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

July 30, 1992

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

Proposed running head: Kotz and Ellis: Caching in Parallel File Systems

Address for proofs:

David Kotz
Assistant Professor
Mathematics and Computer Science
Dartmouth College
6188 Bradley Hall
Hanover NH 03755-3551
email: David.Kotz@Dartmouth.edu
phone: 603-646-1439

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 for multiprocessors with scientific workloads. We examine several writeback policies, and give results of experiments that test their performance.

Symbols used: $<$ % $*$ @

1 Introduction

It is increasingly difficult to provide sufficient I/O bandwidth to keep parallel supercomputers running at full speed for large problems, which may consume and produce immense amounts of data. Recent trends have shown that improvements in the speed of disk hardware are not keeping up with the increasing raw speed of processors. Parallel I/O mechanisms, such as disk striping [19], could provide a significant boost in performance. The challenge is to make this extensive disk hardware bandwidth easily available to parallel programs. 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. The parallel environment and workload raise a number of questions: Are caches useful for parallel scientific applications using parallel file systems? What are the appropriate management policies? Do write-behind and delayed writeback help utilize parallel disk bandwidth? This paper examines these issues, defines some possible policies, and reports results from experiments with these policies.

In the next section we provide more background information, and then in Section 3 we describe the workload and the cache management policies. In Section 4 we present the experiments, performance measures, and results. Section 5 concludes.

2 Background

There are two ways to attach multiple disks to a multiprocessor. The first is to attach a striped disk array to a processor or to the interconnection network. Disk striping [9, 14, 19] declusters the data of a file across numerous disks, accessing them in parallel through a single controller. The second, which also declusters data over many disks, is to attach independent controllers and disks to separate processors or ports on the interconnection network. We call the latter structure *Parallel Independent Disks (PID)*. Examples of a PID architecture include Intel's Concurrent File System [15, 8, 17], the Bridge simulated file system [6, 5] for the BBN Butterfly, and the file system for the nCUBE/2 [12, 4, 17].

While caching has not been studied for parallel file systems, Alan Smith has extensively studied caching in uniprocessors with general-purpose workloads [20].

Uniprocessor and distributed-system file access patterns have been measured many times [7, 13, 1]. Sequential access is most common. Supercomputer file access patterns (a scientific workload) involve huge files (tens to thousands of megabytes) accessed primarily sequentially, sometimes repeatedly [11]. Five parallel scientific applications, chosen from the PERFECT benchmarks [16] and parallelized for an eight-processor Alliant, have only sequential access patterns. [18]. This is only a small sample, however, and the programs are parallelized sequential programs, not parallel programs *per se.* Crockett's discussion of parallel file access also influences our workload model [3]. In summary, little is known about parallel file access patterns, but it appears that some type of sequentiality will dominate.

3 Models and Policies

3.1 Architectural Models

Our architectural model is a multiple instruction stream, multiple data stream (MIMD) shared-memory multiprocessor 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 manager, running on each processor, handles the mapping transparently, managing the disks and all requests for I/O from that processor.

3.2 Workload Model

The lack of a real parallel workload employing parallel I/O leads us to use a synthetic workload in our tests. We work with *file* access patterns rather than *disk* access patterns, an important distinction. That is, we examine the pattern of access to *logical* blocks of the file rather than to *physical* blocks on the disk. Disk access patterns are complicated by the layout of logical blocks on the disk and by access to multiple files. We concentrate on the logical access pattern per file and ignore disk layout issues.

The application accesses *records* in the file (e.g., lines of text, or rows of a matrix), 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. Blocks are the file system's transfer unit to the disk. We assume that the file system internal buffer size is the same as the block size. Thus, one buffer holds one block.

Most files are opened for either reading or writing, with few files updated [7, 13]. In this paper we focus on write-only patterns, and investigate delayed-write policies. We use three representative write-only parallel file access patterns.

lw1 Local Whole file, one process: a single process writes the entire file from start to finish. Since the other processes are idle **lw1** is a degenerate parallel pattern.

seg Segmented: the file is divided into disjoint segments, one per process, and each process writes its segment from start to finish. Thus, each process produces an independent, sequential access pattern.

gw Global Whole file: the entire file is written from beginning to end. The processes write distinct records to the file in a *self-scheduled* order, so that globally the entire file is written exactly once. Typically, the processes choose the "next" record through some mechanism external to the file system, such as atomically incrementing a shared-memory counter. They independently seek and write a record when they are ready. Thus, the file system may receive requests out of order. The file system does not serialize requests, however, allowing them to proceed concurrently, even if they involve the same buffer.

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. We have seen instances of all three patterns.

3.3 Design

In this section we describe some key components of the design of a file cache, including 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.

3.3.1 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. In our experiments, then, we examined the performance of this one-file cache. Allocating resources among multiple files and applications is left for future work.

A cache depends on locality. *Temporal locality* means that recently used data will be used again soon. *Spatial locality* means that data in or near a recently accessed block will be accessed soon. The combination of these observations often leads to *sequential locality,* wherein the most-recently-used (MRU) block is used repeatedly. In the access patterns we expect to see in parallel scientific applications, we see another form of locality, *interprocess locality.* Here, a block used by one process is used soon by another process.

Sequential locality suggests a "toss-immediately" replacement policy [21], which retains the MRU block in memory, allowing older blocks to be replaced ("tossed").[1] This works for uniprocess sequential access patterns, but a parallel analog is needed. Some access patterns (like **seg**) may have several "MRU" blocks, one for each process. Thus, our parallel toss-immediately scheme retains at least the MRU block for each process. The cache must have at least as many buffers as processes, but one buffer per process seems reasonable compared to the other memory required for a process. This scheme is also simple to implement, using a counter for each cache buffer to count processes whose MRU block is in that buffer. When the counter is zero, the block is replaceable.

3.3.2 Write Policies

A cache can improve file-write performance with *write-behind*, where data are 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 usually reduces the number of disk writes. First, data sometimes disappear before being written to disk (by being overwritten or by removal or truncation of the file containing the data). This is not

[1]Note that this is not, strictly speaking, an "MRU" replacement policy, which *replaces* the most-recently-used block!

likely in our workload. Second, spatial locality combines multiple file writes to the same block into a single disk write, which is of particular interest when there is also interprocess locality involved.

The *write policy* determines when "dirty" buffers are "cleaned" (written to disk). If a dirty buffer is written too late, the cache fills with dirty blocks and processes idle waiting for buffers to be cleaned. If a dirty buffer is written too early, it may have to be written again, wasting the first write. We call this a *rewrite mistake.*

In a single-process sequential access pattern it is reasonable to write a block whenever the process begins writing the next block. This technique assumes sequential access: once a block is fully 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 indicate when a block is complete. From the assumption of sequentiality, however, every byte of 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 the WriteFull policy below.

We implemented WriteFull and three simpler write policies for comparison:

WriteThru forces a disk write on every file write request. 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 becomes replaceable. Thus, it issues a write before the buffer is needed for re-use, but after it is no longer in use by some processor. Of course, the buffer is not usable until the write completes.

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. An old, unfilled buffer (no longer in use) will eventually be written. Thus, WriteFull becomes WriteBack during non-sequential access.

WriteThru and WriteBack were included for comparison because they are commonly used in memory caches and distributed file system caches. WriteFree is a compromise between WriteThru and WriteBack.

4 Experiments

We implemented a file system testbed as heavily parameterized parallel program running on a BBN GP1000 parallel processor [2], an MIMD machine. Since the multiprocessor does not have parallel disks, however, they are simulated. The testbed includes the synthetic workload, the file system, and the set of simulated disks. The use of 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. See [10] for more details.

4.1 Experimental Parameters

The parameters described here are the base from which we make other variations. There were 20 processes running on 20 processors. The patterns all wrote 4 MBytes of data, or about 200 KBytes per process. The block and buffer size was 1 KByte, and the record size was usually one block (but varied in one set of tests). This translates to 4000 blocks written to the disk. The cache contained 80 one-block buffers. We also had the capability to turn the cache off, so all requests went directly to the disk.

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.[2] 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[3] access time for the small, inexpensive disk drives that might be used. See [10] for more experimental details, and results of variations of many of these parameters.

[2]Actually, we used an exponential distribution truncated at 150 msec. The exponential nature of the distribution is not important. Once chosen, this delay was a fixed part of the workload, and did not vary from trial to trial.

[3]Recall that we do not assume a contiguous file layout on disk.

4.2 Measures

The primary performance metric is the total execution time. This is real ("wall-clock") time, incorporating all forms of overhead (such as memory contention, policy mistakes, *etc.*) and unexpected effects, and thus it is the best measure of overall performance. It includes the start, steady-state, and finish portions of the workload. The steady-state dominates. Longer access patterns would increase the steady-state proportion, where prefetching and caching are most effective. Thus, our results for the improvement due to caching are conservative.

A note on the data: Every data point in each plot represents the average of five trials (repetitions of the experiment with exactly the same parameters and workload). 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.

4.2.1 The Ideal Execution Time

We compare the experimental execution time to a simple model of the ideal execution time. 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 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. The following 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. In addition to the traditional benefits of overlap between overhead and I/O, and avoiding I/O due to locality (as in the quarter-block records below), the one-processor pattern **lw1** was able to use more than one disk by using delayed writes through the cache. This is a good example of a cache's ability to help applications use parallel disk bandwidth.

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

These experiments evaluate the effectiveness of our write policies across variations in workload and cache size, and seek to answer the following questions: What is the effect of cache size? How do the policies react to interprocess locality? 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?

4.4.1 Cache-size Variation

We varied the cache size from 20 one-block buffers to 200 one-block buffers. 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 the **gw** pattern (Figure 1), WriteBack was clearly slowest, since it delayed the disk write too long. WriteFree delayed the disk write for a full MRU block until the next file system access, which was after the process's compute cycle. With non-zero computation (Figure 2), this delay was too long, slowing down overall execution. This same effect also held for other patterns. Note that between 40 and 80 buffers were the maximum useful cache size for **gw**. Forty buffers corresponds to two buffers per process, which allowed one to be filled while the other is written to disk (double buffering).

The **lw1** patterns ran more slowly than the **gw** patterns, because one process could not drive all 20 disks at full efficiency (Figure 3). Larger caches benefited the **lw1** pattern by allowing more disk parallelism to be used, but this effect was not much of a factor when the 120 seconds of computation dominated.

The write-only **seg** patterns had a difficult disk access pattern (all processes began on the same disk). A large cache allowed processes to continue writing even when some disks were overloaded (Figure 4). In effect, larger caches allowed **seg** to use more disks concurrently. This is an excellent example of the ability of a 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.

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.

4.4.2 Record-size Variation

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 5a 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 (rewriting the same block to disk many times). WriteFree was smarter, waiting until the buffer was mostly unused before issuing a disk write, but it still had some mistakes and did not immediately write the blocks to disk when they were full. 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.

Note the dips in the curves for all but WriteFull. These occur at integral record sizes

(1, 2, 4, 5, 8, and 10 blocks),[4] where there was only one access per block. This avoided any opportunity for mistakes, which were common in the non-integral record sizes.

The results for **lw1** are shown in Figure 5b. 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. The record-size variation for the **seg** pattern (Figure 5c) 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.

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 [21], provided efficient and effective caching for our workload. More complex strategies do not appear to be necessary. Most importantly, it was an effective base for studying write policies for write-only patterns. Caching was able to use locality, including interprocess locality, to help applications use the parallel disk bandwidth.

Given the types of write-only access patterns we expect to be common in scientific workloads, our exploration of four methods shows that WriteFull was consistently at or near the best performance in all situations. A small cache (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.) High-performance parallel file writing in scientific workloads is definitely possible with these simple caching techniques. Further study is needed, however, to determine the effects of multiple files, multiple applications, or of different workload types.

[4]We used these record sizes because they divided the 4000 blocks into an integral number of fixed-size records.

References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212, 1991.

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

[3] T. W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[4] E. DeBenedictus and J. M. del Rosario. nCUBE parallel I/O software. In *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, Apr. 1992. To appear.

[5] P. Dibble, M. Scott, and C. 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.

[6] P. C. Dibble. *A Parallel Interleaved File System.* PhD thesis, University of Rochester, Mar. 1990.

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

[8] J. C. French, T. W. Pratt, and M. 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.

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

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

[11] E. L. Miller and R. H. Katz. Input/output behavior of supercomputer applications. In *Proceedings of Supercomputing '91*, pages 567–576, Nov. 1991.

[12] nCUBE Corporation. nCUBE 2 supercomputers: Technical overview. Brochure, 1990.

[13] J. Ousterhout, H. D. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. 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, Dec. 1985.

[14] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[15] P. 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.

[16] L. Pointer. PERFECT: Performance evaluation for cost-effective transformations: Report 2. Technical Report 964, CSRD — Univ. of Illinois, Nov. 1991. With Addenda 1 and 2.

[17] T. W. Pratt, J. C. French, P. M. Dickens, and S. A. Janet, Jr. A comparison of the architecture and performance of two parallel file systems. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 161–166, 1989.

[18] A. L. N. Reddy and P. Banerjee. A study of I/O behavior of Perfect benchmarks on a multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 312–321, 1990.

[19] K. Salem and H. Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[20] A. J. Smith. Disk cache-miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, Aug. 1985.

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

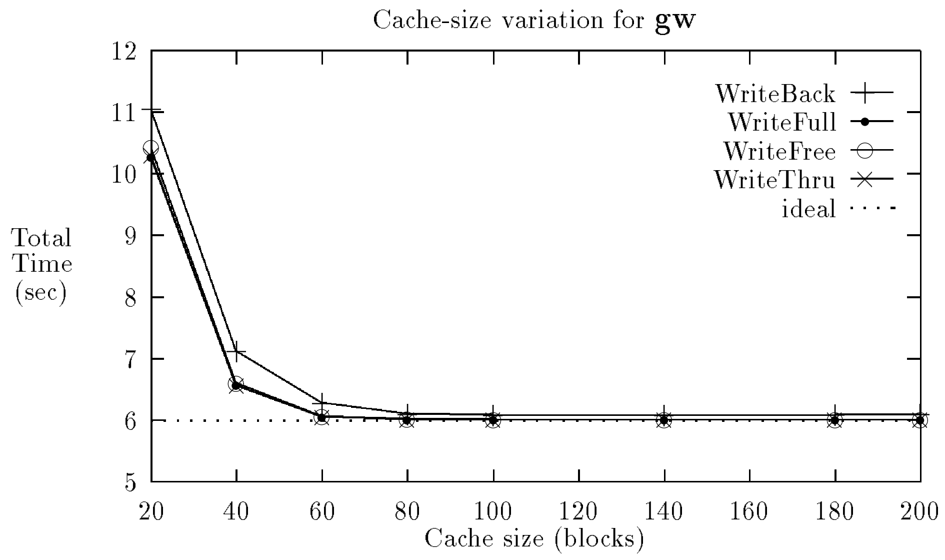


Figure 1: Cache-size variation for write pattern **gw**.

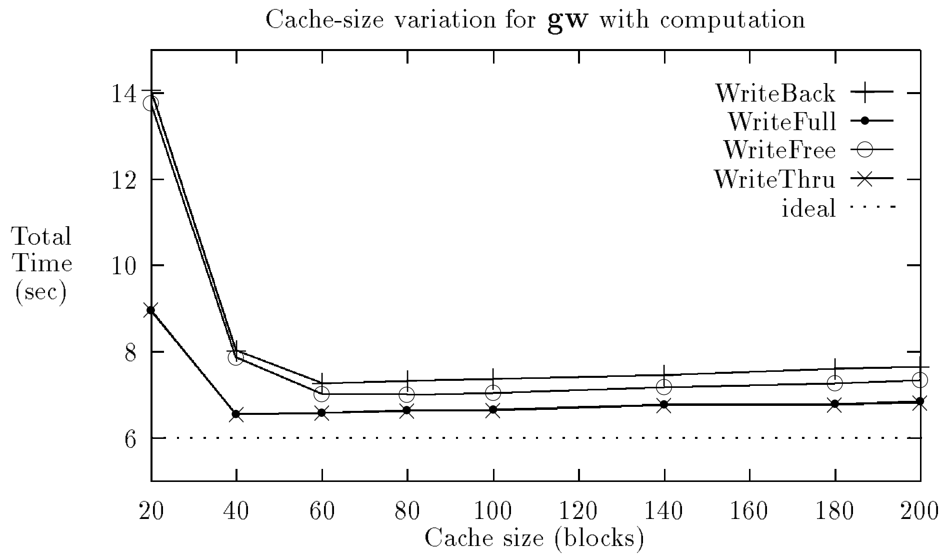


Figure 2: Cache-size variation for write pattern **gw** with computation.

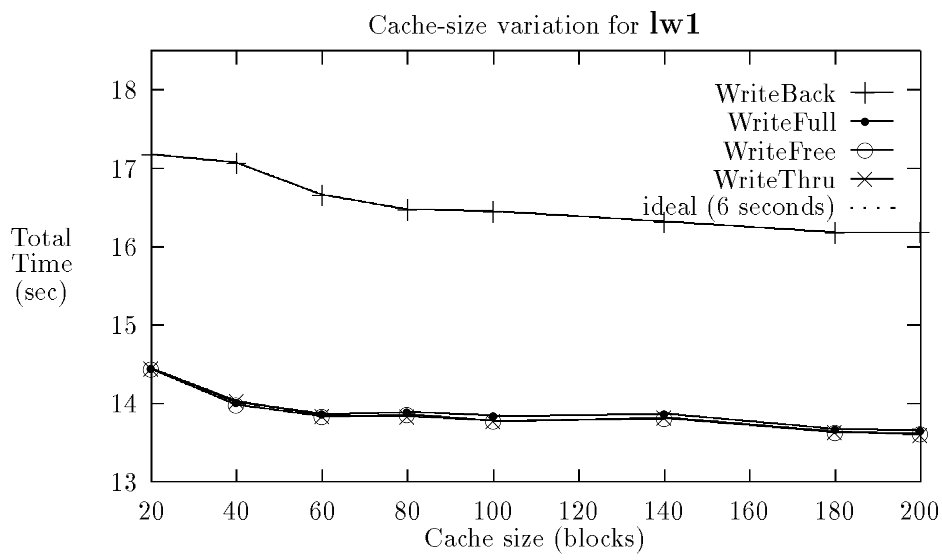


Figure 3: Cache-size variation for write pattern **lw1**.

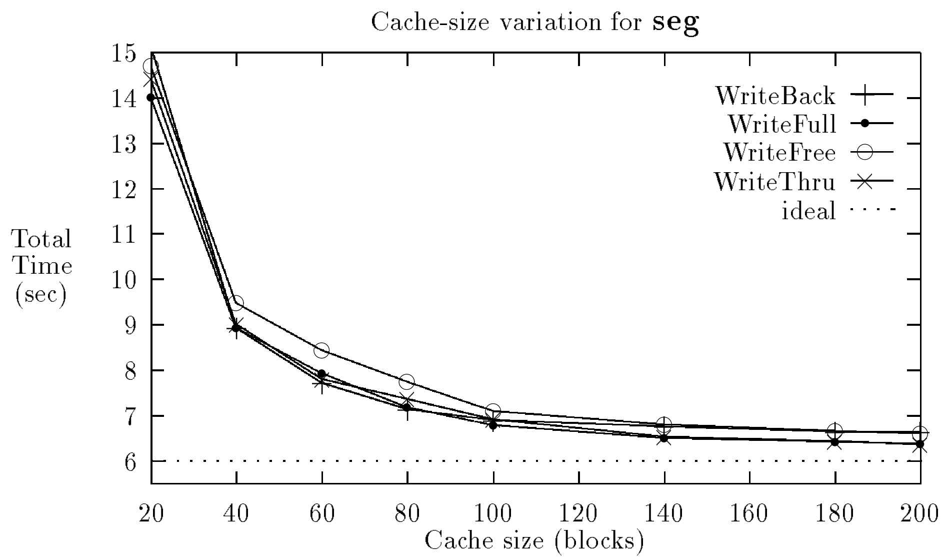
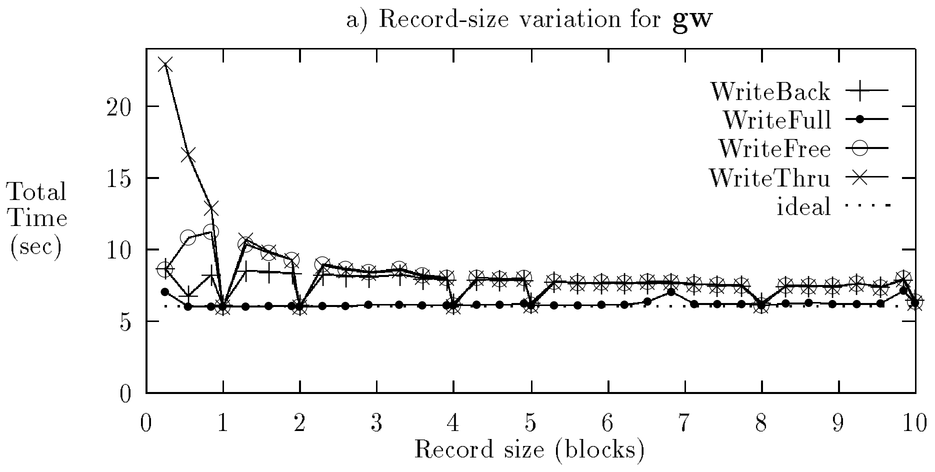
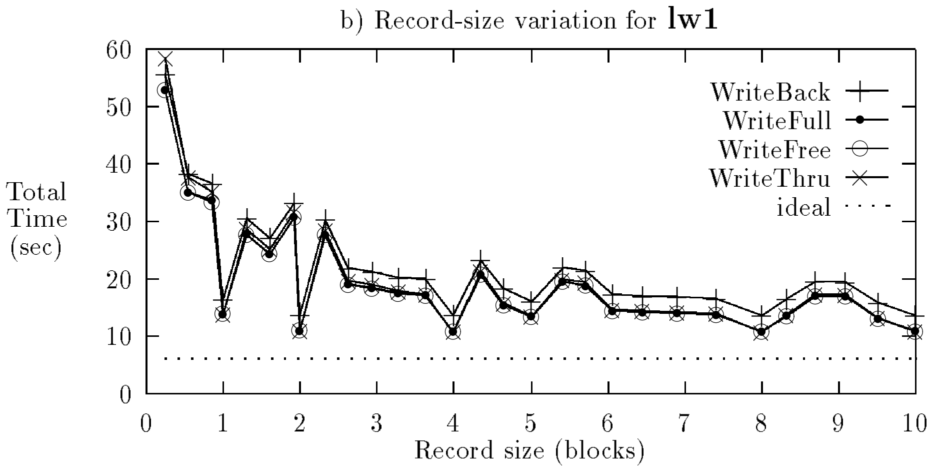


Figure 4: Cache-size variation for write pattern **seg**.





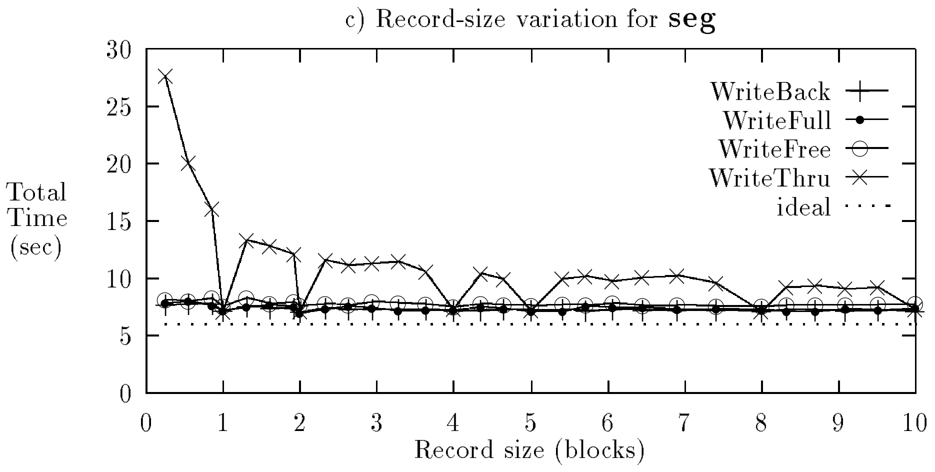


Figure 5: Record-size variation for all three write patterns.

Biographies

David F. Kotz received the A.B. degree in computer science and physics from Dartmouth College, Hanover NH, in 1986. He received the M.S. and Ph.D. degrees in computer science from Duke University in 1989 and 1991, respectively. His research interests include secondary memory management and file system design for MIMD multiprocessors, and algorithms for concurrent data structures. He has been an Assistant Professor of Mathematics and Computer Science at Dartmouth College, Hanover NH, since 1991.

Carla Schlatter Ellis received the B.S. degree from the University of Toledo, Toledo OH, in 1972 and the M.S. and Ph.D. degrees from the University of Washington, Seattle, in 1977 and 1979. She is currently an Associate Professor in the Department of Computer Science, Duke University, Durham NC. Previously, she was a member of the computer science faculties at the University of Oregon, Eugene, from 1978 to 1980, and at the University of Rochester, Rochester NY, from 1980 to 1986.

Captions

Figures:

Figure 1. Cache-size variation for write pattern **gw**.

Figure 2. Cache-size variation for write pattern **gw** with computation.

Figure 3. Cache-size variation for write pattern **lw1**.

Figure 4. Cache-size variation for write pattern **seg**.

Figure 5. Record-size variation for all three write patterns.

Footnotes:

1. Note that this is not, strictly speaking, an "MRU" replacement policy, which *replaces* the most-recently-used block!

2. Actually, we used an exponential distribution truncated at 150 msec. The exponential nature of the distribution is not important. Once chosen, this delay was a fixed part of the workload, and did not vary from trial to trial.

3. Recall that we do not assume a contiguous file layout on disk.

4. We used these record sizes because they divided the 4000 blocks into an integral number of fixed-size records.