources do not become available for re-use until the re-initialized inode (or indirect block) reaches stable storage.

Some calls have a *SYNCIO* flag that tells the file system to guarantee that changes are permanent before returning. All of the schemes we have described support this interface (although the scheduler-enforced ordering schemes will encounter lengthy delays when a long list of dependent writes has formed). It may be useful to augment additional file system calls (e.g., link addition and link removal) with such a flag in order to support lock files.

6.2 Implementation complexity

Of the schemes we compare, the conventional synchronous write approach is clearly the most straight-forward to implement. Moving to an ordering flag scheme is also straight-forward; the synchronous writes become asynchronous with the flag set. Our changes to the device driver required less than 50 lines of C code. Scheduler-enforced ordering with specific request dependencies is considerably more difficult. Our implementation required about 550 lines of C code for the device driver support and 100 lines for the file system changes. The support for specific remove dependencies adds an additional 150 lines of code. The block-copy enhancement described in section 3.3 required an additional 50 lines of code. Our implementation of soft updates consists of 1500 lines of C code and is restricted to the file system and buffer cache modules. Having learned key lessons from an initial implementation, the first author completed the soft updates implementation in two weeks, including most of the debugging.

7 Conclusions and Future Work

The use of synchronous writes to order metadata updates has been identified as a file system performance problem [Ousterhout90, McVoy91, Seltzer93]. By direct measurement, we have compared several alternative implementations. Schemes in which the file system relies on the disk scheduler to appropriately order disk writes outperform the conventional approach by more than 30 percent in many cases (up to a maximum observed difference of 500 percent). Even with this improvement, however, these schemes

[8] Software locking schemes that use lock files may encounter surprises because of this.

fail to achieve the performance levels available using delayed writes.

Therefore, we have introduced a new mechanism, soft updates, that approaches memory-based file system performance (within 5 percent) while providing stronger integrity and security guarantees (e.g., allocation initialization) than most UNIX file systems. This translates into a performance improvement of more than a factor of 2 in many cases (up to a maximum observed difference of a factor of 15).

The implementations compared in this paper all prevent the loss of structural integrity. However, each requires assistance (provided by the *fsck* utility in UNIX systems) when recovering from system failure or power loss. Unfortunately, the file system can not be used during this often time-consuming process, reducing data and/or system availability. We are investigating how soft updates can be extended to provide faster recovery.

While our experiments were performed on a UNIX system, the results are applicable to a much wider range of operating environments. Every file system, regardless of the operating system, must address the issue of integrity maintenance. Many (e.g., MPE-XL^{TM}, CMS^{TM}, Windows NT^{TM}) use database techniques such as logging or shadow-paging. Others (e.g., OS/2^{TM}, VMS^{TM}) rely on carefully ordered synchronous writes and could directly use our results.

Because the soft updates mechanism appears so promising, we plan to compare it to other popular methods of protecting metadata integrity, such as non-volatile RAM (NVRAM), logging and shadow-paging. NVRAM can greatly increase data persistence and provide slight performance improvements as compared to soft updates (by reducing syncer daemon activity), but is very expensive. Write-ahead logging provides the same protection as soft updates, but must use delayed group commit to achieve the same performance levels. Using shadow-paging to maintain integrity is difficult to do with delayed writes. Combined with soft updates, however, late binding of disk addresses to logical blocks [Chao92] could provide very high performance. The log-structured file system [Seltzer93] is a special case of shadow-paging that provides integrity by grouping many writes atomically (with a checksum to enforce atomicity). The large writes resulting from log-structuring can better utilize disk bandwidth, but the required cleaning activity reduces performance significantly.

We hope to make the non-proprietary components of our implementations available in the near term. If interested, please contact the authors.

8 Acknowledgements

We thank Jay Lepreau ("shepherd" for our paper), John Wilkes, Bruce Worthington and the anonymous reviewers for directly helping to improve the quality of this paper. We also thank our "remote hands" during the summer months, Carlos Fuentes. Finally, our research group is very fortunate to have the financial and technical support of several industrial partners, including AT&T/GIS, Hewlett-Packard, Intel, Motorola, SES, HaL, MTI and DEC. In particular, AT&T/GIS enabled this research with their extremely generous equipment gifts and by allowing us to generate experimental kernels with their source code. The high performance disk drives used in the experiments were donated by Hewlett-Packard.

References

[Busch85] J. Busch, A. Kondoff, "Disc Caching in the System Processing Units of the HP 3000 Family of Computers" , *HP Journal*, 36 (2), February, 1985, pp. 21-39.

[Cao93] P. Cao, S. Lim, S. Venkataraman, J. Wilkes, "The TickerTAIP Parallel RAID Architecture", *ACM ISCA Proceedings*, May 1993, pp. 52-63.

[Chamberlin81] D. Chamberlin, M. Astrahan, et. al., "A History and Evaluation of System R", *Communications of the ACM*, 24 (10), 1981, pp. 632-646.

[Chao92] C. Chao, R. English, D. Jacobson, A. Stepanov, J. Wilkes, "Mime: A High-Performance Parallel Storage Device with Strong Recovery Guarantees", Hewlett-Packard Laboratories Report, HPL-CSP-92-9 rev 1, November 1992.

[Chutani92] S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, R. Sidebotham, "The Episode File System", *Winter USENIX Proceedings*, January 1992, pp. 43-60.

[Gaede81] S. Gaede, "Tools for Research in Computer Workload Characterization", *Experimental Computer Performance and Evaluation*, 1981, ed. by D. Ferrari and M. Spadoni.

[Gaede82] S. Gaede, "A Scaling Technique for Comparing Interactive System Capacities", *13th International Conference on Management and Performance Evaluation of Computer Systems*, 1982, pp. 62-67.

[Ganger93] G. Ganger, Y. Patt, "The Process-Flow Model: Examining I/O Performance from the System's Point of View", *ACM SIGMETRICS Proceedings*, May 1993, pp. 86-97.

[Gingell87] R. Gingell, J. Moran, W. Shannon, "Virtual Memory Architecture in SunOS", *Summer USENIX Proceedings*, June 1987, pp. 81-94.

[Hagmann87] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit", *ACM SOSP Proceedings*, 1987, pp. 155-162, published by ACM as *Operating Systems Review*, 21 (5), November 1987.

[HP92] Hewlett-Packard Company, "HP C2244/45/46/47 3.5-inch SCSI-2 Disk Drive Technical Reference Manual", Edition 3, September 1992, Part No. 5960-8346.

[Howard88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, M. West, "Scale and Performance in a Distributed File System", *IEEE Transactions on Computer Systems*, 6 (1), February 1988, pp. 51-81.

[Journal92] NCR Corporation, "Journaling File System Administrator Guide, Release 2.00", NCR Document D1-2724-A, April 1992.

[Kondoff88] A. Kondoff, "The MPE XL Data Management System: Exploiting the HP Precision Architecture for HP's Next Generation Commercial Computer Systems", *IEEE Compcon Proceedings*, 1988, pp. 152-155.

[McKusick84] M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, August 1984, pp. 181-197.

[McKusick90] M. McKusick, M. Karels, K. Bostic, "A pageable memory based filesystem", *UKUUG Summer Conference*, pub. United Kingdom UNIX systems User Group, Buntingford, Herts., July 1990, pp. 9-13.

[McVoy91] L. McVoy, S. Kleiman, "Extent-like Performance from a UNIX File System", *Winter USENIX Proceedings*, January 1991, pp. 1-11.

[Moran87] J. Moran, "SunOS Virtual Memory Implementation", *EUUG Conference Proceedings*, Spring 1988, pp. 285-300.

[Ohta90] M. Ohta, H. Tezuka, "A fast /tmp file system by delay mount option", *Summer USENIX Proceedings*, June 1990, pp. 145-150.

[Ousterhout90] J. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware?", *Summer USENIX Proceedings*, June 1990, pp. 247-256.

[Ritchie78] D. Ritchie, K. Thompson, "The UNIX Time-Sharing System", *Bell System Technical Journal*, 57 (6), July/August 1978, pp. 1905-1930.

[Ruemmler93] C. Ruemmler, J. Wilkes, "UNIX Disk Access Patterns", *Winter USENIX Proceedings*, January 1993, pp. 405-420.

[SCSI93] "Small Computer System Interface-2", ANSI X3T9.2, Draft Revision 10k, March 17, 1993.

[Seltzer93] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, "An Implementation of a Log-Structured File System for UNIX", *Winter USENIX Proceedings*, January 1993, pp. 201-220.

[Ston87] M. Stonebraker, "The Design of the POSTGRES Storage System", *Very Large DataBase Conference*, September 1987, pp. 289-300.

A Soft updates implementation details

This appendix provides some low-level details about our soft updates implementation. It assumes the information in section 4.2 has already been read.

We use a basic structure for storing dependency information, containing two sets of next/previous pointers for internal indexing purposes, a unique identifier, a type (we currently use 11 types), and space for several additional type-specific values (currently 9). We found it useful to have "organizational" dependency structures for inodes and general file blocks (i.e., pages in our system) and to keep these **separate** from the actual file system metadata. This allows the file blocks and in-core inode structures to be re-used without losing or corrupting the dependency information. Whenever a directory block or inode is accessed, we check for an outstanding dependency structure (stored in a hash table) and make certain that any undone updates are reflected in the copy visible to users. Two of the dependency structure types serve this purpose. Three of the type-specific values identify the "owner": the pointer to the virtual file system (VFS) structure, the inode number[9], and the logical block number within the file (or -1 for the inode itself). Each organizational structure also heads two lists of dependency structures: those that are simply waiting for the metadata to be written to disk and those that support undo/redo on portions of the "owner."

Block allocation

When a new block/fragment (direct or indirect) is allocated for a file, the new block pointer should not be written to stable storage until after the block has been initialized. This is the allocation initialization dependency described in the introduction. At the time of allocation, we update the metadata in the normal fashion and allocate[10] an *allocsafe* dependency structure for the new block (for which initialized memory is also allocated) and an *alloc* dependency structure (*allocdirect* or *allocindirect*, as appropriate) for the metadata containing the block pointer (inode or indirect block). The one *allocsafe*-specific value is a pointer to its companion *alloc* structure. There are several *alloc*-specific values: the exact location of the block pointer within the metadata structure, the new value for the pointer, the old value for the pointer (NULL, unless a fragment is being extended), the new and old size of the newly allocated block (necessary for fragment extension), a pointer to the corresponding *allocsafe* structure and a state. The state indicates whether or not the dependency is outstanding and whether or not the in-memory copy is up-to-date (always true for allocation dependencies).

At this point, the implementations for direct and indirect pointers differ; we first describe the support for pointers located in the inode. When the initialized block has been written to disk, the *allocdirect* state is modified accordingly and the corresponding *allocsafe* structure is freed. The next time the inode is written, the *allocdirect* structure will also be freed. If the inode is written **before** the newly allocated block has been initialized, the allocation must first be undone. This is accomplished by replacing the new block pointer with the old and, in the case of fragment extension, reducing the file length appropriately. These values can be changed in the buffer cache block without modifying the *in-core*[11] inode structure. "Redo" operations are only necessary if an in-core structure holding an inode with pending *allocdirect* dependencies is re-used by the file

[9] The inode number uniquely identifies the inode within a file system.

[10] We use a simple, fast management policy for the dependency structures. When more are needed, we allocate a page of kernel memory and break it into a list of structures. Freeing a structure consists of adding it to this list, and allocation simply takes the next structure off the list.

[11] The file system always copies an inode's contents from the buffer cached into an *in-core* (or internal) inode structure before accessing them. So, the inode structure manipulated by the file system is always separate from the corresponding source block for disk writes.

system. When the inode is again brought in-core, the new values replace the old and the inode is marked dirty. If this does not happen within 15 seconds, the inode dependency structure is added to the workitem queue. The service function consists simply of bringing the inode in-core.

For pointers in indirect blocks, we implement things differently. An indirect block can contain many block pointers, making it inefficient to traverse a list of per-pointer structures to undo/redo updates. We associate an *indirdep* dependency structure with each indirect block that has pending allocation dependencies. A block of memory equal in size to the indirect block is allocated with this structure, and initialized with a copy of the "safe" contents. When writing the indirect block to disk, this "safe" block is used as the source. When a newly allocated disk block has been initialized, its *allocsafe* structure is freed, and the corresponding *allocindirect* structure is used to update the "safe" copy and then freed. If there are no remaining dependencies when the indirect block is next written, the *indirdep* structure and the "safe" block are freed. Because indirect blocks generally represent a very small fraction of the cache contents, we force them to stay resident and dirty while they have pending dependencies. This allows us to avoid additional undo/redo embellishments.

Block de-allocation

A de-allocated block should not be re-used before the previous pointer to it has been permanently reset. We achieve this by not freeing the block (i.e., setting the bits in the free map) until the reset block pointer reaches stable storage. When block de-allocation is required, the old block pointer values are placed in a *freeblocks* dependency structure, together with the size of the last fragment, if appropriate. Outstanding *alloc* and *allocsafe* dependencies for de-allocated blocks are freed at this point, since they no longer serve any purpose. The block pointers are then reset in the inode (or indirect block) to release the blocks. After the modified metadata has been written to disk, the *freeblocks* structure is added to the workitem queue. For the special case of extending a fragment by moving the data to a new block, and thus de-allocating the original fragment, we do not consider the inode appropriately "modified" until the *allocdirect* dependency clears. The blocks are freed by the syncer daemon using the same code paths as the original file system. Any dependency structures "owned" by the blocks are considered complete at this point and handled accordingly; this applies only to directory blocks.

Link addition

When a new link is added to a directory block, it should not be written to disk until the pointed-to inode (possibly new) has reached stable storage with its link count[12] incremented. We use an undo/redo approach, as with allocation initialization, to provide this protection. The in-memory copies of the directory block and inode are modified in the normal fashion. In addition, an *addsafe* dependency structure is allocated for the inode and an *add* structure is allocated for the directory block. The one *addsafe*-specific value is a pointer to the new *add* structure. There are several *add*-specific values: the offset of the new directory entry within the block, the pointed-to inode number, a pointer to the *addsafe* structure and a state.

The state serves the same purposes described above for allocation initialization. When the inode has been written, the *addsafe* structure is freed and the state is modified appropriately. When the directory block is next accessed, the *add* structure is freed. If the directory block is written **before** the inode reaches stable storage, the link addition is first undone by replacing the inode number in the directory entry with zero (indicating that the entry is unused). After the directory block write completes, the correct inode number again replaces the zero. Because the directory block's contents are out-of-date, we inhibit all accesses (reads and writes) during the disk write. Also, we do not mark the directory block as dirty immediately after the disk write completes. Rather we allow the system to re-use the cache block (i.e., VM page) if necessary. When the directory block is next accessed, we make certain that all directory entries are up-to-date. If it is not accessed within 15 seconds, its dependency structure is added to the workitem queue. The service function simply accesses the directory block and marks it dirty.

Link removal

When a link is removed, the link count in the inode should not be decremented before the modified directory block reaches the disk. We provide this protection with a deferred approach, like that used for block de-allocation. The directory entry is removed in the normal fashion. If the directory entry has a pending link addition dependency, the *add* and *addsafe* structures are removed and the link removal proceeds unhindered (the add and remove have been serviced with no disk writes!). Otherwise, a *remove* dependency structure is allocated for the directory block. The two *remove*-specific values are the inode number and a pointer to the VFS structure, allowing the previously pointed-to inode to be identified later. When the directory block has been written to disk, the *remove* structure is added to the workitem queue. Its service function consists of decrementing the link count. If the link count becomes zero, the inode is freed using the normal code paths and its blocks are de-allocated as described above.

[12] The link count identifies the number of directory entries pointing to the inode.