A Better Update Policy

Jeffrey C. Mogul
Digital Equipment Corporation Western Research Laboratory

Abstract

Some file systems can delay writing modified data to disk, in order to reduce disk traffic and overhead. Prudence dictates that such delays be bounded, in case the system crashes. We refer to an algorithm used to decide when to write delayed data back to disk as an *update policy*. Traditional UNIX® systems use a *periodic update* policy, writing back all delayed-write data once every 30 seconds. Periodic update is easy to implement but performs quite badly in some cases. This paper describes an *approximate* implementation of an *interval periodic update* policy, in which each individual delayed-write block is written when its age reaches a threshold. Interval periodic update adds little code to the kernel and can perform much better than periodic update. In particular, interval periodic update can avoid the huge variances in read response time caused by using periodic update with a large buffer cache.

1. Introduction

File systems usually cache data and meta-data in a main memory *buffer cache*, in order to improve performance. When a modification is made, the file system may write the new information to stable storage (e.g., disk) immediately, or it may delay the write. This leads to a tradeoff: delaying writes reduces the load on the disk and system overhead, but the data could be lost if the system crashes before the write occurs. In many cases, users can tolerate this vulnerability, and welcome the performance advantages of delayed writes.

UNIX® systems have traditionally supported delayed writes, from the earliest C-language version [13, 14] up through 4.3BSD [6] and its derivatives. In these systems, a modification of a partially-filled data block results in a delayed write, while a modification that fills a block results in an immediate, although asynchronous, write. The ULTRIX™ operating system can be configured to be even more aggressive, delaying all writes of modified data.

Without some bound on the age of a delayed-write block, a system crash could cause loss of arbitrary data. Users would not tolerate this, so the file system does push delayed-write data out to disk, after a while. We use the term *update policy* to describe the algorithm that decides what to write out, and when.

UNIX systems have traditionally used a simple *periodic update* (or ‘‘PU’’) policy: once every 30 seconds, all dirty blocks in the file system’s buffer cache are placed on the output queue for the appropriate disk. Recent analytical and simulation results, presented by Carson and Setia [2], showed that the PU policy actually performs worse in many cases than the write-through (WT) policy (in which all writes are immediate). Their analysis showed that PU causes increased mean response times for read operations; the results presented in this paper show that PU can also increase the variance in read response time.

Because so many systems use the now suspect PU policy, it seemed like a good idea to validate the results of Carson and Setia on actual systems; their analyses and simulations, while careful, had to use certain simplifying assumptions. Carson and Setia suggested that several other update policies would provide better performance, so it also seemed useful to implement one of these and measure its performance.

In this paper, after discussing the theoretical background in some more detail, I describe an implementation of the *interval periodic update* (IPU) policy. I also present the results of some simple measurements of PU and IPU made using an actual implementation, rather than a model. The results appear to bear out the basic conclusions of Carson and Setia. What I found is that although use of delayed writes can improve mean response time, combining delayed writes with periodic update increases the variance in response time, but using interval periodic update both the mean and variance are improved.

2. Theoretical background

In this section, I discuss the previous simulation study, the possible alternative update policies, the choice of an update interval, and how this problem will scale with changes in technology.

2.1. Carson and Setia's results for Periodic Update

The study done by Carson and Setia showed that the periodic update (PU) policy, while easily implemented, can perform quite badly. By dumping all the dirty blocks into the disk queue at once, PU can cause lengthy queueing delays. Latency-sensitive synchronous operations, such as file reads or synchronous writes, are forced to wait behind latency-insensitive asynchronous operations in the queue. If the system were to use a write-through (WT) policy instead, disk write operations would normally be spread out more over time, and the queues would be shorter.

Carson and Setia show that the relative performance of WT and PU, measured in terms of mean read-response time, depends on several parameters:

Read load
: The ratio of read operation arrival rate to the rate that the disk can support. A read load of 1.0 is one that the disk could just barely keep up with, if no writes were done.

Write load
: The ratio of write operation arrival rate to the rate that the disk can support.

Cache hit ratio for writes
: The fraction of write operations that are satisfied by modifying already-dirty blocks in the buffer cache.

They expressed their results in tables showing, for a given read load and write load, what write-hit ratio PU requires in order to match or exceed the performance of WT. (Since WT causes disk writes to occur almost immediately, it cannot benefit from write hits in the cache.)

They found that:

- When the disk is not overloaded, ‘‘the cache must eliminate 80-90 % of all write accesses before the PU policy pays off.’’
- Under heavy loads, the PU policy gets some benefit from write-hits in the cache, and thus reduces the overall disk load. In cases where the WT policy would saturate the disk, the PU policy gives a lower mean read-response time.

Cache hit rates vary, depending on cache size, application, and replacement policy, but we are unlikely to achieve average write-hit rates exceeding 80%. For example, Baker *et al.* [1] traced user-level file access patterns in a distributed system and found that 88% of the bytes written were written sequentially; if the applications in question used traditional buffering strategies, most of these sequential writes would not have hit already-dirty buffer-cache blocks, and so the write-hit rate must have been quite low.

Carson and Setia found an analytical model for the mean response time. Their simulations were used to validate this result, but they apparently did not investigate other statistics besides the mean. As section 4 will show, PU is especially bad for worst-case response time, and for overall variance in response time. It is not hard to construct a situation where PU can lead to worst-case read response times of many seconds.

2.2. Proposed alternative policies

PU performs poorly because it generates long queues at periodic intervals, and subsequent synchronous requests get stuck at the ends of these queues. Perhaps one could improve read-response times by changing the queueing mechanism.

UNIX systems typically maintain a single, unprioritized operation queue for each disk. Suppose that read operations were given priority over asynchronous operations already in the queue. Then, read operations would not ‘‘see’’ a queueing delay caused by the queued delayed writes. Carson and Setia analyzed this *periodic update with read priority* (PURP) policy, and showed that it mostly solves the read-response time problem. I found PURP unsatisfactory, however, for several reasons:

- Modification of the existing disk queue mechanism would require changes to numerous kernel modules, including all disk drivers and many of their clients (file systems, virtual memory systems, etc.)
- Modern disk controllers and drives can queue several operations in their internal buffers. One

would either have to accept the resulting queueing delays, or somehow modify the hardware to support the new queueing mechanism.

- Carson and Setia point out that fixed-priority schemes such as PURP introduce the potential for infinite delays of delayed writes, if the read load is enough to saturate the disk. Peacock [12] reported that adding PURP to System V Release 4 does seem to hurt benchmark performance, although it substantially increases single-file write throughput.

Although implementation of a prioritized queueing scheme should be helpful in general, it is neither a complete solution to the bursty-update problem, nor is it the simplest solution.

Carson and Setia also proposed the *interval periodic update* (IPU) policy, in which each dirty block is written out when its age reaches a threshold. If file modifications are nicely spread out in time, this means that the delayed writes back to the disk will also be spread out. As with PU and PURP, IPU uses the buffer cache to eliminate some disk writes that would be done by WT. Unlike PU and PURP, IPU normally avoids creating large bursts of writes, and so avoids the associated queueing delays. Carson and Setia show that IPU never gives worse mean read response time than WT or PU, although in some situations it may perform worse than PURP.

Anna Hac [3] describes algorithms meant for deciding when to move dirty blocks from the buffer cache to a disk queue. In essence these replace the time-driven update policies with dynamic algorithms, which choose when to schedule disk writes based on the system load and disk queue length. Such adaptive algorithms may perform better than any of the open-loop algorithms described in this paper, but they require more extensive changes to the operating system. I do not have anything useful to say about them, and they merit additional study.

2.3. Choice of update interval

UNIX systems have traditionally used a 30-second interval between writes generated by the PU policy. This means that, ignoring brief queueing delays, no information will be vulnerable to a crash for longer than 30 seconds. (Applications that depend on reliable data storage should arrange to write their data synchronously, using the *fsync*() system call. Many other applications, such as compilers, can afford to use delayed writes because their output can easily be reconstructed, or because if the system crashes during a run, the resulting partial output is useless anyway.)

Under the assumption that modifications occur more or less uniformly over time, the average age of a delayed-write block, when it is written to disk, is 15 seconds.

The IPU policy also has a characteristic time scale, the age at which a dirty buffer is scheduled for writing to the disk. If we set this to 30 seconds, then (again ignoring queueing delays), by definition, no information will be vulnerable to a crash for longer than 30 seconds. Also by definition, the average age of a block, when written to disk, is 30 seconds.

If we choose to set the update delay for IPU the same as the update interval for PU, then both policies expose modified data to exactly the same worst-case vulnerability. Doing so, however, means that the mean age of dirty blocks is roughly twice as it would be with the PU policy. This suggests that IPU should see a higher write-hit ratio, and might avoid a few more disk write operations.

Carson and Setia showed that, as the update interval was increased, the write-hit ratio at which PU began to pay off had to increase as well. We do expect this ratio to increase, but does it increase fast enough? Carson and Setia cite other work suggesting that it might [10] (see also a more recent study [1]). Unfortunately, I know of no actual test of this hypothesis. Still, one might suspect that the increased average lifetime of dirty blocks, when the IPU policy is used, might account for some performance advantage. (Note that the experiments reported in sections 4.1 and 4.2 carefully avoid repeated writes to the same block during an update interval, and so should encounter abnormally low cache hit ratios.)

2.4. Scaling properties

One might ask why, if PU performs so badly, has this not been a problem in practice[1]. The answer is that buffer cache sizes and disk speeds are improving at different rates, which changes the ratio of disk queue length to disk service latency.

4.2BSD and related systems only delay partial-block writes. Since most files are written sequentially, most blocks are filled quickly, and pending delayed writes of partial blocks are turned into asynchronous immediate writes of filled blocks. If the buffer cache is not large enough to hold many entire files, then it makes little sense to delay writes

[1]Some systems have indeed exhibited poor behavior resulting from disk queues containing many asynchronous write requests [12].

of full blocks, since these blocks are unlikely to be referenced again quickly.

Memory chips get larger: this is one of the most reliable laws of recent history. One can quibble over whether the doubling time is 18 months or two years, but main memory sizes do increase (at roughly constant cost) as the years pass, and no other technology trend is quite so steep [5].

This trend means that, if the fraction of main memory used as a buffer cache remains constant, the absolute size of buffer caches is increasing with time. (Many systems, including Mach, Sprite, and recent UNIX implementations, no longer allocate a fixed fraction of main memory for the buffer cache, so it can grow to fill all of memory.) Since mean file sizes do not seem to be increasing as rapidly [1], perhaps as main memories get larger, using delayed writes would increasingly reduce disk traffic because of write-hits in the cache. (Traces do show that a few large files are getting much larger [1], and so caching algorithms should perhaps switch to write-through for any file larger than a certain size.)

Although increasing DRAM densities lead to larger buffer caches and perhaps more use of delayed writes, disk technology trends are less encouraging. Disk access times have improved by perhaps one-third in ten years. Disk densities are increasing more rapidly, doubling every three years [5]. Disk bandwidths tend to scale as the square root of disk density (since density improvements come from both higher signal rates and smaller track spacings), and also benefit from small increases in rotation rate (from 3600 RPM to 5400 RPM), so over the past decade they have improved by perhaps a factor of six.

This means that the time it takes to write the entire buffer cache to disk is growing, in absolute terms. This is the key problem for the PU update policy: the queueing delays caused by its burst of write requests will get worse in the future.

Delayed writes typically are queued in no particular order. If the disk driver does nothing to schedule the write operations more carefully, the rate at which the queue can be drained depends mostly on the disk's average access time. In table 2-1, I show how long it takes to write the entire buffer cache (assuming this is 10% of main memory) for systems typical of 1983 and 1993, and I rashly project current trends 10 years into the future.

If the disk drive system can optimize the order of writes in the queue, in the best case the queue can be drained at full disk bandwidth. (Many UNIX disk drivers do sort requests to avoid seeks [6], and some modern disk drives themselves reorder requests.) In table 2-2, I show how the delays for this process scale over time. These numbers are much better than those in table 2-1, but they are probably unattainable, and in any case they are also getting worse.

Year	**RAM size**	**Buffer size**	**Number of buffers**	**Avg. disk access time**	**Time to write all buffers**
1983	1 MB	512 B	205	35 msec	7.2 sec
1993	64 MB	4 KB	1638	15 msec	26 sec
2003	4 GB	64 KB	6554	6 msec	39 sec

Table 2-1: Scaling for random write of entire buffer cache

Year	**RAM size**	**Buffer cache size**	**Disk Bandwidth**	**Time to write all buffers**
1983	1 MB	103 KB	1 MB/s	0.1 sec
1993	64 MB	6554 KB	5.5 MB/s	1.2 sec
2003	4 GB	410 MB	30 MB/s	13.6 sec

Table 2-2: Scaling for sequential write of entire buffer cache

3. Implementation

In this section, I discuss the implementation of various update policies, including the original UNIX implementation of PU, my approximate implementation of IPU, and Sprite's implementation of a similar policy.

3.1. 4.2BSD implementation of the PU policy

Before describing how I implemented the IPU policy, I will describe the ULTRIX implementation of the PU policy. My code is a simple modification of the ULTRIX implementation.

Every 30 seconds, a daemon process (`/etc/update`) wakes up and does a *sync()* system call. This system call schedules writes for certain file system meta-data (the superblock, for example), and then calls the *bflush()* routine to update delayed writes.

In the original 4.2BSD implementation, *bflush()* traversed a queue containing all of the valid blocks in the buffer cache, and scheduled an immediate write for each delayed-write block. Once a block was written and removed from the queue, the algorithm started again from the beginning of the queue; this is done because the queue could be manipulated by another process while the *bflush()* is waiting for the write to complete. Thus, in the worst case this required time proportional almost to the square of the size of the buffer cache, and as memories grew larger, the `/etc/update` process started to consume a large fraction of the CPU.

Recent versions of ULTRIX solve this problem by keeping a separate list of delayed-write blocks (i.e., blocks that are dirty and have not yet been queued for the disk). The *bflush()* routine simply traverses this list once; it need not examine clean blocks, nor does it have to examine any block more than once.

3.2. Implementation of the IPU policy

Carson and Setia point out that to implement a pure IPU policy would require a somewhat complex timer mechanism. Since the timers in the UNIX kernel are quantized, if one wants to issue write operations in the same order as the blocks are dirtied, then one also needs to maintain a separate ordered queue. The overhead of the queue and timers may not be onerous, but it does complicate the kernel.

But why implement pure IPU? If a practical implementation must use quantized time, why not use a relative coarse grain? If the algorithm that moves delayed-write blocks onto the disk queues runs, say, once per second, then the queues will receive bursts of writes, but the bursts will (on average) be about 3% as large as they would be with the PU policy and a 30-second update interval. There will also be a 1-second quantization error in the maximum vulnerable period for a dirty block, but who cares?

I chose to implement this kind of approximate IPU (or ‘‘AIPU’’). I created an alternative version of the *sync()* system call, *smoothsync()*, that takes as its parameter the age at which a dirty block should be written to disk[2]. The *smoothsync()* system call invokes a modified version of the *bflush()* routine, called *bflush_smooth()*. The main difference is that *bflush_smooth()* only schedules a buffer for writing if it has been dirty for longer than the specified threshold.

I replaced the usual `/etc/update` program, which simply calls *sync()* once every thirty seconds, with one that calls *smoothsync(30)* once a second. This program also forces the file system meta-data to disk once every 30 seconds.

The system must also record the time at which a buffer becomes dirty, using a timestamp field in the header associated with each buffer. I did this by adding a few lines of code to a routine called *brelse()*, which is the only place where a buffer is placed on the delayed-write list. At this point, if the timestamp field is zero, then it is set to the current time; otherwise, it is left alone. The *brelse()* routine is also the only place where buffers are placed on the list of clean buffers; at this point, the timestamp field is set to zero.

Thus, a clean buffer always has a zero timestamp. A dirty buffer always has a timestamp reflecting when it was first dirtied. Further modifications of a dirty block do not update the timestamp; otherwise, a block that was touched more often than once every 30 seconds would never be written to the disk.

ULTRIX already includes a timestamp field, *busy_time*, in the buffer header. This is used only when a buffer is busy (i.e., on a disk operation queue), and so is never used when a buffer is on the delayed-write list. Therefore, it can be ‘‘timeshared’’ between these two uses. Other operating systems, including 4.xBSD, do not have such a timestamp field, and so the implementation of IPU would require its addition. The space overhead is small; using modular arithmetic, a one-byte field would allow maximum ages under 255 seconds.

This modification re-introduces the possibility of N^2 behavior in the worst case (that is, when about half of the buffer cache is due to be written during a single interval). I solved this by using an extra queue. The *bflush_smooth()* routine starts by traversing the delayed-write queue and moving ready-to-write buffers from there onto a pending-write queue; this can be done without blocking, and in time linear in the size of the buffer cache. In the second phase, *bflush_smooth()* writes the blocks on the pending-write queue. It could block during this phase, but because it simply pulls the first entry off of the pending-write queue, the algorithm is linear in the number of ready-to-write blocks, and so is also linear in the size of the buffer cache, even in the worst case.

3.3. Sprite's implementation of an IPU policy

The Sprite operating system [9] implements an approximate IPU policy, although somewhat different from the one I implemented. Sprite keeps track of the first-dirty time for the oldest dirty block of each file. Every five seconds, it scans all the dirty files in its cache, and if a file's oldest dirty block is more than 30 seconds old, all of the file's dirty blocks are written back [4]. Because this policy can in theory cause write-backs of fairly young blocks, it may perform somewhat differently from IPU or

[2]Actually, instead of creating a true system call, I added an *ioctl* request, since this involved writing less code. The net effect should be identical. I added one more *ioctl*, to write file system meta-data to disk; this can be called once every 30 seconds, to preserve existing *sync()* semantics.

AIPU. However, since most files are open for only brief periods [1], and so normally all writes to a file happen more or less simultaneously, the average lifetime of a dirty block should be close to 30 seconds.

4. Results

In this section, I describe some simple measurements comparing my implementation of IPU against the original PU policy. All of these measurements were done using a modified ULTRIX version 4.3 kernel, running on either of two DECstation systems; the hardware is summarized in table 4-1. The SCSI disk drives used apparently do not reorder requests, and the ULTRIX version 4.3 SCSI device driver does not sort requests before issuing them.

In all of the experiments in sections 4.1 and 4.2, two processes ran simultaneously:

- A write-load generator, configured to dirty half the blocks in the buffer cache every 30 seconds, at a rate set so that no block would be touched twice in one minute. This means that none of the writes would hit a dirty block in the cache.
- A read-load generator, which read 10,000 randomly chosen blocks from a large file, measured the read-response time for each block, and generated a histogram of the delays.

On the faster system, the buffer cache held 1228 blocks, and the read-load generator used a 34 Mbyte file. On the slower system, the buffer cache held 614 blocks, and the read-load generator used a 32 Mbyte file. Both the generators used files stored on the same disk.

I varied the system configuration, enabling or disabling the use of delayed writes for full data blocks, and changing the update policy. Six trials were done for each configuration. I did not measure a pure write-through configuration, since I do not expect any modern system to use pure WT, given its known poor behavior.

4.1. Local tests

The first set of tests show how the update policy and use of delayed writes affects response time for reads when the disk is local to the generating host. Figures 4-1 and 4-2 show the results for the faster and slower systems, respectively.

The figures show a point for each trial, plotted with mean read response time on the horizontal axis, and the standard deviation of read response time on the vertical axis. Open squares show results for the default configuration (asynchronous writes, PU policy): moderately high mean response time, but low variance. When I enabled delayed writes without changing the update policy, the mean response time dropped somewhat, but the variance increased tremendously (open circles). I then switched to the AIPU policy (filled circles), which reduced the variance without markedly increasing the mean. One could also use the AIPU policy without delayed writes (filled squares); this results in about the same mean as the default configuration, but slightly less variance.

At first, it seemed strange that the mean read response time is lower for delayed writes, since the write-load generator was constructed to avoid dirty-block cache hits; that is, the total number of write operations should be the same in either case. However, the generator does its writes sequentially, which means that when a group of delayed writes is sent to the disk, they are likely to be directed at nearby disk locations, and many end up in the same cylinder group. This reduces the average number of disk seeks per write (relative to non-delayed writes), and so reduces the load on the disk. Since the read-load generator issues random-access reads as fast as possible, disk seeks are probably the rate-limiting bottleneck, and a reduction in the number of write-related seeks leaves more disk-seek capacity to be used for reads. Also, issuing multiple writes to the same region of the disk may reduce rotational latencies.

For example, figure 4-1 shows that without delayed writes, the system can support about 32 reads/sec., and the write-load generator in this case issues about 20 writes/sec., for a total load of about 52 disk operations/sec. Based on the average access time shown in table 4-1, the disk drive should support about 56 random-access operations/sec, which corresponds closely. With delayed writes, the read rate increases to about 38 reads/sec., which means that the disk should only be doing about 14-17 random writes/sec. This suggests that some fraction of the 20 blocks written are being combined with neighbors.

Figures 4-1 and 4-2 show the standard deviation of the response times, but this hides how truly awful things can be with the PU policy. Figures 4-3 and 4-4 show histograms of response time for the faster and slower systems, respectively. In these histograms, all six trials for each configuration have been combined, and the *x*-axis has been divided into logarithmic buckets.

The danger of combining delayed writes and the PU policy now shows up clearly (open circles on the histograms). Because the PU policy puts all the dirty

Description	**CPU type**	**SPECMark rating**	**Disk type**	**Average access time**	**Bandwidth**
Faster system	DECstation 5000 model 200	18.5	RZ58	18 msec	3.8-5.0 Mbyte/sec
Slower system	DECstation 3100	11.3	RZ57	23 msec	2.2 Mbyte/sec

Table 4-1: Systems used for measurements

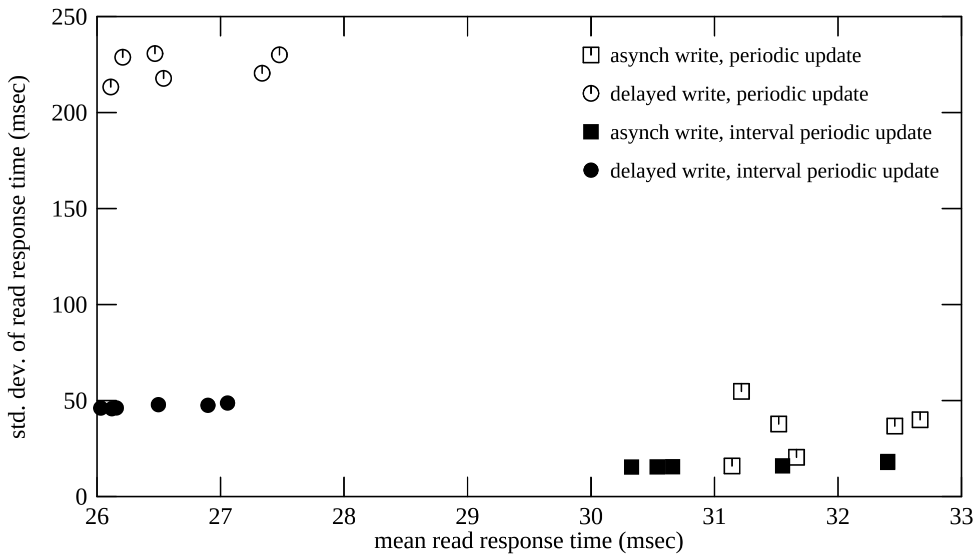

Figure 4-1: Local random reads, fast system

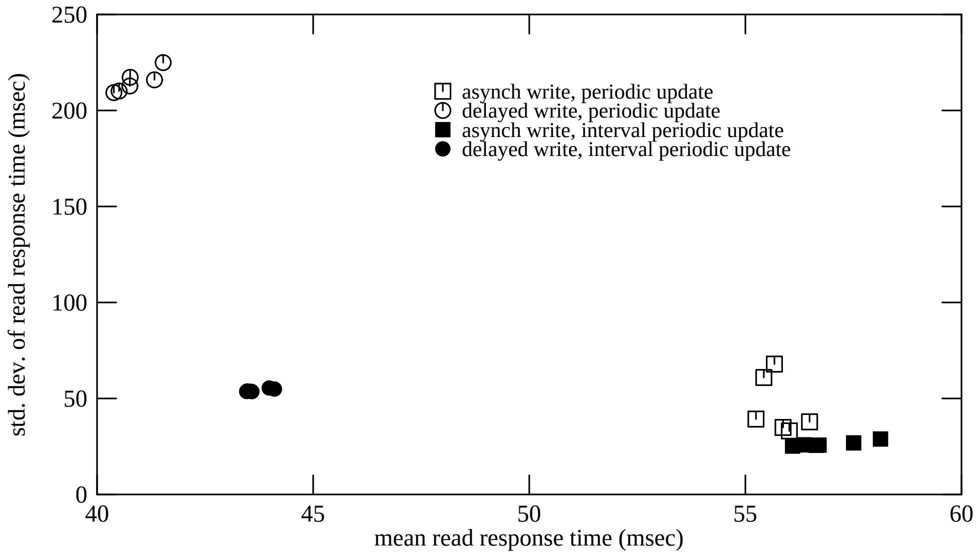

Figure 4-2: Local random reads, slow system

delayed-write blocks on the queue at once, which in these trials could be as high as about 600 blocks, some reads will be delayed by up to 8 or 9 *seconds*. A significant number are delayed by longer than one second.

With the AIPU policy (filled circles), however, only one thirtieth of the dirty blocks show up on the disk queue at any one time. The graph in figure 4-3 shows a small peak in read-response time at about 300 msec. Since we expect 20 blocks (600/30) to be queued once a second, this implies a mean write-delay of about 15 msec, which corresponds closely with the RZ58's specified average access time of 18 msec. More important, this configuration shows a maximum delay of 746 msec, and all but one of the 60,000 samples are below 450 msec. AIPU clearly improves upon PU, in this experiment.

Even when I disabled delayed writes, the PU policy still lead to occasional long delays (up to five or six seconds). In other words, AIPU performs better than PU even if one does not want to abandon the improved safety of using asynchronous writes.

4.2. Remote tests

Most NFS client implementations, to ensure cache consistency and detection of write errors, force delayed writes to the server when a file is closed. This means that NFS clients get little advantage from delayed writes, and do not depend much on the update policy. More recent file service protocols, however, such as Sprite [9], Spritely NFS [15], and NQNFS [7], use explicit cache-consistency protocols and so can benefit from delayed writes.

I ran a set of experiments using Spritely NFS to access a remote disk, using the ‘‘slower’’ system as the client, and the ‘‘faster’’ system as the server. Both the read-load and write-load generators ran on the same client host. The server host supports PrestoServe™ non-volatile RAM (NVRAM). I ran six trials in each of six configurations, as shown in figure 4-5, using random-access reads. I also ran six trials in each of four configurations, as shown in figure 4-6, using sequential reads; in this set of trials, delayed writes were always used. The sequential-read generator cycled through the blocks of a file much larger than the buffer cache on either the client or server, so no reads were satisfied by the caches.

These experiments showed less conclusive results than the local-disk experiments. For random reads (figure 4-5), delayed write combined with PU results in slightly poorer read response time than delayed write with AIPU, although several AIPU trials exhibited much higher variance than any other trials. PrestoServe also seems to be generally beneficial.

For sequential reads (figure 4-6), AIPU seems mostly to reduce the variation between trials. The mean response time is slightly worse for AIPU than for PU without PrestoServe, and slightly better with PrestoServe.

4.3. Kernel-build benchmark

To see if the update policy had any effect on a ‘‘real’’ application, I measured the time it took to compile and link the entire ULTRIX V4.3 kernel, under different combinations of update policy and write policy. These tests were all run on the ‘‘faster’’ system, with all kernel source and object files on the local disk. This process creates about 43 MB of object files, and a similar amount of temporary file data. I ran three trials in each configuration; the means of the results are shown in table 4-2.

The AIPU policy shows a small but clear advantage over the PU policy, especially when using delayed writes. In fact, when using delayed writes with the PU policy, the net elapsed time is actually slightly worse than the normal ULTRIX configuration. The combination of delayed writes and AIPU is about 2.1% faster than the asynch write/PU combination. Note that this benchmark is rather CPU-bound; on these trials, the CPU idle time averaged between 12% and 14%. I would expect AIPU to show a larger benefit on a more I/O-bound application.

The table also shows the number of disk writes charged to the processes involved in the build. The kernel charges a process the first time a block is dirtied; subsequent writes to a dirty block are not counted. We see that while use of delayed writes substantially decreases the write count, by increasing the chance that a write will hit a dirty block, the AIPU policy provides another big decrease, probably by increasing the average lifetime of a dirty block. The combination of delayed writes and AIPU eliminates over 35% of the disk write operations; AIPU by itself accounts for less than half of the improvement.

The table does not show the number of read I/Os charged, since this hardly varies at all with the update policy, and only a few per cent with the use of delayed writes.

4.4. CPU costs of update mechanism

Does the update policy have any effect on the CPU-time cost of doing the updates? The AIPU policy scans the list of dirty blocks 30 times more often than the PU policy does, so one might expect it to consume more CPU time.

I measured the CPU time charged to the `/etc/update` update process during the kernel-build benchmark. This includes both user-mode time (which should be nearly zero) and kernel-mode time (which accounts for all CPU time spent in the *bflush()* routine, as well as other activity). It does not include kernel-mode time spent as a result of disk interrupts. The results are shown in table 4-3; note that the underlying measurements were done with 1-second resolution, and so small variations in the results are not significant.

The table shows that, not surprisingly, aggressive use of delayed writes does increase the CPU time spent in finding delayed-write blocks and scheduling them for disk I/O. Contrary to my expectation, however, AIPU actually reduces the CPU cost of doing updates (although the total cost is in either case insignificant).

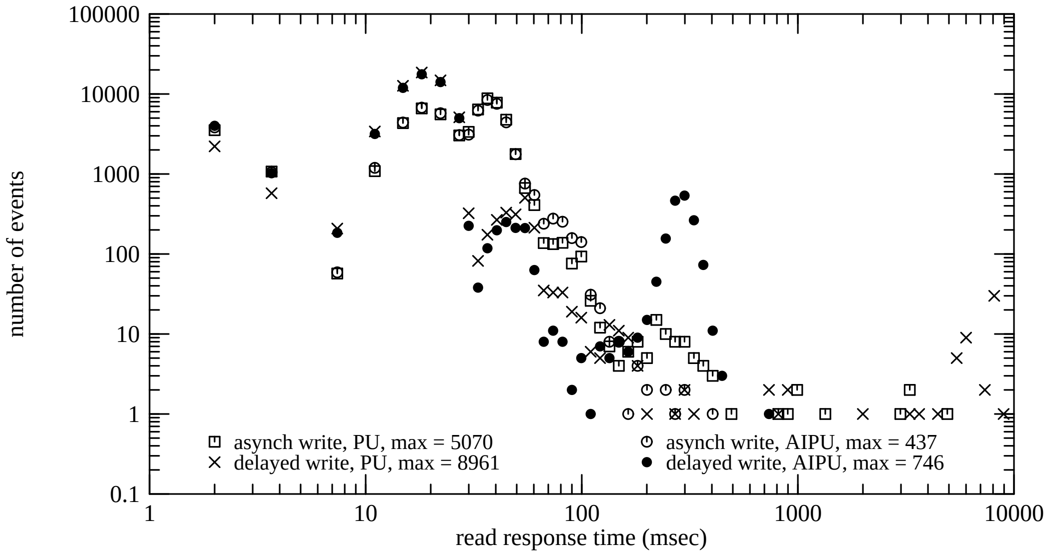

Figure 4-3: Local random reads, fast system (histogram of response times)

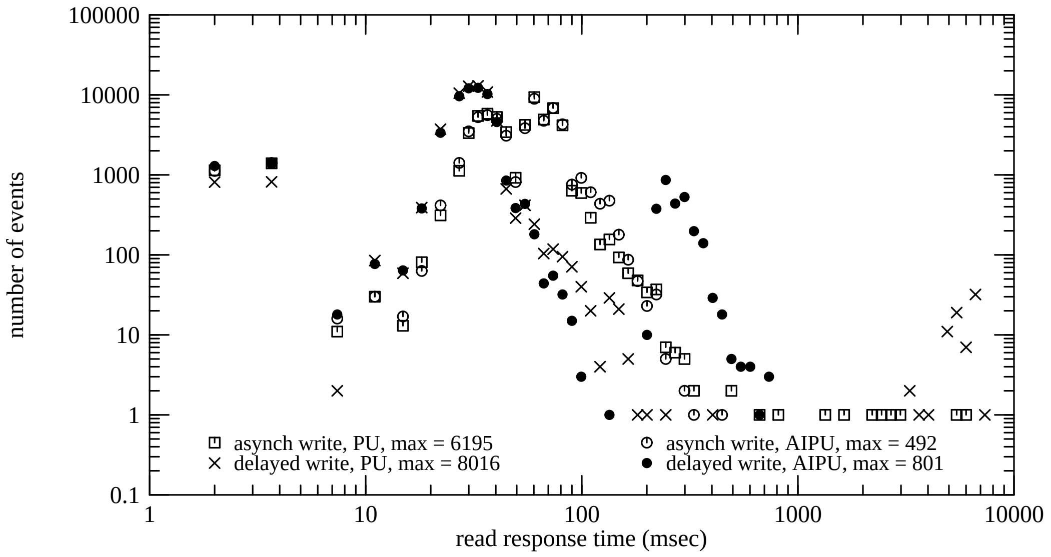

Figure 4-4: Local random reads, slow system (histogram of response times)

I cannot provide a definite explanation, but I suspect that the cause may be the difference in the number of delayed-write blocks actually written to disk. Both algorithms scan every delayed-write block in the buffer cache, and since the AIPU algorithm does this 30 times more often, this suggests that the cost of actually scanning blocks does not dominate the CPU time; the time is probably spent in the device driver[3]. I ran additional trials using AIPU with a 30-second interval between updates (but still using a 30-second age threshold for writing back dirty blocks). This took more CPU time than AIPU with a 1-second interval, but still less than PU.

Table 4-3 shows that AIPU scans far more blocks than PU, because PU scans each block exactly once, but AIPU may scan a dirty block many times before deciding that it is old enough to write to disk. However, AIPU actually writes fewer blocks than PU, because AIPU allows the average dirty block to stay in the cache longer. Recall that with PU, the average age of a block when written to disk is 15 seconds, but with AIPU the average is 30 seconds, assuming a uniform rate of file writes. (AIPU-30, with a 30-second threshold and a 30-second interval between updates, yields an average age of 45

[3]Note that for the kernel-build benchmark, PU with delayed writes scans 16749 blocks in 6.0 CPU seconds, placing a lower bound of 6/16.7 = 0.36 msec per block scanned. This corresponds to several hundred instruction executions, much more than could be accounted for by the scanning loop itself, so most of this time must be spent in the disk driver.

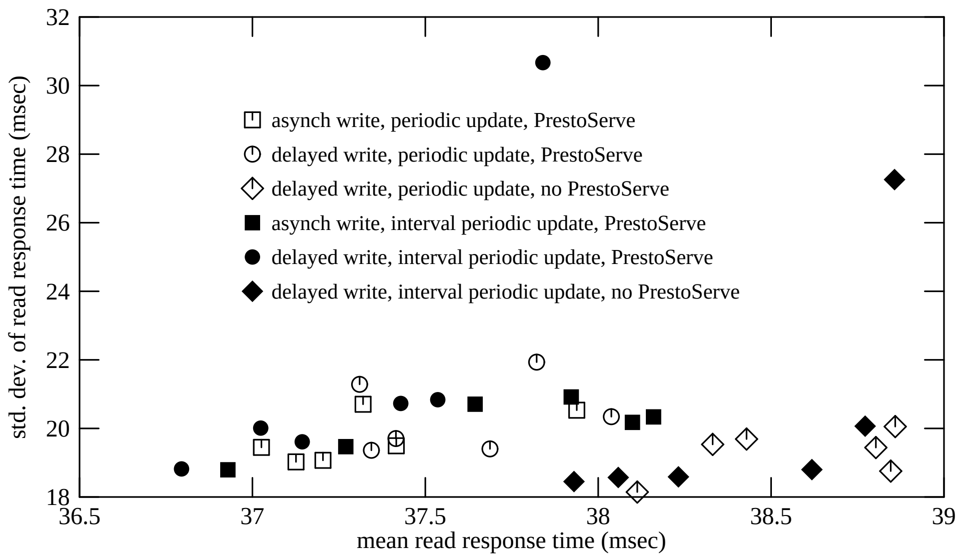

Figure 4-5: Remote random reads

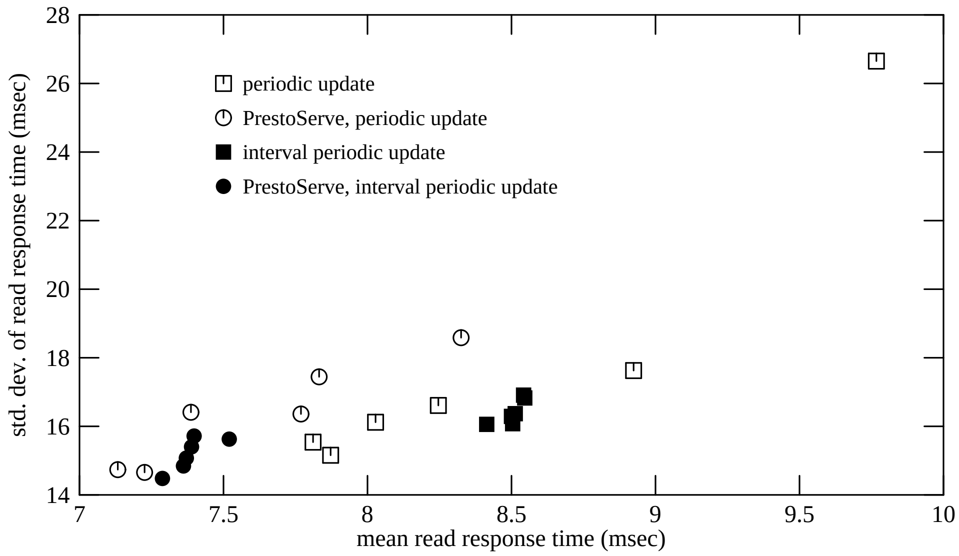

Figure 4-6: Remote sequential reads

	Periodic update	**Approx. interval periodic update**	**Relative elapsed time**
Asynch writes	3280 sec. elapsed 40548 writes	3234 sec. elapsed 37854 writes	0.986
Delayed writes	3298 sec. elapsed 31523 writes	3206 sec. elapsed 25966 writes	0.972

Mean values for 3 trials

Table 4-2: Elapsed time on kernel-build benchmark

seconds, and so does slightly fewer writes than AIPU-1).

A dirty block that stays in the cache for a longer time is more likely to be modified again before being written to disk, which results in fewer modifications of clean blocks, and hence fewer writes to disk. I counted the number of modifications of currently dirty blocks (in the *brelse()* routine); the numbers are shown in table 4-3. AIPU with delayed writes results in moderately more dirty-block modifications (426K vs. 419K for PU); the difference in dirty-block

	PU, async writes	**PU, delayed writes**	**AIPU-1, async writes**	**AIPU-1, delayed writes**	**AIPU-30, async writes**	**AIPU-30, delayed writes**
CPU time used by `/etc/update` (mean of 3 trials)	4.3 sec.	6.0 sec.	2.0 sec.	2.3 sec.	2.7 sec.	2.7 sec.
Blocks scanned by `/etc/update`	9531	16749	290214	508448	14593	25847
Blocks written by `/etc/update`	9531	16749	5375	10023	5266	9519
Dirty blocks modified during benchmark	391K	419K	393K	426K	405K	426K
biowait() sleep events during benchmark	8234	8274	7607	8121	7496	7751

Table 4-3: Statistics for kernel-build benchmark

modifications is roughly the same as the difference in delayed-write disk I/Os.

The reduction in actual disk I/Os, caused by longer cache lifetimes and more dirty-block modifications, may account for all of the elapsed-time advantage of AIPU over PU on the kernel-build benchmark. As table 4-3, with AIPU-1 and especially AIPU-30, the kernel ‘‘sleeps’’ less often for disk I/O than it does with PU (although I could not measure the total sleep time). However, this effect should not contribute to the random-access results in sections 4.1 and 4.2, since these experiments were constructed to avoid any cache hits on file writes.

4.5. Burstiness of file writes

The advantage of AIPU over PU is that the latter clumps together all delayed writes from a 30-second period into a single burst of writes, while the former preserves the original spacing of the file writes (with 1-second resolution). If the file writes themselves arrive in bursts, this eliminates AIPU’s advantage. In the worst case, when all file writes during a 30-second period occur nearly at once, both policies should perform the same.

In the experiments reported in sections 4.1 and 4.2, the write-load generator distributed file writes uniformly over time, which is the best case for AIPU. The kernel-build benchmark should be more representative of real use; how bursty is its file-write pattern? (Note that this kind of single-user benchmark is more likely to exhibit burstiness than a multi-user benchmark, because the latter will tend to spread out the file-system load among several jobs.)

I modified the *bflush()* and *bflush_smooth()* routines to keep a histogram of the number of blocks they queue for the disk on each invocation. I then ran one trial of the kernel-build benchmark for each of PU, AIPU-1, and AIPU-30. In all cases, the update period (age at which a block is queued to the disk) was 30 seconds.

With AIPU-1, since blocks are queued 30 seconds after the corresponding file write, the burstiness in queue-batch size directly mirrors the burstiness in the file-write pattern (with 1-second granularity).

The results are shown in figure 4-7, in the form of cumulative distributions for the number of blocks written as a function of the burst size. Each policy writes a different number of blocks (see table 4-3), so the curves do not end at the same ordinate. For AIPU-1 (with 1 second between updates and a 30-second period), 95% of all delayed-write blocks written were queued by *bflush_smooth()* in bursts of 40 blocks or fewer. The largest burst contained 104 blocks. In other words, the application delivered relatively small bursts of writes to the file system.

For PU (with 30 seconds between updates), 95% of the delayed-write blocks queued were in bursts of *more* than 62 blocks, and 50% were queued in bursts of more than 182 blocks. The largest burst contained 344 blocks, which (assuming an 18 msec. mean access time) delayed any subsequent synchronous operation by over 6 seconds.

To summarize figure 4-7, the kernel-build benchmark does indeed spread out its writes over periods longer than a second. This results in far smaller disk-queue bursts when AIPU-1 is used than when PU is used.

5. Future work

Carson and Setia proposed using a periodic update with read priority (PURP) policy. Although I resisted implementing PURP, because of its greater complexity, in the general case one cannot avoid long disk queues even with IPU or an approximation. For

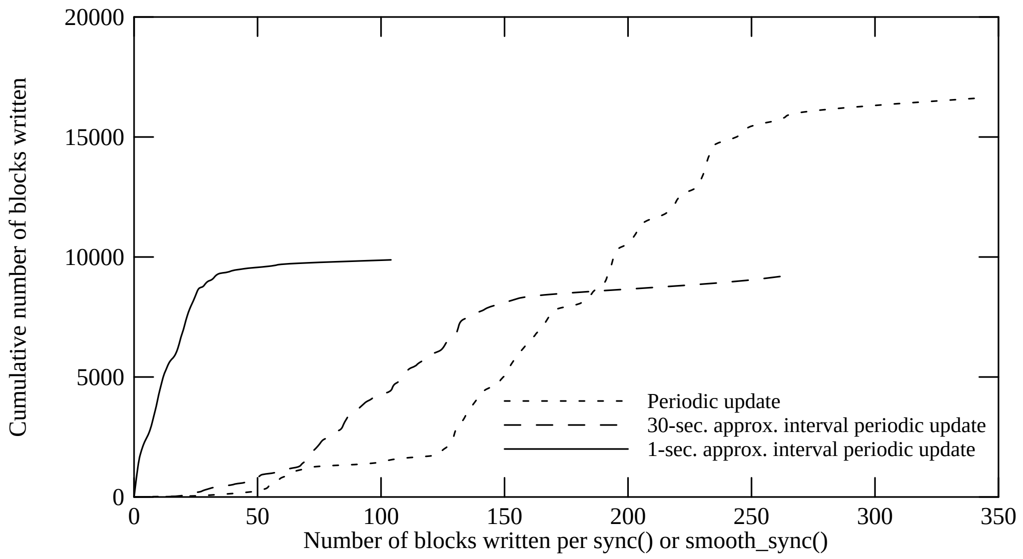

Figure 4-7: Burst-size distributions for writes during kernel-build benchmark

example, if an application manages to dirty the entire buffer cache within a second or so, thirty seconds later an IPU policy will schedule writes for all those blocks, and the effect will be the same as with the PU policy. In other words, IPU depends on a relatively uniform distribution of file writes (across time) to achieve its more uniform distribution of disk writes.

If long disk queues are inevitable, and most of the entries on such queues are inherently asynchronous, then giving priority to reads and synchronous writes should improve response time. I suspect that, even with a priority scheme, an IPU policy could outperform PU. Suppose the buffer cache is entirely filled by dirty blocks; then, a read operation must wait until the system cleans a block before it can complete. The IPU policy generates clean blocks once a second or so (assuming a uniform distribution of disk writes, across time), but the PU policy only does this every 30 seconds. Thus, with PU, reads would be more likely to block waiting for a free buffer. This is speculation; we need experiments, simulation, or more formal analysis to discover the truth.

Peacock [11, 12] has described several systems that use PURP, but apparently did not explore modified update policies. He found that adding read priority to System V Release 4, in which the buffer cache can be quite large, actually reduced benchmark performance by preventing asynchronous requests from getting a sufficient share of the disk.

My experiments have all used a 30-second period for PU and for the lifetime of dirty blocks in IPU, and a one-second granularity for IPU. I suspect that use of different periods and granularities, within reasonable limits, will not make a big difference, but this should be the subject of additional experiments.

Some UNIX file system implementations attempt to cluster several blocks together when performing a disk write [8, 11]. That is, if the cache contains several dirty blocks that are adjacent on disk, the file system or disk driver attempts to write them all at once, which improves throughput by eliminating seeks and rotational delays. Clustering can be done in several different ways, and may interact with the delayed write policy (that is, are all data writes delayed, or only partially-filled blocks?) and with the update policy. The ULTRIX systems tested in section 4 use a clustering algorithm; I have not done experiments to see if this affects the relative performance of update policies.

6. Summary and conclusions

The experiments described in this paper show that

- Use of delayed writes can improve overall file system performance, including read response times, on both synthetic and actual workloads,
- But when delayed writes are combined with the traditional periodic update policy, variance in read response time increases significantly, and benchmark performance may decrease,
- So one should use a better update policy, such as interval periodic update or an approximation, whenever one uses a delayed write policy.

I also showed that one can easily implement an approximate interval periodic update policy, with remarkably limited changes to the kernel of a traditional operating system.

Acknowledgements

Scott Carson and Sanjeev Setia helped to encourage me to perform these experiments. John Ousterhout, Jim Gray, Kent Peacock, and the anonymous reviewers provided useful comments during the preparation of this paper.

References

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proc. 13th Symposium on Operating Systems Principles*, pages 198-212. Pacific Grove, CA, October, 1991.

[2] Scott D. Carson and Sanjeev Setia. Analysis of the Periodic Update Write Policy For Disk Cache. *IEEE Transactions on Software Engineering* 18(1):44-54, January, 1992.

[3] Anna Hac. Design Algorithms for Asynchronous Write Operations in Disk-Buffer-Cache Memory. *J. Systems Software* 16(3):243-253, November, 1991.

[4] John Hartman. Private communication. 1993.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[6] Samuel J. Leffler, Marshall Kirk McCusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, MA, 1989.

[7] Rick Macklem. Not Quite NFS, Soft Cache Consistency for NFS. In *Proc. Winter 1994 USENIX Conference*, pages 261-278. San Francisco, CA, January, 1994.

[8] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Proc. Winter 1991 USENIX Conference*, pages 33-43. Dallas, TX, January, 1991.

[9] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems* 6(1):134-154, February, 1988.

[10] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. 10th Symposium on Operating Systems Principles*, pages 15-24. Orcas Island, WA, December, 1985.

[11] J. Kent Peacock. The Counterpoint Fast File System. In *Proc. Winter 1988 USENIX Conference*, pages 243-249. Dallas, TX, February, 1988.

[12] J. Kent Peacock. File System Multithreading in System V Release 4 MP. In *Proc. Winter 1992 USENIX Conference*, pages 19-29. San Antonio, TX, June, 1992.

[13] Dennis M. Ritchie. Private communication. 1994.

[14] D.M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *AT&T Technical Journal* 57(6):1905-1929, July-August, 1978.

[15] V. Srinivasan and Jeffrey C. Mogul. Spritely NFS: Experiments with Cache-Consistency Protocols. In *Proc. 12th Symposium on Operating Systems Principles*, pages 45-57. Litchfield Park, AZ, December, 1989.

Jeffrey Mogul received an S.B. from the Massachusetts Institute of Technology in 1979, and his M.S. and Ph.D. degrees from Stanford University in 1980 and 1986. Since 1986, he has been a researcher at the Digital Equipment Corporation Western Research Laboratory, working on network and operating systems issues for high-performance computer systems. He is a member of ACM, Sigma Xi, ISOC, and CPSR, the author or co-author of several Internet Standards, an associate editor of *Internetworking: Research and Experience*, and was Program Committee Chair for the Winter 1994 USENIX Technical Conference.

Address for correspondence: Digital Equipment Corporation Western Research Laboratory, 250 University Avenue, Palo Alto, California, 94301 (mogul@wrl.dec.com)