12-1991

Caching and Writeback Policies in Parallel File Systems

David Kotz
Dartmouth College

Carla Schlatter Ellis
Duke University

Caching and Writeback Policies in Parallel File Systems*

David Kotz

Dept. of Math and Computer Science
Dartmouth College
Hanover, NH 03755-3551
David.Kotz@Dartmouth.edu

Carla Schlatter Ellis

Dept. of Computer Science
Duke University
Durham, NC 27706
carla@cs.duke.edu

Abstract

Improvements in the processing speed of multiprocessors are outpacing improvements in the speed of disk hardware. Parallel disk I/O subsystems have been proposed as one way to close the gap between processor and disk speeds. Such parallel disk systems require parallel file system software to avoid performance-limiting bottlenecks. We discuss cache management techniques that can be used in a parallel file system implementation. We examine several writeback policies, and give results of experiments that test their performance.

1 Introduction

As computers grow more powerful, it becomes increasingly difficult to provide sufficient I/O bandwidth to keep them running at full speed for large problems, which may consume immense amounts of data. Disk I/O has always been slower than processing speed, and recent trends have shown that improvements in the speed of disk hardware are not keeping up with the increasing raw speed of processors. This widening access-time gap is known as the I/O crisis [13, 16]. The problem is compounded in typical parallel architectures that multiply the processing and memory capacity without balancing the I/O capabilities.

The most promising solution to the I/O crisis is to extend parallelism into the I/O subsystem. One such approach is to connect many disks to the computer in parallel, spreading individual files across all disks. Parallel disks could provide a significant boost in performance — possibly equal to the degree of parallelism, if there are no significant bottlenecks in the I/O subsystem and if the I/O requests generated by applications can be mapped into lower-level operations that drive the available parallelism. Thus, the first challenge to the designers of a multiprocessor file system is to configure parallel disk hardware to avoid bottlenecks (e.g., shared busses), and to avoid further bottlenecks in the system software. An effective file system for a multiprocessor must itself be fully parallel to scale with additional processors or disks. The second challenge is to make this extensive disk hardware bandwidth easily available to application programs. To meet these challenges we propose a highly parallel file system implementation that incorporates caching and prefetching as a means of delivering the benefits of a parallel I/O architecture to the user programs.

This paper concentrates on multiprocessor file systems intended for scientific applications. These applications typically push the leading edge of computing technology, such as multiprocessors, placing tremendous demands on both CPU and I/O systems. Most file caching studies have examined general-purpose workloads (e.g., [16]), where files are much smaller [12, 5]. The parallel environment and workload raise a number of questions: Are caches useful for parallel scientific applications using parallel file systems? If so, in what way? What are the appropriate management policies?

Different workload characteristics, including a new form of locality, lead us to new policies. The sequential access patterns in the workload suggest prefetching and write-behind. Prefetching is the focus of [9, 10, 8], and write-behind is the focus of this paper. What policies are most appropriate for buffering writes for these parallel scientific-application workloads? Do write-behind and delayed writeback help? In what way? This paper examines these issues, defines some new policies, and reports results from experiments with these policies.

In the next section we provide more background information on parallel I/O, caching, and file system workloads. In Section 3 we describe the testbed, the workload, the experimental methods, and the cache management policies. In Section 4 we present the experiments, performance measures, and results. Section 5 concludes.

2 Background

Much of the previous work in I/O hardware parallelism involves disk striping. In this technique, data of a file are interleaved across numerous disks and accessed synchronously in parallel [15, 7, 13]. These schemes rely on a single controller to manage all of the disks.

For multiprocessors, one form of parallel disk architecture is based on the notion of *parallel, independent disks*, using multiple conventional disk devices

*This research was supported in part by NSF grants CCR-8721781 and CCR-8821809 and DARPA/NASA subcontract of NCC2-560.

addressed independently and attached to separate processors. The files may be interleaved over the disks, but the multiple controllers and independent access to the disks make this technique different from disk striping. Examples of this architecture include the Concurrent File System [14, 6] for the Intel iPSC/2 multiprocessor, and the Bridge [4, 3] file system for the BBN Butterfly parallel computer.

File caching is a technique used in most modern file systems. Caching has not been studied for parallel file systems, but Alan Smith has extensively studied disk caching in uniprocessors with general-purpose workloads. In [16], his simulations show that disk caching is an effective way to boost the performance (as measured by the cache miss ratio) of the I/O subsystem (e.g., an 8 MByte cache can service 80–90% of I/O requests).

File access patterns have never been studied for parallel computers, but have been studied extensively for uniprocessors [5, 12]. These studies found that sequential access, usually of the entire file, is the major form of access. Supercomputer file access patterns (a scientific workload) involve huge files (tens to thousands of megabytes) accessed primarily sequentially, sometimes repeatedly [11]. Parallel file access has been discussed by Crockett [2], but he did not study an actual workload.

3 Models and Methods

3.1 Architectural Models

Our architectural model is a multiple instruction stream, multiple data stream (MIMD) multiprocessor. A subset of the problems and many of our proposed solutions (although not our implementation) may also apply to message-based distributed-memory architectures.

We represent the disk subsystem with parallel, independent disks. We assume an interleaved mapping of files to disks, with blocks of the file allocated round-robin to all disks in the system. The file system handles the mapping transparently, managing the disks and all requests for I/O. There is a file system manager running on each processor. This spreads the I/O overhead over all processors and allows the use of all processors for computation, rather than reserving a set of processors exclusively for I/O.

3.2 Workload Model

Parallel file systems and the applications that use them are not sufficiently mature for us to know what forms might be typical. Parallel applications may use patterns that are more complex than those used by uniprocessor versions of the same application. The lack of a real parallel workload employing parallel I/O leads us to use a synthetic workload in our tests, which captures such nuances of real workloads as sequentiality, regularity, and inter-process interactions.

We work with file access patterns, rather than disk access patterns. That is, we examine the pattern of access to *logical* blocks of the file rather than *physical* blocks on the disk. Thus, we make no assumptions of disk layout. Note also that the application is accessing *records* in the file, which are translated into accesses to logical file *blocks* by the interface to the file system. The file system internals, which are responsible for caching, see only the block access pattern.

In our research we do not investigate read/write file access patterns, because most files are opened for either reading or writing, with few files updated [5, 12]. We expect this to be especially true for the large files used in scientific applications. Thus we consider read-only patterns, used to demonstrate the benefits of caching, and write-only patterns, used to investigate delayed-write policies.

All sequential patterns consist of a sequence of accesses to sequential *portions*. A portion is some number of contiguous blocks in the file. Note that the whole file may be considered one large portion. The accesses to this portion may be sequential when viewed from a *local* perspective, in which a single process accesses successive blocks of the portion. We call these *locally sequential access patterns,* or just local patterns. This is the traditional notion of sequential access used in uniprocessor file systems.

Alternatively, the pattern of accesses may only look sequential from a *global* perspective, in which many processes share access to the portion, reading disjoint records within the portion. We call these *globally sequential access patterns,* or just global patterns. If the reference strings of all the processes are merged with respect to time, the accesses follow a (roughly) sequential pattern. The pattern may not be strictly sequential due to the slight variations in the global ordering.

We use eight representative read-only parallel file access patterns. Four of these are local patterns, three are global patterns, and one is random. The sequential nature of the patterns imply a low rate of data rereferencing, which is important for caching. The details of the sequentiality are only important for prefetching.

lw Local Whole file: every process reads the entire file from beginning to end. It is a special case of a local sequential pattern with a single portion.

lfp Local Fixed-length Portions: each process reads many sequential portions. The sequential portions have regular size, although at different places in the file for each process.

lrp Local Random Portions: like **lfp**, but using portions of irregular (random) size. Portions may overlap by coincidence.

seg Segmented: the file is divided into a set of non-overlapping contiguous segments, one per process.

gw Global Whole file: the entire file is read from beginning to end. The processors read distinct records from the file in a self-scheduled order, so that globally the entire file is read exactly once.

gfp Global Fixed-length Portions: (analogous to **lfp**) processors cooperate to read fixed-size sequential portions.

grp Global Random Portions: (analogous to **lrp**) processors cooperate to read random-size sequential portions.

rnd Random: records are accessed at random.

We use three representative write-only parallel file access patterns. Two of these are local patterns and one is a global pattern.

lw1 A single process writes the entire file from start to finish. The other processes are idle.

seg The file is divided into disjoint segments, one per process, and each process writes its segment from start to finish.

gw Like its read-only counterpart, this pattern writes records of the file in a self-scheduled order.

Note that these patterns are not necessarily representative of the *distribution* of the access patterns actually used by applications. We feel that this set covers the *range* of patterns likely to be used by scientific applications.

3.3 Methods

Our methodology is experimental, using a mix of implementation and simulation. We implemented a file system testbed called RAPID-Transit ("Read-Ahead for Parallel Independent Disks") on a BBN GP1000 Butterfly parallel processor [1], an MIMD machine. Since the multiprocessor does not have parallel disks, they are simulated. The testbed is a heavily parameterized parallel program, incorporating the synthetic workload (the application), the file system (interface and manager), and the set of simulated disks. The file system allocates and manages a buffer cache to hold disk blocks, described below. The testbed gathers statistics on many aspects of the performance of the file system. This implementation of the policies on a real parallel processor, combined with real-time execution and measurement, allows us to directly include the effects of memory contention, synchronization overhead, inter-process dependencies, and other overhead, as they are caused by our workload under various management policies. This method allows us to evaluate whether *practical* caching policies can be implemented. See [8] for more details.

In this section we describe one simple replacement policy, which determines the blocks to replace when a free buffer is needed, and several write policies, which determine when new data are written back to disk.

Buffer Replacement Policy: We associate an instance of the cache with a particular open file, caching the logical blocks of the file rather than the physical blocks of the disk. This is a shared cache concurrently servicing the requests of all processes within a parallel application.

The workload plays a significant role in determining the appropriate cache policies. Scientific applications often read and write several megabytes or gigabytes of data, generally sequentially [11]. For a cache to succeed, the workload must exhibit some locality. *Temporal locality*, where recently used data will be used again soon, is not present when large files (much larger than the cache size) are accessed sequentially, even if the files are accessed repeatedly. *Spatial locality*, where other data near or in a recently accessed block will be accessed soon, is a strong component of sequential access patterns. The combination of these observations leads to a "toss-immediately" replacement policy, where only the most recently used (MRU) block remains in the cache. This is more appropriate than the traditional LRU policy [17] (although of course it is identical to LRU with a stack size of one).

In the access patterns we expect to see in parallel scientific applications, another form of locality occurs. With *interprocess locality*, a block used by one process is used soon by another process (when, for example, each is reading different small records from the same block).

We extend the toss-immediately strategy to parallel access patterns as follows: any block that is not the MRU block of any process may be replaced. Thus the cache must have at least as many buffers as processes. Our policy has many advantages. It ensures that the MRU block of each process remains in the cache until that process has clearly finished with it. This is important, because locality makes it likely that the process will use its MRU block again. If there were only one global MRU block, toss-immediately would replace some blocks still in use. If there were a global LRU policy, which had a single LRU stack, an active process could use many blocks, artificially aging the blocks of less-active processes and thus forcing them out. Finally, ours is simple to implement: each buffer has a counter in shared memory indicating the number of processes that consider this block to be their MRU block. Thus, interprocess locality is directly included. When the count reaches zero, the block is free for replacement. If the block is dirty (containing data not yet written to disk), the block must be written to disk and the disk write completed before the buffer may be re-used. Buffers that are available for replacement are kept in a global free list.

Write Policies: A cache can improve file-write performance with *write-behind*, where data is written into a buffer, allowing the application to continue while the buffer is written to disk. If the disk write is not initiated immediately, it is termed "delayed writeback," which traditionally has several advantages:

- Some data disappears before it is written to disk (by being overwritten or by removal or truncation of the file containing the data), and thus disk load is reduced. This is not likely in our workload.
- Bursts of write activity can be absorbed by a cache, asynchronously writing the data to disk while the application continues.
- Where there is spatial locality (e.g., when multiple file writes are made to the same block), caching avoids multiple writes to the disk. This is of prime interest when there is also interprocess locality involved.

The *write policy* determines when the "dirty" buffers are "cleaned" (written to disk). If a dirty buffer is written too late, the cache fills with dirty blocks and processes must idle waiting for buffers to be cleaned. If a dirty buffer is written too early, costly mistakes may be made. There are two types of mistakes possible in write-only access patterns: reread and rewrite. If the application writes to a buffer after the buffer has been written to disk, the disk write was a *rewrite* mistake. If the application writes to a block that has already been flushed from the cache, causing the block to be read back from disk, the extra write and read is a *reread* mistake.

A technique that is appropriate for a single-process sequential access pattern is to write a block whenever the process moves on to the next block (or, if you track the file pointer carefully, when the process writes the last byte in the block). This technique assumes sequential access: once a block is written by the process, it will not be rewritten. In a multiprocess application with interprocess locality, however, the actions of any one process do not clearly indicate when a block is complete. From the assumption of sequentiality, however, every byte of the file (and hence of any block in the file) is written exactly once. Thus it is safe to write the block to disk when all bytes of the block have been written. This leads directly to our WriteFull policy below.

We implemented several distinct write policies:

WriteThru, the simplest scheme, forces a disk write on every file write request from the application. This is ideal for blocks accessed only once.

WriteBack delays the disk write until the buffer is needed for another block.

WriteFree issues a disk write when the buffer enters the free list. Thus, it issues a write before the buffer is needed for re-use, but after it is no longer in use by some processor. This is a compromise between WriteThru and WriteBack.

WriteFull issues the disk write when the buffer is "full," defined to be when the number of bytes written to the buffer is exactly equal to the size of the buffer in bytes.

4 Experiments

We first briefly demonstrate the need for a cache, and then examine the capabilities of the four write policies.

4.1 Experimental Parameters

In all of our experiments, we fix most of the parameters and then vary one or two parameters at a time. The parameters described here are the base from which we make other variations. Each combination of parameters represents one test case.

There were 20 processes running on 20 processors. The patterns all accessed 4 MBytes of data, divided up for local patterns as 200 KBytes per process. The cache block size was always 1 KByte, and the record size was usually one block (in one set of tests we experiment with other record sizes). Note that in most patterns this translates to 4000 blocks read from (or written to) the disk, but in **lw** only 200 distinct blocks are read since all processes read the same set of 200 blocks. The cache contained 80 one-block buffers. We also had the capability to turn the cache off, so all requests went to the disk with no cache overhead.

After each record was accessed, delay was added in some tests to simulate computation; this delay was exponentially distributed with a mean of 30 msec. All other tests had no delay after each access, simulating an I/O-intensive process.

The file was interleaved over 20 disks, at the granularity of a single block. Disk requests were queued in the appropriate disk queue. The disk service time was simulated using a *constant* artificial delay of 30 msec, a reasonable approximation of the average access time in current technology for small, inexpensive disk drives of the kind that might be replicated in large numbers on a multiprocessor system.

4.2 Measures

The RAPID-Transit testbed records many statistics intended to measure and interpret performance. The primary performance metric for measuring the performance of an application is the total execution time. This, and all time measures in the testbed, is real time. Total execution time incorporates all forms of overhead (such as memory contention, reread mistakes, *etc.*) and unexpected effects, and thus it is the best measure of overall performance.

A note on the data: Every data point in each plot represents the average of five trials. The *coefficient of variation* (*cv*) is the standard deviation divided by the mean (average). For all experiments in this paper, the *cv* was less than 0.065 (usually much less), meaning that the standard deviation over five trials was less than 6.5% of the mean. In each table and plot we give the maximum *cv* of all data points involved.

The Ideal Execution Time: We compare the experimental execution time to a simple model of the ideal execution time. The total execution time is a combination of the computation time, the I/O time, and overhead. In the ideal situation, there is no overhead, and either all of the I/O is overlapped by computation or all of the computation is overlapped by I/O. Thus, the ideal execution time is simply the maximum of the I/O time and the computation time. This assumes that the workload is evenly divided among the disks and processors and that the disks are perfectly utilized. No real execution of the program can be faster than the ideal execution time. With the base parameter values, both the I/O and the computation times are 6 seconds, and thus the ideal execution time is also 6 seconds. The ideal I/O time for **lw** is shorter, only 0.3 seconds, since it only reads 200 blocks from disk. The ideal computation time for **lw1** with computation (and thus the ideal execution time) is 120 seconds since there is only one processor involved.

4.3 Caching

Using the testbed, we ran all of our access patterns with and without caching. Our point is not to demonstrate the superiority of our particular buffer-replacement policy, but to demonstrate the basic benefit of a cache (from temporal and spatial locality). We also hope to determine the effects of interprocess locality. The cache, when used, contained 80 one-block buffers. There was no computation involved in these access patterns.

The following table shows the results of experiments on our full set of read-only access patterns. With one-block records, there was actually a slight performance degradation due to caching overhead. There was no improvement because most of these patterns did not rereference data in the cache (i.e., there was no temporal locality). Some patterns (**lrp**, **grp** and **rnd**) made some rereferences, but so rarely that they were insignificant. The **lw** pattern had many rereferences (interprocess temporal locality), but execution time did not improve with caching because all processes read the same block almost simultaneously, and used only one disk at a time. Thus interprocess locality was important, but not beneficial here.

The situation changed significantly when the record size was one-quarter block. Except in the **rnd** pattern, each block was referenced four times, once for each quarter-block record in the block. Without a cache, the block was read four times from the disk. With a cache, spatial locality (in the local patterns) and interprocess spatial locality (in the global patterns) was used to avoid wasting disk bandwidth. (Note that the benefits would be larger for smaller record sizes, and significant for all non-integral record sizes.) Because of the interprocess locality in the global access patterns, however, four processes waited for each four-record block to be read from the disk, and thus only one-fourth of all disks were in use at any time. Prefetching can avoid this underutilization; see [9, 8, 10] for further study of read-only patterns and prefetching.

Read-only patterns
Total execution time, in seconds ($cv < 0.038$)

	One-block		Quarter-block	
Pattern	No Cache	Cache	No Cache	Cache
lfp	6.3	6.7	24.6	7.1
lrp	8.3	8.4	41.0	8.6
lw	6.9	7.1	36.5	7.3
seg	6.9	7.1	36.5	7.5
gfp	6.3	6.5	60.4	25.5
grp	6.4	6.7	60.4	25.4
gw	6.3	6.5	60.4	25.4
rnd	10.6	10.7	41.1	40.8

The next table shows the results of experiments on our write-only access patterns. Here we compared the simple WriteBack caching policy with not caching. Section 4.4 compares write policies. Caching was faster in **gw**, since the delayed write allowed some overlap between overhead and I/O. The **lw1** pattern was most improved because, with delayed writes, this one-processor pattern was able to use more than one disk. This is an example of a cache's ability to help applications use parallel disk bandwidth. Experiments with quarter-block records demonstrate the real power of caching: without a cache, all writes to a disk block after the first write had to read the block from the disk, update the block, and write the block back to disk (a reread mistake). With n records per block, a cache reduced the $2n - 1$ disk accesses per block to one per block.

Write-only patterns
Total execution time, in seconds ($cv < 0.015$)

	One-block		Quarter-block	
Pattern	No Cache	Cache	No Cache	Cache
lw1	127.3	16.4	853.1	55.7
seg	6.9	7.2	63.3	7.7
gw	6.3	6.1	103.0	8.7

4.4 Write-Policy Experiments

We designed a set of experiments to evaluate the effectiveness of our write policies across variations in workload and cache size. These experiments seek to answer the following questions: What is the effect of cache size? Is a large cache useful? How do the policies react to the record size? In particular, how do they handle the interprocess locality in **gw**? Which (if any) policy is the most generally successful? Can a smart write-buffering policy help an application to better use the available parallel I/O bandwidth?

Cache-size Variation: In these experiments, the cache size varied from 20 one-block buffers to 200 one-block buffers (1 to 10 blocks per process). The record size was one block, so each block was accessed only once. Note that WriteFull and WriteThru are inherently equivalent in these access patterns, because the buffer is full when it is first written.

In **gw** with computation, shown in Figure 1, WriteBack was clearly slowest, since it delayed the disk write too long. WriteFree is also slower than WriteThru or WriteFull. This is because WriteFree delays the disk write for a full MRU block until the next file system access, which is after the process's compute cycle (without computation, WriteFree is similar to WriteThru and WriteFull). This delay was too long, slowing down overall execution. Note that between 40 and 80 buffers were the maximum useful cache size. Forty buffers corresponds to two buffers per process, which allowed one to be filled while the other is written to disk. The results for **gw** without computation give similar conclusions.

The **lw1** patterns ran more slowly than the **gw** patterns, because one process could not drive all 20 disks at full efficiency (Figure 2). WriteBack was much worse than the other methods, and WriteFree again was slow with computation. Larger caches benefited the **lw1** pattern by allowing more disk parallelism to be used.

The write-only **seg** patterns had a difficult disk access pattern (all processes began on the same disk). A large cache helped to alleviate the resulting disk contention, as seen in Figure 3, since the larger cache allowed processes to continue writing even when some disks were overloaded. In effect, large caches allowed

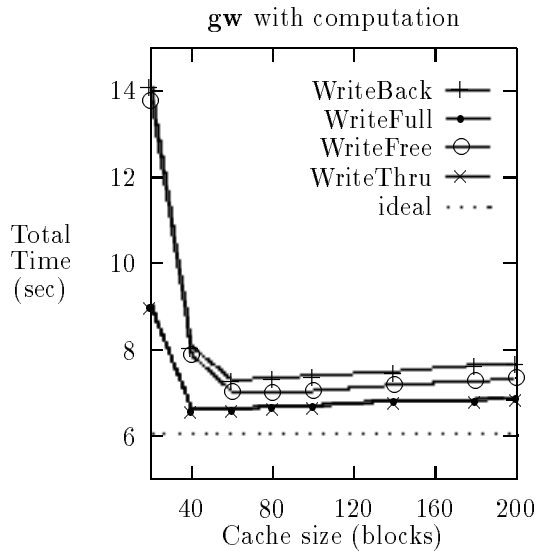

Figure 1: Cache-size variation.

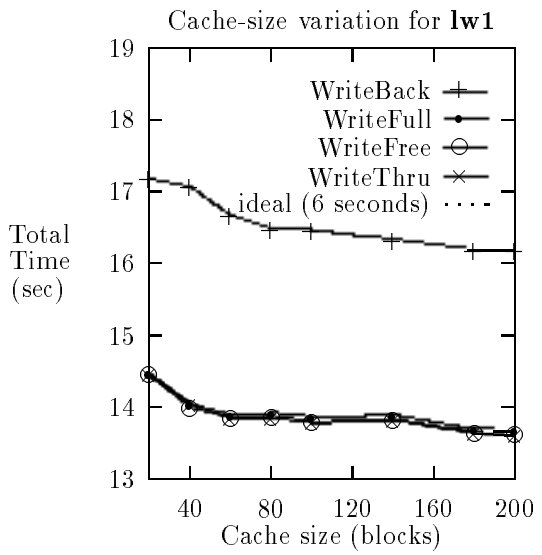

Figure 2: Cache-size variation.

a long pipeline to form, using more disks concurrently than with a short pipeline. This is especially important as processor speeds increase relative to disk speeds. This is an excellent example of the ability of a well-managed cache to help a simple-minded program access the potentially high bandwidth of parallel disks. The results for **seg** with computation are not shown since they offer no new insights.

From these results, both WriteThru and WriteFull (essentially equivalent here) appear to be good write-buffering methods, in that they had the best overall performance. In some cases a large cache was needed to absorb disk contention problems (as in **seg**) or a high write request rate (as in **gw** without computation), but generally two buffers per process were sufficient. For the experiments in the next section we chose an 80-block cache (four buffers per process) because that was a reasonable compromise for all workloads, based on the results in this section.

Record-size Variation: In these experiments we varied the record size of the access pattern with a fixed cache size of 80 one-block buffers. The total amount of data written, in blocks, was fixed. The variation includes both integral and non-integral record sizes (relative to the block size). The latter are important because they cause multiple accesses to many blocks, which should clearly differentiate WriteThru and WriteFull.

Figure 4a shows the record-size variation for the write-only **gw** access pattern. WriteThru is clearly a poor choice for small record sizes, due to a huge number of rewrite mistakes. WriteFree was smarter, waiting until the buffer was mostly unused before issuing a disk write, but it was still not perfect due to some mistakes and to not immediately writing the blocks to disk when they finally were ready to be written. The dips occur because there can be no mistakes with integral record sizes. WriteBack was sometimes faster than WriteFree because it had fewer rewrite mistakes. Finally, the WriteFull method had a nearly perfect 6-second execution time over all record sizes, because it issued the write precisely when the block was ready to go to disk, and made no mistakes.

The results for **lw1** are shown in Figure 4b. The high execution times were due to reduced I/O parallelism, because (due to overhead) one process could not keep 20 disks busy, even with an 80-block cache. With non-integral record sizes this overhead was increased due to repeated accesses to some blocks. Thus, the time varies widely for non-integral record sizes. Otherwise, the results are no surprise: WriteBack was usually slowest, and WriteThru also slow for small non-integral record sizes.

The record-size variation for the **seg** pattern (Figure 4c) shows that WriteThru was slowest, due to rewrite mistakes. Because of the sequential access pattern on each processor, none of the others had rewrite mistakes, and none had reread mistakes.

Thus, record size was an important factor in the performance of our write methods. For integral record sizes, all methods were essentially independent of record size. For non-integral sizes, all but WriteFull made many mistakes. WriteFull was thus the most generally successful write policy.

5 Conclusion

A relatively simple cache management strategy, based on toss-immediately, provided efficient and effective caching for our workload. Most importantly, it was an effective base for studying write policies for

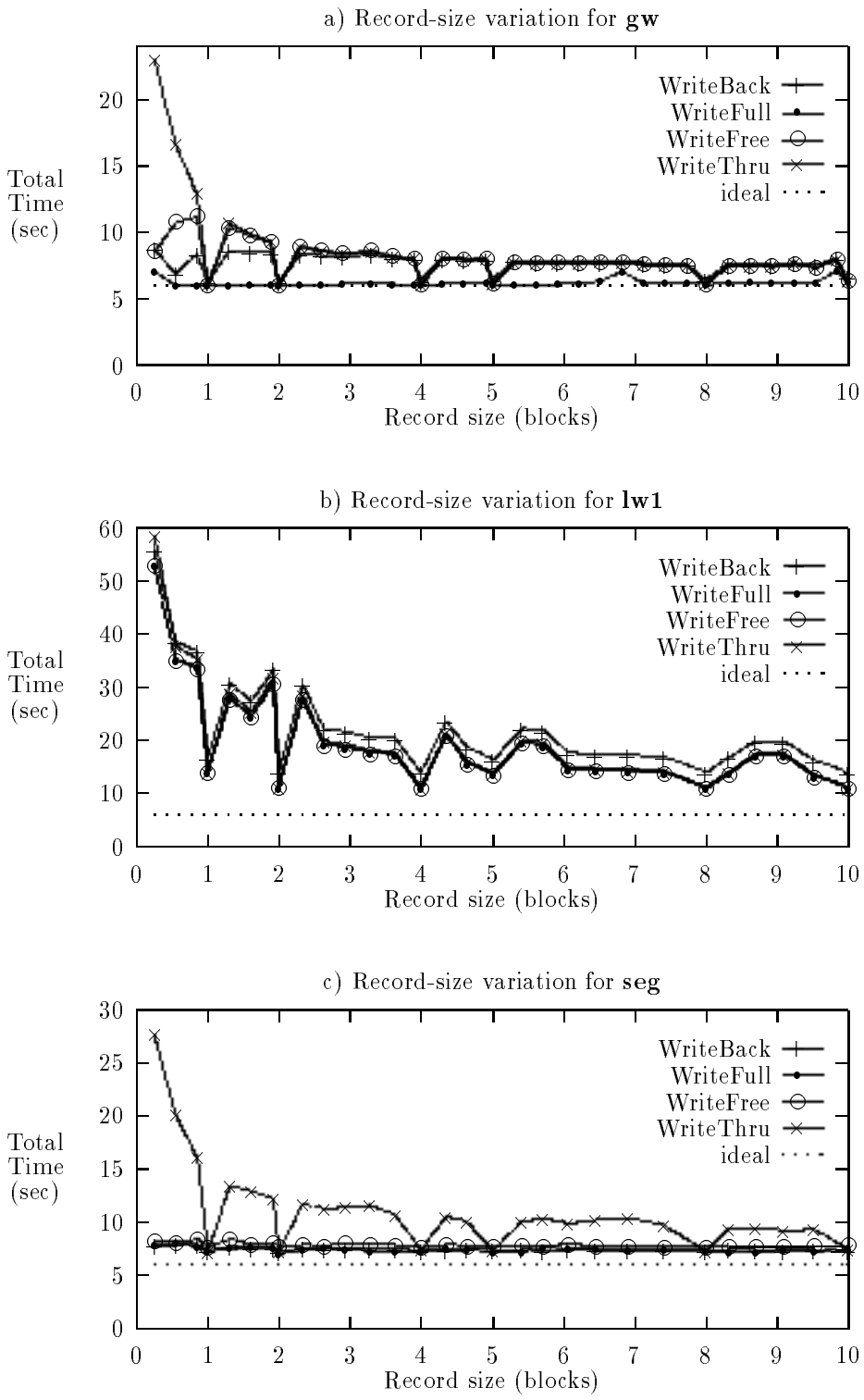

Figure 4: Record-size variation for all three write patterns.

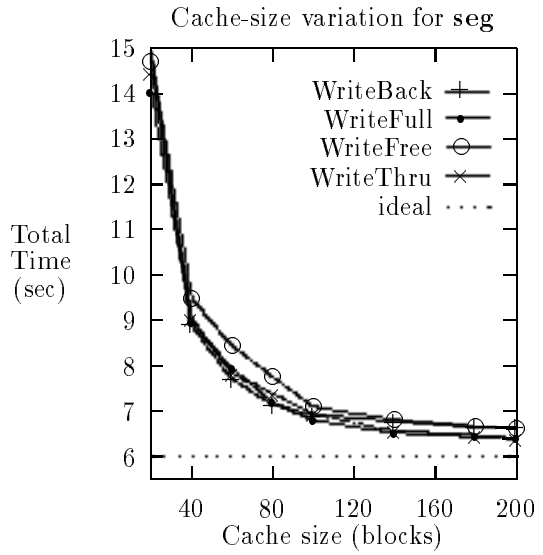

Figure 3: Cache-size variation.

write-only patterns. Caching was often able to use locality, including interprocess locality, to help applications use the parallel disk bandwidth. In applications where caching could not be expected to help, the cache overhead caused a slight (though tolerable) slowdown.

Given the types of write-only access patterns we expect to be common in scientific workloads, our exploration of four methods shows that WriteFull, the most sophisticated of the methods, was consistently at or near the best performance in all situations. A fairly small cache (40–80 blocks, i.e., 2–4 blocks per process) was sufficient to obtain the best performance, except in the **seg** pattern, where larger caches helped mask the disk contention. Large caches were thus only useful when there was high disk contention. (Although we did not study bursty I/O, larger caches should also be useful for absorbing bursts of write activity.)

References

[1] BBN Advanced Computers. *Butterfly Products Overview*, 1987.

[2] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[3] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[4] Peter C. Dibble. *A Parallel Interleaved File System*. PhD thesis, University of Rochester, March 1990.

[5] Rick Floyd. Short-term file reference patterns in a UNIX environment. Technical Report 177, Dept. of Computer Science, Univ. of Rochester, March 1986.

[6] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of a parallel input/output system for the Intel iPSC/2 hypercube. *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 178–187, 1991.

[7] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[8] David Kotz. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors*. PhD thesis, Duke University, April 1991. Available as technical report CS-1991-016.

[9] David Kotz and Carla Schlatter Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.

[10] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for parallel file systems. In *First International Conference on Parallel and Distributed Information Systems*, December 1991. To appear.

[11] Ethan Miller. Input/Output behavior of supercomputing applications. Technical Report UCB/CSD 91/616, University of California, Berkeley, 1991. Submitted to Supercomputing '91.

[12] John Ousterhout, Hervé Da Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson. A trace driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[13] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[14] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[15] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[16] Alan Jay Smith. Disk cache-miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[17] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.