NASA-CR-202744

File-System Workload on a Scientific Multiprocessor

David Kotz and Nils Nieuwejaar
Dartmouth College

The Charisma project records individual read and write requests in live, multiprogramming, parallel workloads. This information can be used to design more efficient multiprocessor systems.

Many scientific applications have intense computational and I/O requirements. Although multiprocessors have permitted astounding increases in computational performance, the formidable I/O needs of these applications cannot be met by current multiprocessors and their I/O subsystems. To prevent I/O subsystems from forever bottlenecking multiprocessors and limiting the range of feasible applications, new I/O subsystems must be designed.

The successful design of computer systems (both hardware and software) depends on a thorough understanding of their intended use. A system's designer optimizes the policies and mechanisms for the cases expected to be most common in the user's workload. In the case of multiprocessor file systems, however, designers have been forced to build file systems based only on speculation about how they would be used, extrapolating from file-system characterizations of general-purpose workloads on uniprocessor and distributed systems or scientific workloads on vector supercomputers (see sidebar on related work). To help these system designers, in June 1993 we began the Charisma project, so named because the project sought to *characterize I/O in scientific multiprocessor applications* from a variety of production parallel computing platforms and sites.

The Charisma project is unique in recording individual read and write requests in live, multiprogramming, parallel workloads (rather than from selected or nonparallel applications). In this article, we present the first results from the project: a characterization of the file-system workload on an iPSC/860 multiprocessor running production, parallel scientific appli-

1063-6552/95/$4.00 © 1995 IEEE

The iPSC/860 and CFS

The iPSC/860 is a distributed-memory, message-passing, MIMD machine. The compute nodes are based on the Intel i860 processor and are connected by a hypercube network. I/O is handled by dedicated I/O nodes, which are each connected to a single compute node rather than directly to the hypercube interconnect. The I/O nodes are based on the Intel i386 processor and each is connected to a single SCSI disk drive. There may also be one or more service nodes that handle such things as Ethernet connections or interactive shells.

The iPSC/860 at NASA Ames has 128 compute nodes, each with 8 Mbytes of memory, and 10 I/O nodes, each with 4 Mbytes of memory and a single 760 Mbyte disk drive. There is also a single service node that handles a 10-Mbit-per-second Ethernet connection to the host computer. The total I/O capacity is 7.6 Gbytes and the total bandwidth is less than 10 Mbytes per second.

Intel's CFS stripes each file across all disks in 4-Kbyte blocks. Compute nodes send requests directly to the appropriate I/O node for service. Only the I/O nodes have a buffer cache. CFS provides four *I/O modes* to help the programmer coordinate parallel access. Mode 0 gives each process its own file pointer; mode 1 shares a single file pointer among all processes; mode 2 is like mode 1, but enforces a round-robin ordering of accesses across all nodes; and mode 3 is like mode 2, but restricts the access sizes to be identical. More details about CFS and its performance can be found in work by Pierce,[1] Nitzberg,[2] and French et al.[3]

References

1. P. Pierce, "A Concurrent File System for a Highly-Parallel Mass Storage System," in *Fourth Conf. Hypercube Concurrent Computers and Applications*, Vol. 1, Golden Gate Enterprises, Los Altos, Calif., 1989, pp. 155-160.

2. B. Nitzberg, "Performance of the iPSC/860 Concurrent File System," Tech. Report RND-92-020, NAS Systems Division, NASA Ames, Dec. 1992.

3. J.C. French, T.W. Pratt, and M. Das, "Performance Measurement of the Concurrent File System of the Intel iPSC/2 Hypercube," *J. Parallel and Distributed Computing*, Vol. 17, No. 1-2, Jan./Feb. 1993, pp. 115-121.

cations at NASA's Ames Research Center. We use the resulting information to address the following questions:

- What did the job mix look like — that is, how many jobs ran concurrently? How many processors did each use? How many files?
- How many files were read and written? Which were temporary files? What were their sizes?
- What were typical read and write request sizes, and how were they spaced in the file? Were the accesses sequential? In what way?
- What forms of locality were there? How might caching be useful?
- What are the implications for file-system design?

Methods

To be useful to a system designer, a workload characterization must be based on a realistic workload similar to what is expected of it in the future. This meant that we had to trace a multiprocessor file system that was in use for *production* scientific computing. The Intel iPSC/860 at NASA Ames' Numerical Aerodynamics Simulation (NAS) facility met this criterion (see sidebar). (The facility's three newer multiprocessors, an Intel Paragon, a Thinking Machines CM-5, and an IBM SP-2, do not yet have a mature production workload.)

Ideally, a workload characterization is an architecture-independent representation of the work generated by a group of users in a particular type of computing environment. However, since the architectures of different parallel I/O subsystems are so diverse, any observed workload will be tied to a particular machine. While we have tried to factor out these effects as much as possible, some care should be taken in generalizing our results.

Data collection

For our study, one trace file was collected for the entire file system. We traced only the I/O that involved the Concurrent File System (CFS); we did not record any I/O that was done through standard input and output or to the host file system (all limited to sequential, Ethernet speeds). We collected data for about 156 hours over a period of three weeks. While we did not trace continuously for the whole three weeks, we tried to get a realistic picture of the whole workload by tracing at different times of the day and of the week, including nights and weekends. The period covered by a single trace file ranges from 30 minutes to 22 hours. The longest continuously traced period was about 62.5 hours. Tracing was usually initiated when the machine was idle. For those few cases in which a job was running when we began tracing, the job was not traced. Tracing was stopped in one of two ways: manually or by a system crash. The machine was usually idle when a trace was manually stopped.

The trace files begin with a header record containing enough information to make the file self-descriptive, and continue with a series of event records (one per event). These events include individual read and write requests, as well as operations like file extensions and deletions.

Related work

There has never been an extensive study of a production scientific workload on a multiprocessor file system. Related file-system workload studies can be classified as characterizing general-purpose workstations (or workstation networks), scientific vector applications, or scientific parallel applications.

General-purpose workstations. Uniprocessor file access patterns have been measured many times. Ousterhout et al. measured isolated Unix workstations,[1] and Baker et al. measured a distributed Unix (Sprite) system.[2] All of these studies cover general-purpose (engineering and office) workloads with uniprocessor applications.

Scientific vector applications. Some studies specifically examined scientific workloads. Del Rosario and Choudhary provide an informal characterization of grand-challenge applications.[3] Miller and Katz traced specific I/O-intensive Cray applications to determine the per-file access patterns, focusing primarily on access rates.[4] Pasquale and Polyzos studied I/O-intensive Cray applications, focusing on patterns in the I/O rate.[5] All of these studies are limited to uniprocess applications on vector supercomputers.

Scientific parallel applications. Reddy et al. chose five sequential scientific applications from the Perfect benchmarks and parallelized them for an eight-processor Alliant, finding only sequential file-access patterns.[6] This study is interesting, but far from what we need: the sample size is small; the programs are parallelized sequential programs, not parallel programs *per se*; and the I/O itself was not parallelized. Cypher et al. studied individual parallel scientific applications, measuring temporal patterns in I/O rates.[7]

References

1. J. Ousterhout et al., "A Trace-Driven Analysis of the Unix 4.2 BSD File System," in *Proc. 10th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1985, pp. 15–24.

2. M.G. Baker et al., "Measurements of a Distributed File System," in *Proc. 13th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1991, pp. 198–212.

3. J.M. del Rosario and A. Choudhary, "High Performance I/O for Parallel Computers: Problems and Prospects," *Computer*, Vol. 27, No. 3, Mar. 1994, pp. 59–68.

4. E.L. Miller and R.H. Katz, "Input/Output Behavior of Supercomputer Applications," in *Proc. Supercomputing 91*, IEEE Computer Society Press, Los Alamitos, Calif., 1991, pp. 567–576.

5. B.K. Pasquale and G.C. Polyzos, "A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload," in *Proc. Supercomputing 93*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 388–397.

6. A. L. Narasimha Reddy and P. Banerjee, "A Study of I/O Behavior of Perfect Benchmarks on a Multiprocessor," in *Proc. 17th Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 312–321.

7. R. Cypher et al., "Architectural Requirements of Parallel Scientific Applications with Explicit Communication," in *Proc. 20th Annual Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 2–13.

Since one of the Charisma project's goals is to organize and facilitate a multiplatform file-system tracing effort, we have defined a large set of event records suitable for both single- and multiple-instruction multiple-data systems (SIMD and MIMD systems).

On the iPSC/860, high-level CFS calls are implemented in a library that is linked with the user's program. We instrumented the library calls to generate an event record each time they were called. The event records were buffered at each compute node and periodically sent to a data collector running on the service node. The collector then wrote the data to the central trace file (itself on CFS). The collector's use of CFS was not recorded in the trace.

Since our instrumentation was almost entirely within a user-level library, there were some jobs whose file accesses were not traced. These included most system programs (such as ls, cp, and ftp) as well as user programs that were not relinked during the period we were tracing. We did, however, record all job starts and ends through a separate mechanism. While we were tracing, 3016 jobs were run on the compute nodes, of which 2237 were only run on a single node. We actually traced at least 429 of the 779 multinode jobs and at least 41 of the single-node jobs. As a tremendous number of the single-node jobs were system programs, it is not surprising nor necessarily undesirable that so many were untraced. In particular, there was one single-node job that was run periodically and that accounted for over 800 of the single-node jobs, simply to check the status of the machine. There was no way to distinguish between a job that was untraced from a job that simply did no CFS I/O, so the numbers of traced jobs are a lower bound.

One of our primary concerns was to minimize the degree that our measurement perturbed the workload. We identified three ways that it might do so. Our first concern was network contention. We expected users' jobs to generate a great many event records. Had we sent a message to the data collector for each event record, we would have created unreasonable congestion near the collector, or perhaps in the overall machine. Since large messages on the iPSC are broken into 4-Kbyte blocks, we created a buffer of that size on each node to hold local event records. This buffer let us reduce the number of messages sent by more than 90% without stealing much memory from user jobs.

The second concern was local CFS overhead. Since we were tracing every I/O operation in a production envi-

Table 1. Summary statistics of the traces we collected, and of files opened.[2] Only those jobs whose file accesses were caught by our library are included here.

Jobs		470	
Mbytes	Read	38,812	
	Written	44,725	
Files	Opened	63,779	100%
	Read	14,540	22.8%
	Written	44,500	69.8%
	Both	2259	3.5%
	Neither	2480	3.9%

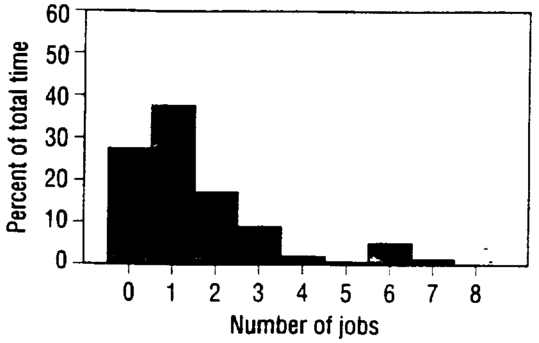

Figure 1. Amount of time the machine spent with the given number of jobs running. This data includes all jobs, even if their file access could not be traced.

ronment, it was imperative that the per-call overhead be kept to a minimum to avoid inconveniencing the users. By buffering records on the compute nodes, we were able to avoid the cost of message passing on every call to CFS.

Our final concern was that we might increase contention for the I/O subsystem. We tried to minimize this by creating a large buffer for the data collector and writing the data to CFS in large sequential blocks. Although we collected about 700 Mbytes of data, our trace files accounted for less than 1% of the total traffic.

Simple benchmarking of the instrumented library revealed that the overhead our instrumentation added was virtually undetectable in many cases. The worst case we found was a 7% increase in execution time on one run of the NAS NHT-1 Application-I/O Benchmark.[1] After the instrumented library was put into production use, anecdotal evidence suggests that there was no noticeable performance loss.

Analysis

The raw trace files required some simple postprocessing before they could be easily analyzed. This postprocessing included data realignment, clock synchronization, and chronological sorting.

Since each node buffered 4 Kbytes of data before sending it to the central data collector, the raw trace file contained only a partially ordered list of event records. Ordering the records was complicated by the lack of synchronized clocks on the iPSC/860. Each node maintains its own clock; the clocks are synchronized at system startup but each drifts significantly and differently after that. We partially compensated for the asynchrony by timestamping each block of records when it left the node and again when it was received at the data collector. From the difference between the two we could approximately adjust the event order to compensate for each node's clock drift relative to the collector's clock. This technique allowed us to get a closer approximation of the event order. Nonetheless, it is still an approximation, so much of our analysis is based on spatial, rather than temporal, information.

Results

We characterize the workload from the top down, beginning with the number of jobs in the machine and the number and use of files by all jobs. We then examine individual I/O requests by looking for sequentiality, regularity, and sharing in the access pattern. Finally, we evaluate the effect on caching through trace-driven simulation.

Jobs

Table 1 provides an overview of this workload's characteristics. Figure 1 gives an initial look into the details behind Table 1 by showing the amount of time the machine spent running a given number of jobs. For more than a quarter of the traced period, the machine was idle (that is, running zero jobs). For about 35% of the time, it was running more than one job, sometimes as many as eight. Although not all jobs use the file system, a file system clearly must provide high-performance access by many concurrent, presumably unrelated, jobs. While uniprocessor file systems are tuned for this situation, most research into multiprocessor file systems has ignored this issue, focusing on optimizing single-job performance.

Of course, some of the jobs in Figure 1 were small, single-node jobs, and some were large parallel jobs. Figure 2 shows the distribution of compute nodes used by each job. Single-node jobs dominated the job population, although large parallel jobs dominated node usage. This dichotomy would be larger in new "self-hosting" parallel systems. The lesson here is that a successful file system must allow both small, sequential jobs and large, highly parallel jobs access to the same files under a variety of conditions and system loads.

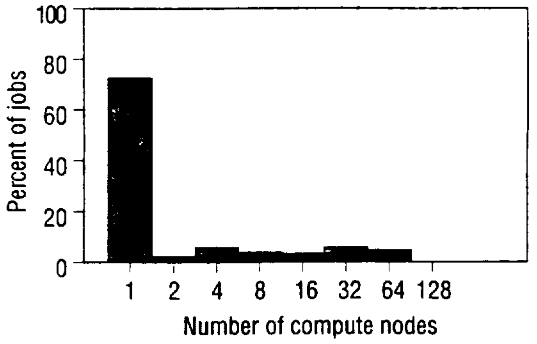

Figure 2. Distribution of the number of compute nodes used by jobs in our workload (even those whose file access could not be traced). The iPSC limits the choice to powers of 2.

Table 2. Among traced jobs, the number of files opened by jobs was often small (1–4).

No. of files	No. of jobs
1	71
2	15
3	24
4	120
5+	240

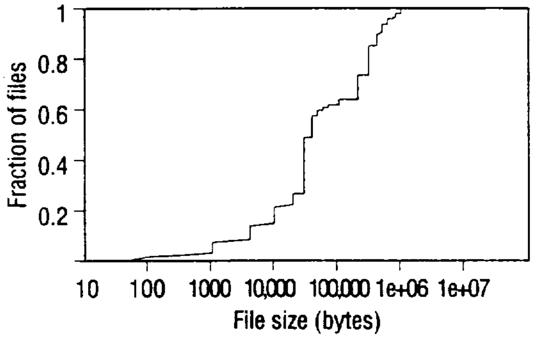

Figure 3. Cumulative distribution function (CDF) of the number of files of each size at close. For a file size *x*, CDF(*x*) represents the fraction of all files that had *x* or fewer bytes.

Files

In Table 1, files are classified by how they were actually used rather than by the mode in which they were opened. Note that many more files were written than were read (more than three times as many). It appears that programmers of traced applications often found it easier to open a separate output file for each compute node, rather than coordinating writes to a common output file — as evidenced by the substantially smaller average number of bytes written per file (1.2 Mbytes) than average bytes read per file (3.3 Mbytes). Also, there were extremely few files that were read and written in the same open. This is common in Unix file systems[3] and may be accentuated here by the difficulty in coordinating concurrent reads and writes to the same file (the CFS file-access modes are of little help for read-write access). We suspect that most of the files that were not accessed at all were opened by applications that terminated prematurely.

Table 2 shows that most jobs opened only a few files over the course of their execution, although a few opened many (one job opened 2217 files). Some of the jobs that opened a large number of files were opening one file per node. Although not all files were open concurrently, file-system designers must optimize access to several files within the same job.

We found that only 0.61% of all opens were to "temporary" files (a file deleted by the same job that created it), and nearly all of those may have been from one application. The rarity of temporary files and of files that were both read and written in the same open indicates that few applications chose to use files as an extension of memory for an "out of core" solution. Many of the NASA Ames applications are computational fluid dynamics codes, for which out-of-core methods are in general too slow.

Figure 3 shows that most of the files accessed were large (10 Kbytes to 1 Mbyte). (Because there were many small files and many distinct peaks across the range of sizes, there was no constant granularity that captured the detail we felt was important in a histogram. We chose to plot the file sizes on a logarithmic scale with pseudo-logarithmic bucket sizes: The bucket size between 10 and 100 bytes is 10 bytes; between 100 and 1000 it is 100 bytes, and so on.)

It is important to note that each of the largest jumps in the figure is primarily due to one or two applications; undue emphasis should not be placed on the specific numbers as opposed to the general tendency toward larger files. Although these files were larger than those in a general-purpose file system,[4] they were smaller than we would expect to see in a scientific supercomputing environment.[5] We suspect that users limited their file sizes due to the small disk capacity (7.2 Gbytes) and limited disk bandwidth (10 Mbytes per second peak).

I/O request sizes

Figures 4 and 5 show that the vast majority of reads are small, but that most bytes are transferred through large reads. Indeed, 96.1% of all reads were for fewer than 4000 bytes, but those reads transferred only 2.0% of all data read. Similarly, 89.4% of all writes were for fewer

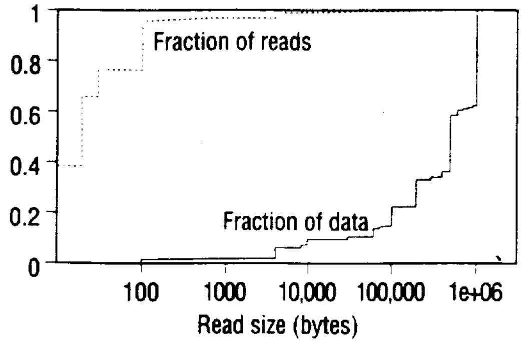

Figure 4. CDF of the number of reads by request size and of the amount of data transferred by request size.

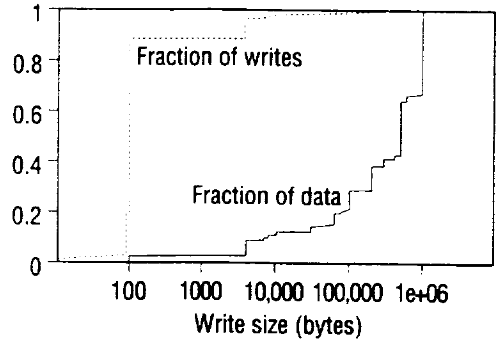

Figure 5. CDF of the number of writes by request size and of the amount of data transferred by request size.

than 4000 bytes, but those writes transferred only 3% of all data written. The number of small requests is surprising due to their poor performance in CFS.[6] The jump at 4 Kbytes indicates that some users have optimized for the file-system block size, but it appears that most users prefer ease of programming over performance.

The figures show spikes in the number of small requests as well as in the data transferred by 1-Mbyte requests. While the spikes of small requests occurred throughout the tracing period, one trace alone (probably one job alone) contributed the spike at 1 Mbyte. Although the specific position of the spikes is likely due to the effect of individual applications, we believe that the preponderance of small request sizes is the natural result of parallelization by distributing file data across many processors, and would be found in other workloads using a similar file-system interface.

SEQUENTIALITY

A common characteristic of file workloads, particularly scientific workloads, is that files are accessed sequentially. We define a *sequential* request to be one that is at a higher file offset than the previous request from the same compute node, and a *consecutive* request to be a sequential request that begins where the previous request ended. Figures 6 and 7 show the amount of sequential and consecutive access to files with more than one request in our workload.

The most notable features of these graphs are the spikes at 0% and 100%; most files were either entirely sequential (or consecutive) or not at all. Not surprisingly, access to read-write files was primarily nonsequential. By far, most read-only and write-only files were 100% sequential. Most (86%) write-only files were 100% consecutive, but that was largely due to the fact that most write-only files were written only by one processor. Only 29% of read-only files, however, were 100% consecutive. The remainder (nonconsecutive, sequential read-only files) were the result of interleaved access, where successive records of the file are accessed by different nodes; from the perspective of an individual node, some bytes must be skipped between one request and the next.

I/O-REQUEST INTERVALS

We define the number of bytes skipped to be the *interval size*. Consecutive accesses have interval size 0. The number of *different* interval sizes used in each file, across all nodes that access that file, is shown in Table 3. A surprising number of files were read or written in one request per node (that is, there were no intervals). Over

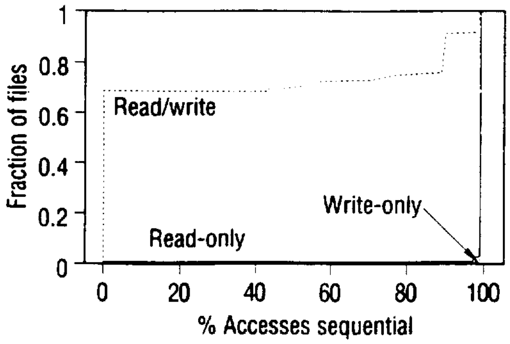

Figure 6. CDF of sequential access to files on a per-node basis.

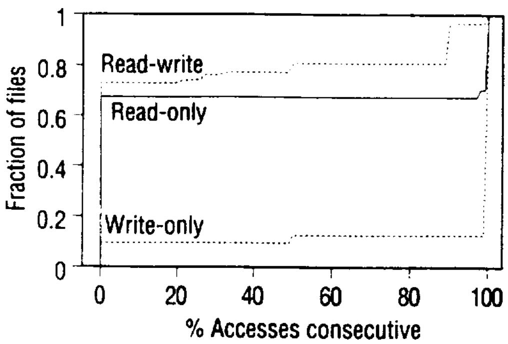

Figure 7. CDF of consecutive access to files on a per-node basis.

Table 3. The number of different interval sizes used in each file across all participating nodes. Zero represents those cases where only one access was made to a file, per node.

No. of sizes	No. of files	Percent of total files
0	23,291	36.5
1	37,148	58.2
2	2561	4.0
3	105	0.2
4+	674	1.0

Table 4. The number of different request sizes used in each file across all compute nodes. Files with zero different sizes were opened and closed without being accessed.

No. of sizes	No. of files	Percent of total files
0	2480	3.9
1	25,523	40.0
2	32,779	51.4
3	2510	3.9
4+	487	0.8

99% of the 1-interval-size files were consecutive accesses (in other words, every interval had size 0). The remainder of 1-interval-size files, along with the 2-interval-size files, represent 5% of all files, and indicate another form of highly regular access pattern. Only 1.2% of all files had 3 or more different interval sizes, and their regularity (if any) was more complex.

To get a better feel for this regularity, we also counted the number of different *request sizes* used in each file, as shown in Table 4. More than 90% of the files were accessed with only one or two request sizes. Combining the regularity of request sizes with the regularity of interval sizes, many applications clearly used regular, structured access patterns.

Strided access

A series of requests to a file is a *simple-strided* access pattern if each request is the same size, and if the offset of the file pointer is incremented by the same amount between each request. This would correspond, for example, to the series of I/O requests generated by a node within an application reading a column of data from a matrix stored in row-major order. A portion of the file accessed with a strided pattern is a *strided segment*. A *nested-strided* access pattern is recursively similar to a simple-strided access pattern, in that it is composed of strided segments separated by regular strides in the file.

Higher-level analysis revealed that well over 90% of the accesses in the traced workload were part of either a simple- or a nested-strided access pattern caused by the distribution of data across multiple compute nodes.[7] Of the files in the workload, 26% were accessed (at least in part) in a strided fashion, and nearly 1/3 of those were accessed in a nested-strided fashion. Of the remaining files, 99% either had too few accesses to exhibit any pattern, were only accessed by a single node, or were accessed in a consecutive pattern. Thus, less than 1% of all files were accessed in an irregular, parallel access pattern.

Synchronization

Given the regular request sizes and interval sizes shown in Tables 3 and 4, Intel's I/O modes would seem to be helpful. Our traces show, however, that over 99% of the files used mode 0; that is, less than 1% used modes 1, 2, or 3. Tables 3 and 4 give a hint as to why: although there were few different request sizes and interval sizes, there were often more than one, something not easily supported by the automatic file modes. It may also be that these modes were slower than mode 0, so that programmers chose not to use them.

Sharing

A file is *shared* if more than one job or process opens it. If the opens overlap in time, the file is *concurrently shared*. It is *write-shared* if one of the opens involves writing the file. In uniprocessor and distributed-system workloads, concurrent sharing is known to be rare.[4] In a parallel file system, concurrent file sharing among processes within a job is presumably the norm, while concurrent file sharing between jobs is likely to be rare. Indeed, in our traces we saw a great deal of file sharing within jobs, and *no* concurrent file sharing between jobs. The interesting question is *how* the individual bytes and blocks of the files were shared. Figure 8 shows the percentage of files (that were concurrently opened by multiple nodes) with varying amounts of byte and block sharing. There was more sharing for read-only files than for write-only or read-write files, which is not surprising given the complexity of coordinating write sharing. Indeed, 70% of read-only files had 100% of their bytes shared, while 90% of write-only files had no bytes shared. While half of all read-write files (not shown in Figure 8) were 100% byte-shared, 93% of them were 100% block-shared,

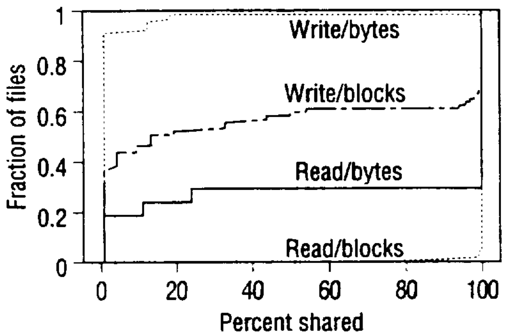

Figure 8. CDF of file sharing between nodes in read-only and write-only files at byte and block granularity.

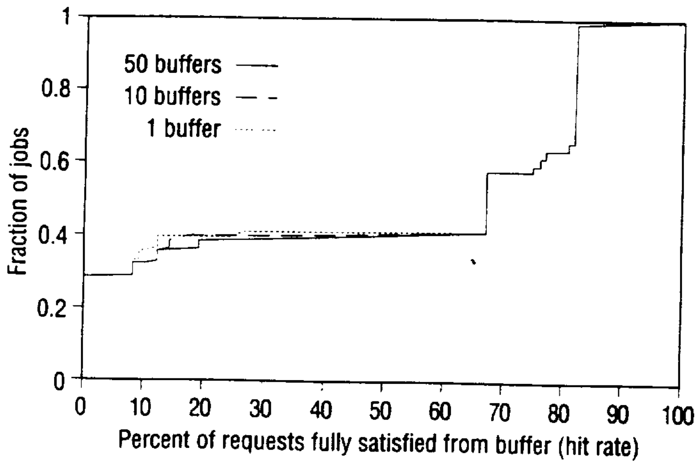

Figure 9. Results of compute-node caching simulation. Hit rates differed from job to job, with three distinct clumps, indicating that the cache either helped or did not. One buffer was as good as many buffers.

which would stress a cache consistency protocol, if present. Overall, the amount of block sharing implies strong *interprocess* spatial locality, and suggests that caching might be successful.

Caching

Buffering and caching are common in traditional file systems, and — with the right policies — can be successful in multiprocessor file systems. One advantage of buffers is to combine several small requests (which were common in this workload) into a few larger requests that can be more efficiently served by disk hardware. Indeed, with redundant disk arrays common on today's multiprocessors (such as the Intel Paragon and the KSR-2), it is even more important to avoid small requests at the disk level. Fortunately, the small requests seen in Figures 4 and 5, when coupled with small interval size, lead to spatial locality. Other potential benefits may come from temporal or interprocess locality in the access pattern.

In a distributed-memory machine, it is possible to place a buffer cache at the compute nodes, at the I/O nodes, or both. We evaluated all three with trace-driven simulation.

Compute-node caching

The amount of block sharing in write-only and read-write files show that any attempt to maintain write buffers at the compute nodes would necessitate a cache consistency protocol, so we restricted our effort to read-only files. The results of a simple trace-driven simulation of a compute-node cache of 4-Kbyte (one block), read-only buffers with least recently used replacement are shown in Figure 9. We consider a *hit* to be any request that was *fully* satisfied from the local buffer (that is, with no request sent to an I/O node).

Caching success, as indicated by a high hit rate, was limited to a subset of the jobs: 40% of the jobs had a greater than 75% hit rate, but 30% of the jobs had a 0% hit rate. Further, for those jobs where a cache was beneficial, a single one-block buffer per compute node was usually sufficient. A single buffer could maintain a high hit rate in patterns with a small request size (which was common; see Figures 4 and 5) and a short (perhaps zero) interval size. Clearly there was spatial locality in our workload, and not much temporal locality, or multiple buffers would have helped more. (Multiple buffers were useful in very few jobs, apparently those that were interspersing reads from more than one file. In those cases, a single buffer *per file* would have been appropriate.) In short, it appears that a one-block buffer per compute node, per file, may be useful for read-only files, but a careful performance analysis is still necessary.

I/O-node caching

Given the apparent interprocess locality, I/O-node caching should be successful. To find out, we ran a trace-driven simulation of I/O-node caches, with 4-Kbyte buffers managed by either a least recently used or FIFO replacement policy. These I/O-node caches served all compute nodes, all files, and all jobs, according to our best guess of the event ordering within our traces. We assumed the file was striped in a round-robin fashion at a one-block granularity. No compute-node cache was used.

Figure 10 shows the results of the simulation With least-recently-used (LRU) replacement, a small cache (4000 4-Kbyte buffers over all I/O nodes) was sufficient to reach a 90% hit rate. With FIFO replacement, nearly 20,000 buffers were needed to obtain a 90% hit rate, because FIFO does not give preference to blocks with high locality. It made little difference whether the buffers were focused on a few I/O nodes or spread over many (that is, the hit rates were similar; performance is another issue). The success of such a small cache, coupled with the apparent lack of intraprocess locality in many jobs (see Figure 9), reconfirms the presence of interprocess spatial locality.

As a final test, we simulated the combination of a single buffer per compute node and a cache at each of 10 I/O nodes. The result was a only a 3% reduction in the I/O-node hit rate when each I/O node had a small cache of 50 buffers. This further suggests that most of the hits in the I/O-node cache were indeed a result of interprocess locality because, as Figure 9 shows, the limited intraprocess locality was filtered out by the compute-node cache.

In constrast, Miller and Katz's tracing study[5] found little benefit from caching, although it did show a benefit from prefetching and write-behind. Both their workload and ours involve sequential access patterns; the difference is that the small requests in our access pattern lead to intraprocess spatial locality, and the distribution of a sequential pattern across parallel compute nodes leads to interprocess spatial locality, both of which could be successfully captured by caching.

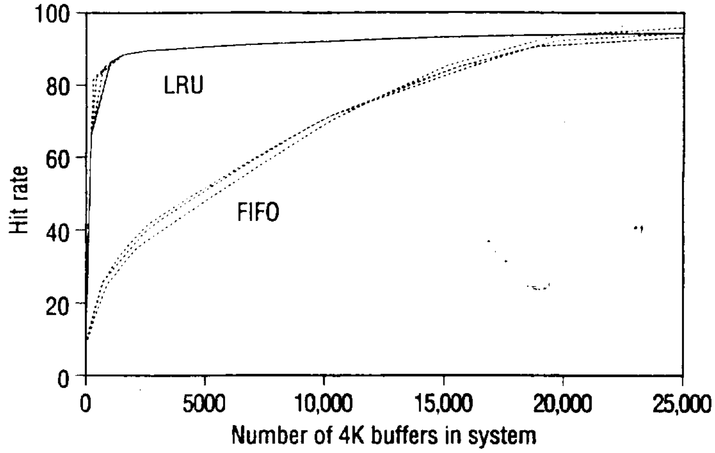

Figure 10. Results of I/O-node caching simulation. Each line represents a complete run of the simulation with a fixed number of I/O nodes ranging from 1 to 20.

Although this workload had many characteristics in common with those in previous studies of scientific applications and file systems (large file sizes, sequential access, little inter-job concurrent sharing), parallelism had a significant effect on some workload characteristics (smaller request sizes, and lots of intrajob concurrent file sharing) and added some new characteristics (nonconsecutive sequential access and interprocess spatial locality). A multiprocessor used for scientific applications will not be well served by a file system ported from a distributed system, which was tuned for a different set of workload characteristics. In particular, parallelism leads to new, interleaved access patterns with no temporal locality, and high interprocess spatial locality at the I/O node.

Compute-node caches are probably best implemented as a single buffer per file (but only if carefully managed for consistency). I/O-node caches can effectively combine small requests from many compute nodes, avoiding extraneous disk I/O and raising the potential for large disk I/Os, a significant benefit when the I/O nodes serve redundant disk arrays (which favor large transfers) rather than individual disks. Replacement policies other than least-recently-used or FIFO can optimize for sequential access and interprocess locality, rather than traditional spatial and temporal locality.[8]

Ultimately, we believe that the file-system interface *must change*. The current interface forces the programmer to break down large parallel I/O activities into small, noncontiguous requests. While compute-node and I/O-node caching can help, it would be better to support *strided I/O requests* from the programmer's interface to the compute node, and from the compute node to the I/O node. A strided request can express a regular request and interval size (which were common in our workload), effectively increasing the request size, lowering overhead, and perhaps eliminating the need for compute-node buffers. The Cray file system is an example of a system that supports strided requests.

Some of our results may be specific to workloads on Intel CFS file systems, or to NASA Ames' workload (computational fluid dynamics). Although the exact numbers are workload-specific, we believe that our conclusions are applicable to scientific workloads running on loosely coupled MIMD multiprocessors with a CFS-like interface — that is, an interface that encourages interleaved access and an independent file pointer for each node. This category includes many current multiprocessors.

We plan to continue collecting traces from other machines and environments to broaden and deepen the experimental data, and strengthen the generality of our conclusions. One such project has already produced results.[9] We may also convert these and other results into a meaningful, synthetic benchmark of parallel I/O. We will use this new knowledge to design a better multiprocessor file system, and use the traces in trace-driven simulations of new policies for caching, prefetching, load balancing, paging, and so forth. We also plan to evaluate new I/O architectures.

Acknowledgments

Many thanks to the NAS division at NASA Ames, to the individuals there who helped us with the iPSC/860 tracing effort (Jeff Becker, Russell Carter, Chris Kuszmaul, Art Lazanoff, Bill Nitzberg, and Leigh Ann Tanner), and to the many users who agreed to be traced. Many thanks also to Mike Best, Carla Ellis, Sam Fineberg, Orran Krieger, Apratim Purakayastha, Bernard Traversat, and the rest of the Charisma group for their helpful discussions. This research was supported in part by the NASA Ames Research Center under Agreement Number NCC 2-849.

References

1. R. Carter et al., "NHT-1 I/O Benchmarks," Tech. Report RND-92-016, NAS Systems Division, NASA Ames, 1992.
2. D. Kotz and N. Nieuwejaar, "Dynamic File-Access Characteristics of a Production Parallel Scientific Workload," Tech. Report PCS-TR94-211, Dept. of Math and Computer Science, Dartmouth College, Apr. 1994; revised Aug. 1994.
3. R. Floyd, "Short-Term File Reference Patterns in a Unix Environment," Tech. Report 177, Dept. of Computer Science, Univ. of Rochester, 1986.
4. M.G. Baker et al., "Measurements of a Distributed File System," *Proc. 13th ACM Symp. Operating Systems Principles*, ACM, New York, 1991, pp. 198–212.
5. E.L. Miller and R.H. Katz, "Input/Output Behavior of Supercomputer Applications," *Proc. Supercomputing 91*, IEEE Computer Society Press, Los Alamitos, Calif., 1991, pp. 567–576.
6. B. Nitzberg, "Performance of the iPSC/860 Concurrent File System," Tech. Report RND-92-020, NAS Systems Division, NASA Ames, 1992.
7. N. Nieuwejaar and D. Kotz, "A Multiprocessor Extension to the Conventional File-System Interface," Tech. Report PCS-TR94-230, Dept. of Computer Science, Dartmouth College, 1994.
8. D. Kotz and C.S. Ellis, "Caching and Writeback Policies in Parallel File Systems," *J. Parallel and Distributed Computing*, Vol. 17, No. 1–2, Jan./Feb. 1993, pp. 140–145.
9. A. Purakayastha, et al., "Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor," to appear in *Proc. IPPS '95*, IEEE Computer Society Press, Los Alamitos, Calif., 1995.

David Kotz has been an assistant professor of computer science at Dartmouth College since 1991. His research interests include parallel operating systems and architecture, multiprocessor file systems, transportable agents, single-address-space operating systems, parallel computer performance monitoring, and parallel computing in computer science education. He received his BA in computer science and physics from Dartmouth College in 1986, and his MS and PhD degrees in computer science from Duke University in 1989 and 1991, respectively. He is a member of the IEEE Computer Society, the ACM, and Usenix associations.

Nils Nieuwejaar is a graduate student in computer science at Dartmouth College. His research interests include parallel operating systems, multiprocessor file systems, and tools for developing parallel applications. He received his BA in computer science and philosophy from Bowdoin College in 1992.

The authors can be reached at the Dept. of Computer Science, Dartmouth College, Hanover, NH 03755-3510; Internet: dfk@cs.dartmouth.edu or nils@cs.dartmouth.edu. More information about the Charisma project can be found at http://www.cs.dartmouth.edu/research/charisma.html.