Optimizing Delay in Delayed-Write File Systems

Peter M. Chen

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
pmchen@eecs.umich.edu

Abstract: Delayed writes are used in most file systems to improve performance over write-through while limiting the amount of data lost in a crash. Delayed writes improve performance in three ways: by allowing the file cache to absorb some writes without ever propagating them to disk (write cancellation), by improving the efficiency of disk writes, and by spreading bursts out over time. For each of these benefits, this paper determines the optimal value for the delay interval: the smallest delay that achieves as good (or nearly as good) performance as an infinite delay. The largest value for optimal delay of any of the three benefits is $\frac{fileCacheSize}{diskTransferRate}$. This value is 1-10 seconds for current systems, implying that the delay used today (30 seconds) is much too large; a smaller delay would minimize data loss yet maintain the same performance.

1 Introduction

Most file systems cache data in memory to improve performance [Leffler89]. Reading data from the file cache is straightforward. Writing data to the file cache, however, involves a tradeoff between reliability and performance that depends on when the new data is propagated to disk. If data is written immediately and synchronously to disk (write-through), the system's throughput for writes becomes limited to the disk's transfer rate. To mitigate this effect, many systems propagate the new data to disk at some later time; this is called *delayed writes*. While delayed writes improve performance, they risk losing data on a system crash or power outage. At the extreme, a file cache with an infinite delay is a pure write-back cache; this is suitable primarily for temporary files.

A long delay can lose large amounts of data if the system crashes, while a short delay sacrifices performance. Real systems balance these two factors to try to achieve performance almost as high as a pure write-back file cache while not losing too much data if the system crashes. A common, though somewhat arbitrary, choice for the delay is 30 seconds.[1]

Delayed writes affect performance in several ways.

- The most significant effect is *write cancellation*: the ability to absorb writes in the file cache and never need to write new data to disk. This can happen in a variety of ways. For example, a program such as a compiler might delete data within 30 seconds of writing it, or a user of an editor might write one version of a file, then overwrite the file with a new version after more editing. In general, if the delay is longer than the lifetime of the file, then no write to disk need be done [Baker91]. At its best, write cancellation allows a system to sink new data at memory speeds rather than the disk speeds.
- Second, delayed writes can improve the efficiency of writing data to disk. This can happen in a variety of ways, most of which take advantage of having more data available to write to disk. For example, disk scheduling [Seltzer90], logging [Rosenblum92], and clustering [McVoy91] are all more effective when the amount of data to write increases. Delaying the writes for a longer period of time allows more data to accrue in the file cache.
- Third, delayed writes allow a system to spread a burst of writes out over time. Write-through caches limit the rate at which programs can generate file data to the speed of disk. Delayed writes give the system the ability to perform the writes from a burst during a future idle period. A benefit related to spreading out bursts is the ability to do asynchronous writes. These writes are scheduled to disk immediately

1. Systems that delay data for 30 seconds will sometimes write certain data through to disk. For example, metadata such as directories and inodes is often written synchronously to disk [Ganger94], and some Unix systems schedule an asynchronous disk write as soon as a modification fills an entire block [Mogul94].

but return to the user before the disk has completed the write. This increases the amount of overlap between the processor and the disk system, though asynchronous events can make programming more challenging.

In this paper's model for delayed writes, delay is defined as the time between writing new data and propagating that data to disk. The model assumes that deleting or overwriting data renders that data worthless and hence makes it unnecessary to propagate the deleted or overwritten data to disk. Under this model, data that is touched at least once per delay interval is never written to disk. I believe this is a reasonable design choice, since overwriting a file vastly diminishes the worth of the old values.[2]

The goal of this paper is to provide a basis for choosing delay for a given workload and system. For each reason given above, I determine the *optimal delay*, which I define as *the smallest delay that achieves as high (or nearly as high) performance as an infinite delay*. I show that the optimal delay is less than $\frac{fileCacheSize}{diskTransferRate}$. This value is 1-10 seconds for current systems, implying that the delay used today (30 seconds) is much too large; a smaller delay would minimize data loss yet maintain the same performance.

2 Write Cancellation

In this section, I determine the optimal delay using a simple, steady-state workload. Sections 3 and 4 refine the workload and system to include bursts and the feedback effect of delay on efficiency. In this section, optimal delay is defined as the smallest delay that achieves the same performance *in terms of write cancellation* as an infinite delay.

2.1 Basic Model

Start with the following simple workload: the user is continuously writing a set of files at a fixed throughput. Each file is deleted a fixed lifetime after being written. The amount of live data at any given time (the *file set size*) is thus $workloadThroughput \times fileLifetime$. For the time being, I will assume there are no reads or bursts; these will be addressed in later sections. With this simple workload, it is straightforward to determine the optimal delay according to the following three rules:

Rule 1: If workload throughput is less than the throughput that the disk can sustain, optimal delay is zero.

This rule argues that if the disk can sustain the entire workload, then write cancellation is not needed and the delay can be chosen to minimize data loss. This is the most controversial and pervasive rule in this paper: that a low amount of write traffic should be optimized for reliability, not performance. This assumes that the user's workload is fixed; in other words, the user does not issue a workload that proceeds as rapidly as possible over the long term. Short-term bursts that do proceed as rapidly as possible will be addressed in Section 4. The effect of this assumption on read performance will be addressed later in this section. Readers uncomfortable with this rule can pick a lower value for disk transfer rate; for example, it is easy to accept that writing to disk at 0.01 MB/s is no worse than not writing to disk at all. Section 3 will show that a small delay allows the disk to write at close to the disk transfer rate, so this rule places a lower bound on workload throughput of 1-10 MB/s with current disk technology. A delay of zero refers to an immediately issued, asynchronous write.

Rule 2: If the file set size is greater than the file cache size, optimal delay is zero.

This rule argues that if the workload will thrash the file cache, no write cancellation will occur, even with an infinite delay. Thus a delay of zero achieves the same performance as an infinite delay.

Rule 3: If neither of the first two rules applies, optimal delay is equal to the file lifetime.

2. Some users may require that the disk holds some version of the data even if the data has been overwritten. To satisfy this, the system could write overwritten data to disk periodically, where the period used is much longer than the normal delay (reflecting the diminished value of the old data). The model in this paper assumes that the period used to write old data is infinite.

This delay allows all data to be deleted before they are written to disk, just as with an infinite delay.

Figure 1 illustrates these rules by graphing the optimal delay versus file set size and workload throughput for a fixed file cache size of 100 MB and disk transfer rate of 5 MB/s. Optimal delay is zero over much of the range shown and has a maximum value of only 20 seconds. The maximum value for optimal delay occurs at the largest file set that fits in the file cache and the slowest workload throughput that is greater than the disk can sustain.

Rule #1 argues that the workload throughput must be above a certain threshold for write cancellation to improve performance ($workloadThroughput > diskTransferRate$). Rule #2 argues that the file set size must be below a certain threshold for write cancellation to occur even with an infinite delay ($fileCacheSize > fileSetSize$). Multiplying these two rules together implies that for delay to be non-zero, $workloadThroughput \times fileCacheSize > diskTransferRate \times fileSetSize$, which can be re-written as:

$$fileLifetime = \frac{fileSetSize}{workloadThroughput} < \frac{fileCacheSize}{diskTransferRate} \qquad \text{Equation 1}$$

This constrains the maximum value for the optimal delay in Rule #3 to be less than the time it takes to write the entire file cache to disk.

Knowing either file set size or workload throughput allows a tighter bound on optimal delay than Equation 1. For example, consider the Sprite file system measurements [Baker91], where the file cache size ranged from 0-24 MB with an average of 7 MB. The Sprite study measured a very low steady-state workload throughput of 8 KB/s per active user (including both reads and writes). With this low workload throughput, Rule #1 recommends a delay of zero. However, given the average file cache size of 7 MB, Equation 1 constrains the maximum optimal delay under *any* workload throughput to be no more than $\frac{7MB}{diskTransferRate}$. With a disk transfer rate of 1 MB/s, the maximum optimal delay would range from 0-7 seconds.

Thus only a short delay is needed to achieve the primary benefits of write cancellation. This conclusion differs from prior work such as [Baker92] and [Ousterhout85], which sought to minimize the number of disk writes by lengthening the delay. The key difference is Rule #1, which argues that with a low

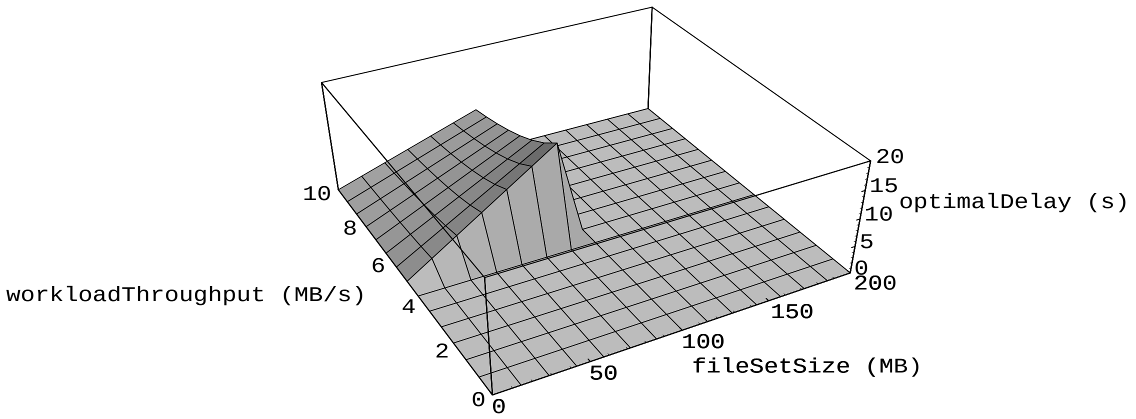

Figure 1: Example of Optimal Delay as a Function of File Set Size and Workload Throughput. The three rules in Section 2.1 constrain the delay that minimizes data loss while still performing as well (in terms of write cancellation) as an infinite delay. This figure graphs the optimal delay for a system with a 100 MB file cache and disk transfer rate of 5 MB/s. For most of the range shown, the optimal delay is zero (fileSetSize > fileCacheSize or workloadThroughput < diskTransferRate). In the range where optimal delay is non-zero, delay is equal to fileSetSize / workloadThroughput. Note that the maximum value for optimal delay is only 20 seconds.

enough workload throughput (such as that used in [Baker92]), there is no benefit gained by lowering the number of disk writes. A moderately high workload throughput, say 2 MB/s, makes it difficult to achieve the high file lifetimes reported in [Baker91] without thrashing the file cache. For example, achieving a 100 second file lifetime while writing at 2 MB/s requires a 200 MB file cache!

2.2 Effect of Reads on Write Cancellation

The above analysis assumes that the user only issues writes. The presence of reads changes Rules #1 and #2. Fortunately, the following analysis shows that these changes effectively cancel each other out.

If the read workload *at disk* is intense, it becomes important to decrease the number of writes to disk even if the write throughput is low; this increases optimal delay. On the other hand, the presence of read traffic implies that the workload's write throughput is less than the total workload throughput; this decreases optimal delay. For example, the write throughput in the Sprite traces was only 1.6 KB/s. Rule #1 then becomes:

Rule 1': If workload *write* throughput is less than the throughput that the disk can sustain *without slowing read traffic at the disk,* optimal delay is zero.

This may decrease the lower-bound on workload throughput used in Section 2.1. However, the disk may be able to sustain a write traffic close to the total disk transfer rate without causing read performance to suffer. First, file caches filter reads quite effectively, so many workloads will not significantly load the disk with read traffic. Without any write cancellation, a relatively small fraction of disk traffic will be reads [Rosenblum92]; thus reads will only decrease moderately the lower bound on write throughput derived from Rule #1.

The second reason that the disk may be able to sustain a high write traffic without hurting read performance is that disk reads can be serviced at a higher priority than disk writes [Carson92]. This makes read performance largely independent of write traffic, as long as the disk can sustain the total traffic in the steady state. Prefetching also decouples read performance independent of write traffic, as long as the disk can sustain the aggregate read and write traffic [Patterson95].

Read traffic can also lower the optimal delay by decreasing the file cache space available for holding dirty data. For example, file caches in the Sprite traces averaged 7 MB, but much of this was likely used to hold clean data. This changes Rule #2 to be

Rule 2': If the file set size is greater than the file cache size *available for dirty data,* optimal delay is zero.

Combining these two changes to Rule #1 and #2, reads change the optimal delay to be either zero or $\frac{fileCacheSizeAvailableForWrites}{diskTransferRateAvailableForWrites}$. The following analysis shows that this ratio is less than $\frac{fileCacheSize}{diskTransferRate}$, implying that optimal delay in the presence of reads is even smaller than the optimal delay with no read traffic.

Consider the system shown in Figure 2. The file cache size available for writes can be estimated as $\frac{write_{fileCache}}{write_{fileCache} + read_{fileCache}} \times fileCacheSize$. This assumes that the amount of space used by each component of a workload is proportional to the intensity of the component. The disk transfer rate available for writes is $diskTransferRate - read_{disk}$. We can then rewrite $\frac{fileCacheSizeAvailableForWrites}{diskTransferRateAvailableForWrites}$ as:

$$delay = \frac{fileCacheSizeAvailableForWrites}{diskTransferRateAvailableForWrites} = \frac{fileCacheSize \times \frac{write_{fileCache}}{write_{fileCache} + read_{fileCache}}}{diskTransferRate - read_{disk}}$$

Equation 2

The numerator of Equation 2 shows that a higher fraction of write traffic at the file cache (relative to reads) increases delay. To bound delay, the next step is to express the maximum value for write traffic in terms of read traffic. Note that write$_{disk}$ with an infinite delay must be less than $diskTransferRate - read_{disk}$, otherwise the system cannot sustain the workload under any delay interval. This constraint can be written as

$$write_{disk}(infiniteDelay) < diskTransferRate - read_{disk} \qquad \text{Equation 3}$$

If we estimate the write miss ratio with an infinite delay to be the same as the read miss ratio, that is, $\frac{write_{disk}(infiniteDelay)}{write_{fileCache}} = \frac{read_{disk}}{read_{fileCache}}$, Equation 3 can be written as

$$write_{fileCache} < \frac{read_{fileCache}}{read_{disk}} \times (diskTransferRate - read_{disk}) \qquad \text{Equation 4}$$

Equation 4 expresses the maximum amount of write traffic into file cache, and this yields the largest value for delay in Equation 2. Plugging this maximum value for write$_{fileCache}$ into Equation 2 reduces Equation 2 to $\frac{fileCacheSize}{diskTransferRate}$. Hence the largest value for optimal delay in the presence of reads is no larger than the optimal delay without read traffic.

This result is consistent with intuition: reads lower the file cache size available for storing data by the same amount as they lower the disk transfer rate available for writes. For example, workloads with more reads will likely use more file cache space for storing read data than workloads with more writes.

3 Increasing the Efficiency of Writes

The second way that delayed writes can improve performance is by increasing the efficiency of writing to disk. A variety of optimizations depend on delayed writes, such as disk scheduling, clustering, and logging. Most of these optimizations take advantage of the extra data accrued during the delay to make writes more efficient. Longer delays allow more data to accrue, and this improves efficiency, where efficiency is defined as the effective disk throughput divided by the disk transfer rate. The efficiency of writing to disk affects the lower bound on workload throughput used in Rule #1 and throughout the paper.

The relationship between efficiency and delay depends on workload throughput, data locality, and disk characteristics, as well as the optimizations used to improve efficiency. Efficiency can be calculated as

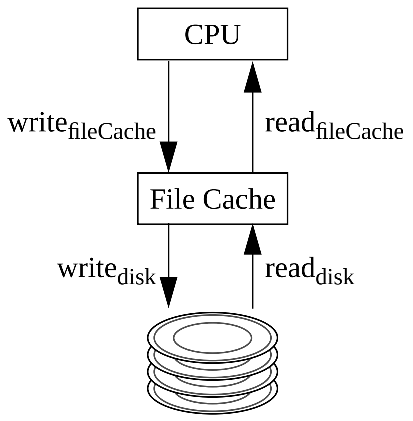

Figure 2: Analyzing the Effect of Reads on Optimal Delay. This figure presents the terminology used in analyzing the effect of reads on optimal delay. The label on each arrow indicates the amount of read or write traffic between each component.

follows for systems that use logging (or some similar optimization such as clustering) to collect data into a single write [Rosenblum92, Seltzer95]:

$$effectiveDiskThroughput = \frac{size}{diskPositioningTime + \frac{size}{diskTransferRate}} \qquad \text{Equation 5}$$

$$efficiency = \frac{effectiveDiskThroughput}{diskTransferRate} = \frac{size}{size + diskPositioningTime \times diskTransferRate}$$

Equation 6

For example, a DEC RZ26L disk can sustain a data transfer rate between 2.7 and 5.5 MB/s (assume an average of 4 MB/s) and has a positioning time (average seek + average rotation) of 15 ms. With these characteristics, collecting 60 KB before writing achieves 50% efficiency, 540 KB achieves 90% efficiency, and 6 MB achieves 99% efficiency. Because workload throughput must be greater than the disk throughput (Rule #1), this amount of data accrues quickly.[3] Within two seconds, enough data is collected to achieve 99% efficiency. This result allows us to assume throughout the paper that the disk can sustain traffic close to its transfer rate (minus the portion of the transfer rate needed for reads, as discussed in Section 2.2).

Several things may increase the delay needed to achieve high efficiency. Systems that only use less powerful optimizations such as disk scheduling will require more data and a correspondingly longer delay to achieve high efficiency. Also, a high level of read traffic at the disk decreases the amount of write throughput the disk can sustain, which decreases the lower bound on workload throughput and increases the delay needed.

4 Spreading out Bursts

The third way that delayed writes can improve performance is by allowing the system to spread bursts of user activity over a longer period of time. My analysis assumes that a burst instantaneously fills the file cache with a given amount of dirty data. An instantaneous burst represents a worst-case scenario for bounding delay, because it requires the longest time to write the last byte of the burst to disk.

To determine optimal delay, note that the largest burst that the file cache can store has burst size equal to the file cache size. This takes $\frac{fileCacheSize}{diskTransferRate}$ time units to drain to disk, so delay equal to this time is as good as an infinite delay. Taking read traffic into account follows the same analysis as in Section 2.2.

For bursty workloads, the system must take into account the time it takes to write the burst to disk when calculating when to start writing to disk. For example, if the burst size is 10 MB, the maximum delay is 15 seconds, and the disk transfer rate is 5 MB/s, the system should start writing to disk after $15 - \frac{10}{5} = 13$ seconds.

5 Effect of Technology Trends

The rules developed above depend on system parameters, and it is interesting to examine how current technology trends affect optimal delay.

As CPU speeds continue to increase, workload throughput will also tend to increase, though perhaps not quite as rapidly as CPU speeds [Baker91]. This lowers the optimal delay for the first two effects. The

3. By setting workload throughput to be equal to disk throughput, and substituting (workload throughput * delay) for size, Equation 5 can be solved for the minimum delay needed for the disk to sustain a given workload throughput: $delay = \frac{diskPositioningTime}{1 - \frac{workloadThroughput}{diskTransferRate}}$

file lifetime ($\frac{fileSetSize}{workloadThroughput}$) of any workload that fits in the cache will decrease, which lowers the optimal delay for write cancellation. In addition, the increased data traffic into the file cache lowers the delay needed to make writes efficient.

As memories get cheaper, file cache sizes will continue to grow. This increases the optimal delay for write cancellation and spreading out bursts, since the file cache will be able to store larger file set sizes and larger bursts.

As disk transfer rates continue to rise, the lower bound on workload throughput will also rise. This decreases optimal delay for the same reasons as CPUs getting faster, and it also decreases the time needed to write the largest possible burst to disk.

[Mogul94] combines current technology trends for memory sizes and disk transfer rates and estimates that $\frac{fileCacheSize}{diskTransferRate}$ was approximately 1 second in 1993 and will increase to 14 seconds in 2003, implying that the 30 second delay used today is much too long and will stay too long for another decade.

6 Experimental Verification

The main goal of this paper is to derive some simple rules for determining optimal delay, and hence a thorough experimental performance study is beyond the scope of this paper. To help ensure that the analysis did not overlook any important effects, however, I modified Digital Unix V3.0 running on a DEC AXP 3000/600 to conform to my model of delayed writes and verified the rules on a few simple workloads. The disk used was a 1.0 GB DEC RZ26L, with an average seek time of 9.5 ms, an average rotational latency of 5.6 ms (5400 RPM), and a transfer rate of 2.7-5.5 MB/s. The tested system had 128 MB of memory, and the file cache can grow as large as 80-90 MB.

The standard version of Digital Unix uses *periodic update*, where sync runs every 30 seconds and writes out all dirty data to disk. This policy guarantees that dirty data stays in the file cache at most 30 seconds; however the average age of a block when it is written to disk is only 15 seconds. I implement *interval periodic update* in a manner similar to [Mogul94]. This policy writes dirty data out to disk approximately *delay* seconds after it has last been modified. This is accomplished by modifying sync to only write data to disk if it is older than the delay interval, and calling sync once a second to check for data that has crossed this threshold. I implemented interval periodic update only for data; metadata is written *asynchronously* whenever sync is called (once per second).

I start by verifying the rules regarding write cancellation. Running the Andrew benchmark [Howard88, Ousterhout90] serves to verify Rule #1, because the disk can easily sustain the write traffic generated by Andrew. The Andrew working set also fits easily in the file cache [Chen94], hence most of the disk bandwidth is available for disk writes. Andrew's running time is approximately the same for both infinite delay (16.3 seconds) and zero delay (16.7 seconds). Note that zero delay may allow dirty data to remain in the file cache for up to one second and is not equivalent to synchronous write-through. Performance with a synchronous, write-through file system is much worse[4] than with zero delay for several reasons. First, a synchronous, write-through system would write to disk much more often than once per second, and this drastically lowers disk efficiency. Second, writes would be synchronous, and this prevents the overlap of processing and I/O.

To verify Rule #2 of write cancellation, I run a synthetic workload that simultaneously reads a 120 MB file and writes a different 120 MB file. Both reads and writes are done sequentially in 1 MB units. Since the workload thrashes the file cache, no extra write cancellations occur by delaying writes to disk. Hence the total throughput achieved with infinite delay (3.1 MB/s) is about the same as the total throughput achieved with zero delay (3.0 MB/s).

To verify Rule #3 of write cancellation, I run a synthetic workload composed of two simultaneous processes. One process reads a 120 MB file from disk as fast as possible; the other process repeatedly

4. Using a synchronous, write-through file system increases the running time of Andrew to over 300 seconds [Chen96].

writes a 60 MB file into the file cache at a target throughput of 10 MB/s. Lifetime for the written file is fixed at 6 seconds. As before, both reads and writes are done sequentially in 1 MB units. The resulting graph (Figure 3) has two distinct regions of performance. When delay is less than 7 seconds, sync writes data to disk. This limits write performance to less than the target throughput of 10 MB/s, and it also hinders read performance. When delay is at least 7 seconds, no writes to disk occur. Note that increasing delay to infinity achieves no better performance than a delay of 7 seconds.

Figure 4 shows how write efficiency improves as the amount of data to be written increases. I measure write throughput by writing a specific amount of data to a single file, then measuring the time fsync takes to write the data to disk. No other syncs are done in the system while this test is taking place. As predicted in Section 3, writing a few MB to disk at a time achieves nearly optimal performance. For example, writing

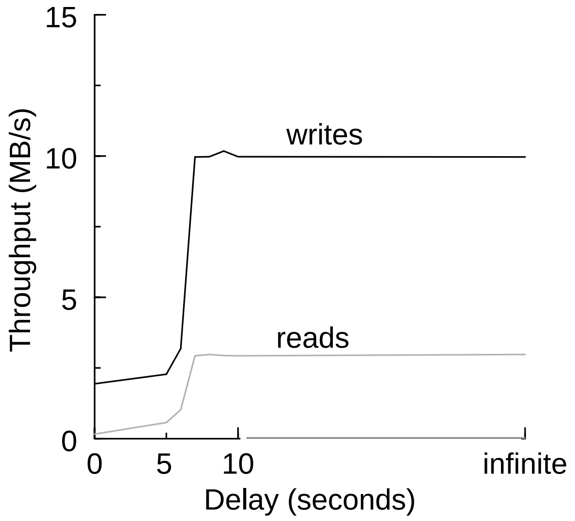

Figure 3: Verifying Rule #3 of Write Cancellation. This figure shows that any delay greater than the file lifetime achieves the same performance as an infinite delay. Two processes ran simultaneously: one reads a 120 MB file from disk as fast as possible; the other writes a 60 MB file to disk repeatedly at a target throughput of 10 MB/s. This yields a file lifetime of 6 seconds. When delay is less than the file lifetime, the file cache must write data back to disk. This limits writes to disk speeds and hinders read performance.

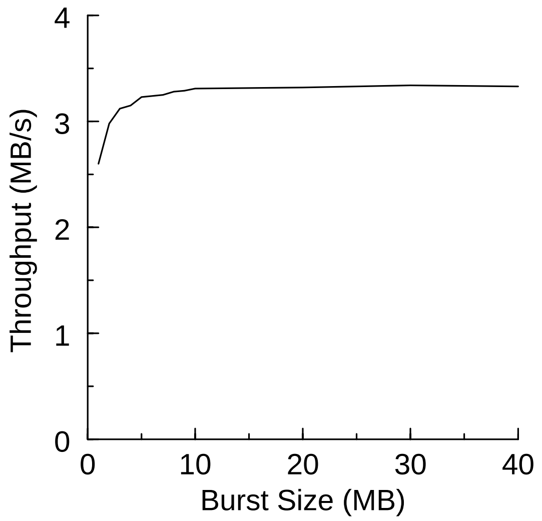

Figure 4: Improving Write Efficiency. As predicted in Section 3, writing a few MB to disk at a time is sufficient to achieve nearly optimal efficiency.

5 MB to disk achieves 97% of the maximum performance. The disk can sustain 3 MB/s, so a 2-second delay allows enough data to be collected to achieve nearly very high efficiency.

Figure 5 measures the time required to drain bursts of different sizes to disk, using the same workload used to measure write efficiency. The file cache can absorb at most 80-90 MB; bursts longer than that force some data to be written to disk during the burst. Hence the largest burst that needs to be drained to disk at the time of the call to fsync is 80-90 MB, and this takes about 30 seconds.

7 Related Work

Much past work has been done on using delayed writes to improve file system performance, though I know of none that has determined the optimal delay interval for a given system and workload. One of the earliest papers on the topic examines the file cache miss ratios for a zero-second (write-through), 30-second, 5-minute, and infinite (pure write back) delay [Ousterhout85]. This is closely related to file lifetimes (more recent measurements appear in [Baker91]). Focusing on miss ratio ignores Rule #1, which argues that misses do not matter if write throughputs are below a certain threshold. In fact, the throughputs in [Ousterhout85] are well below any reasonable threshold: total workload throughput for both reads and writes is only a few hundred bytes/second. [Baker91] finds more intense workloads (8 KB/second), but these are still easily sustainable by a disk. My analysis suggests a delay of zero (asynchronous writes) for these light workloads, or at most $\frac{fileCacheSize}{diskTransferRate}$ for any arbitrary workload on the system.

Two recent papers form the groundwork for my analysis. Carson and Setia compare a delay of zero (write-through) with the standard, 30-second delay (periodic update) [Carson92]. Their main metric is the average response time seen by reads. They find that, under light load ($workloadThroughput < diskTransferRate$), the large batch of updates generated by periodic update hurts read response time; this favors write-through and is consistent with Rule #1. Periodic update performs better than write-through only if there is a large amount of write cancellation or a large gain in efficiency. Carson and Setia propose two ways to prevent periodic update from hurting read response time, interval periodic update and periodic update with read priority. My analysis uses interval periodic update, which writes data out after a certain delay; this spreads writes to disk more evenly in time than periodic update. Periodic update with read priority allows reads to bypass the write queue to disk; this allows the read response time to be largely independent of the write traffic, as long as the disk is not saturated.

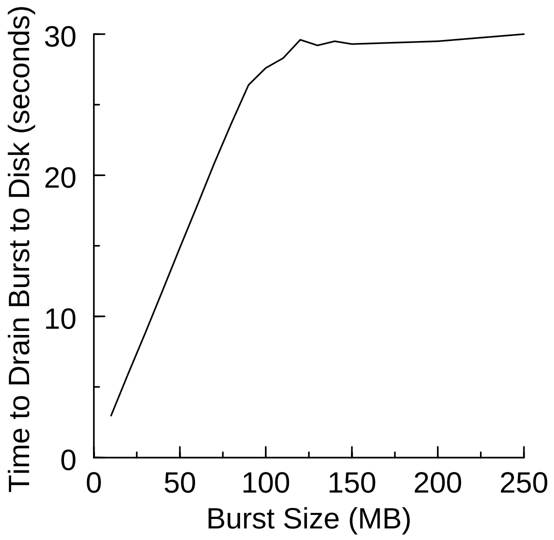

Figure 5: Measuring the Time Required to Drain to Disk Bursts of Different Sizes. The workload generates a burst of a specific size, then measures the time required to fsync that burst. The file cache can absorb at most 80-90 MB; bursts longer than that force some data to be written to disk during the burst.

Mogul compares read response time in Ultrix under periodic update, interval periodic update, and write-through [Mogul94]. His workload is carefully tailored to avoid write cancellation, thus his work focuses on the efficiency gains made possible by delayed writes. He finds that delayed writes can lower read response time by making writes more efficient, although the large batch of writes introduced by periodic update increases the variance in read response time. The efficiency gained by delaying writes was fairly small (10-20%) because the system used did not re-order, log, or cluster disk requests.

My analysis extends the work of Carson, Setia, and Mogul. I use the interval periodic update proposed and implemented in [Carson92, Mogul94] and determine the optimal delay based on the characteristics of the workload and system.

Many researchers are investigating the use of non-volatile RAM to improve file system performance and reliability [Baker92, Wu94, Akyurek95, Chen96]. This paper argues that the delay used in delayed-write file systems should be much smaller than the current 30 seconds. However, a delay of a few seconds is still inferior to the synchronous, write-through semantics guaranteed by non-volatile RAM. This is because the permanence guaranteed by any delayed-write file system is asynchronous and depends on time; in contrast, a synchronous, write-through file system can guarantee permanence after a particular step in a program (for example, after each write or fsync). Lowering delay thus shrinks but can not eliminate the window of data vulnerability.

8 Summary and Recommendations

This paper has analyzed the three main performance benefits of delayed writes: cancelling writes, making disk writes more efficient, and spreading out bursts of writes. For each, I have derived the optimal delay, which is the shortest delay that achieves as high (or nearly as high) performance as an infinite delay. For all three benefits, the optimal delay is zero if the disk can sustain the workload's write traffic without penalizing read traffic. This places a lower bound on workload throughput throughout the analysis.

For write cancellation, the optimal delay is zero if the file cache is smaller than the file set size. If the file cache can hold the workload's file set, then the optimal delay is the file lifetime, that is $\frac{fileSetSize}{workloadThroughput}$. This limits delay to be less than $\frac{fileCacheSize}{diskTransferRate}$, because $fileCacheSize > fileSetSize$ and $diskTransferRate < workloadThroughput$. The presence of read traffic was shown to not increase optimal delay.

For improving the efficiency of disk writes, the optimal delay depends on data layout, locality, and scheduling. A simple model for disk efficiency for a logging file system shows that a relatively small delay (1-2 seconds) provides enough data to make writes 99% as efficient as an infinite delay, assuming the workload is writing data quickly enough to require delayed writes.

For spreading out bursts of writes, the optimal delay is equal to the time it takes the largest burst to drain to disk, or $\frac{fileCacheSize}{diskTransferRate}$.

For all benefits, the maximum possible value for optimal delay is $\frac{fileCacheSize}{diskTransferRate}$. [Mogul94] estimates $\frac{fileCacheSize}{diskTransferRate}$ at 1 second in 1993 and 14 seconds in 2003, implying that the 30 second delay used today is much too long. I therefore recommend that systems use a lower delay to minimize data loss without losing performance.

9 References

[Akyurek95] Sedat Akyurek and Kenneth Salem. Management of partially safe buffers. *IEEE Transactions on Computers*, 44(3):394–407, March 1995.

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, October 1991.

[Baker92] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-Volatile Memory for Fast Reliable File Systems. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 10–22, October 1992.

[Carson92] 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.

[Chen94] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance (full version). *ACM Transactions on Computer Systems*, 12(4):308–339, November 1994.

[Chen96] Peter M. Chen, Wee Teck Ng, Gurushankar Rajamani, and Christopher M. Aycock. The Rio File Cache: Surviving Operating System Crashes. Technical Report CSE-TR-286-96, University of Michigan, March 1996.

[Ganger94] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. *1994 Operating Systems Design and Implementation (OSDI)*, November 1994.

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[Leffler89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley Publishing Company, 1989.

[McVoy91] L. McVoy and S. Kleiman. Extent-like Performance from a Unix File System. *Winter Usenix 1991*, pages 33–44, January 1991.

[Mogul94] Jeff Mogul. A Better Update Policy. In *Proceedings of the Summer 1994 USENIX Conference*, pages 99–111, June 1994.

[Ousterhout85] John K. Ousterhout, Herve Da Costa, et al. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 1985 Symposium on Operating System Principles*, pages 15–24, December 1985.

[Ousterhout90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings USENIX Summer Conference*, pages 247–256, June 1990.

[Patterson95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 79–95, December 1995.

[Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[Seltzer90] Margo I. Seltzer, Peter M. Chen, and John K. Ousterhout. Disk Scheduling Revisited. In *Proceedings of the Winter 1990 USENIX Technical Conference*, pages 313–324, January 1990.

[Seltzer95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, and Venkata Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. In *Proceedings of the Winter 1995 USENIX Conference*, pages 249–264, January 1995.

[Wu94] Michael Wu and Willy Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System.

In *Proceedings of the 1994 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1994.