Dynamic File-Access Characteristics of a Production Parallel Scientific Workload

David Kotz　　　　　　Nils Nieuwejaar

Department of Computer Science
Dartmouth College, Hanover, NH 03755-3510
{dfk,nils}@cs.dartmouth.edu

Abstract

Multiprocessors have permitted astounding increases in computational performance, but many cannot meet the intense I/O requirements of some scientific applications. An important component of any solution to this I/O bottleneck is a parallel file system that can provide high-bandwidth access to tremendous amounts of data in parallel to hundreds or thousands of processors.

Most successful systems are based on a solid understanding of the expected workload, but thus far there have been no comprehensive workload characterizations of multiprocessor file systems. This paper presents the results of a three week tracing study in which all file-related activity on a massively parallel computer was recorded. Our instrumentation differs from previous efforts in that it collects information about every I/O request and about the mix of jobs running in a production environment. We also present the results of a trace-driven caching simulation and recommendations for designers of multiprocessor file systems.

1 Introduction

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 usage. 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. To fill this gap, the CHARISMA project began in June 1993 to **CHAR**acterize **I/O** in **S**cientific **M**ultiprocessor **A**pplications 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 non-parallel applications). This paper presents the first results from the project: a characterization of the file-system workload on an iPSC/860 multiprocessor running production, parallel scientific applications at NASA's Ames Research Center. We use the resulting information to address the following questions:

- What does the job mix look like: how many jobs run concurrently? how many processors did each use? how many files did each use?

- How many files were read and written? What were their sizes? Which were temporary files?

- What were typical read and write request sizes, and how were they spaced in the file? Were the accesses sequential, and in what way?

- What forms of locality were there? How might caching be useful?

- What are the implications for file-system design?

In the next section we describe previous studies of file-system workload, multiprocessor file systems, and

file-system caching. In Section 3 we outline our research methods, and in Section 4 present our results. Section 5 draws the overall conclusions.

2 Related work

As background, we describe many of the previous studies of file-system workload as well as some current multiprocessor file systems and caching studies.

2.1 Workload

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. Floyd and Ellis [12, 13] and Ousterhout *et al.* [28] measured isolated Unix workstations, and Baker *et al.* measured a distributed Unix (Sprite) system [1]. 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 [10]. Powell measured file sizes on a Cray-1 file system [31]. Miller and Katz traced specific I/O-intensive Cray applications to determine the per-file access patterns [25], focusing primarily on access rates. Pasquale and Polyzos studied I/O-intensive Cray applications, focusing on patterns in the I/O rate [29]. All of these studies are limited to uniprocess applications on vector supercomputers.

Scientific parallel applications. Crockett [7] and Kotz [20] hypothesize about the character of a parallel scientific file-system workload. Cormen and Kotz [6] discuss the needs of parallel-I/O algorithms. 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 [32]. 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.* [8] studied individual parallel scientific applications, measuring temporal patterns in I/O rates. Galbreath *et al.* [16] present a useful high-level characterization based on anecdotal evidence.

2.2 Existing file systems

To increase parallelism, all large multiprocessor file systems decluster blocks of a file across many disks, which are accessed in parallel. Most extend a traditional file abstraction (a growable, addressable sequence of bytes) with some parallel file-access methods. The most common provide I/O "modes" that specify whether and how parallel processes share a file pointer [7, 30, 33, 2, 17]. Some are based on a memory-mapped interface [23, 22]. Some provide a way for the user to specify per-process logical views of the file [5, 9]. Some provide SIMD-style transfers [34, 24, 16]. PIFS (Bridge) [11] allows the file system to control which processor handles which parts of the file, to encourage memory locality. Clearly, the industrial and research communities have not yet settled on a single new model for file access. Some aspects of the workload, therefore, are dependent on the particular file-access model provided to the user. The implications of this fact for our study are discussed in Section 5.

2.3 Multiprocessor file system caching

Caching and prefetching are successful in multiprocessor file systems [19, 20]. Pratt and French found that the caching and prefetching supplied with Intel's Concurrent File System (CFS) does improve performance [15]. Recent studies have found that CFS caching and prefetching work well in limited situations, but that the throughput of CFS can be disappointing relative to the capabilities of the hardware [27, 3]. Miller and Katz drove a cache simulation using traces from a Cray supercomputer and found that access locality was not high enough for significant benefits to be realized from a file system cache [25].

2.4 Intel 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 has a port for SCSI disk drives. There may also be one or more service nodes that handle as Ethernet connections or interactive shells [26].

Intel's Concurrent File System (CFS) [30, 15, 27] provides a Unix-like interface to the user with the addition of four *I/O modes* to help the programmer coordinate parallel access to files. 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. CFS stripes each file across all disks in 4 KB blocks. Compute nodes send requests directly to the appropriate I/O node. Only the I/O

nodes have a buffer cache.

3 Methods

To be useful to a system designer, a workload characterization must be based on a realistic workload similar to that which is expected to be used in the future. For our purposes, 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's Numerical Aerodynamics Simulation (NAS) facility met this criterion (their three newer multiprocessors, an Intel Paragon, a Thinking Machines CM-5, and an IBM SP-2 do not yet have a mature production workload). Their iPSC has 128 compute nodes, each with 8 MB of memory, and 10 I/O nodes, each with 4 MB of memory and a single 760 MB disk drive [26]. There is also a single service node that handles a 10-Mbit Ethernet connection to the host computer. The total I/O capacity is 7.6 GB and the total bandwidth is less than 10 MB/s.

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 try to factor out these effects as much as possible, we must note that some care should be taken in generalizing the results.

3.1 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. This means that any I/O which was done through standard input and output or to the host file system (all limited to sequential, Ethernet speeds) was not recorded. We collected data for about 156 hours over a period of 3 weeks. While we did not trace continuously for the whole 3 weeks, we tried to get a realistic picture of the whole workload by tracing at all 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. Since one of the goals of the CHARISMA project is to organize and facilitate a multi-platform file system tracing effort, we have defined a large set of event records suitable for both SIMD and MIMD systems [21].

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 (e.g., `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 multi-node 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 which was run periodically, and which 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 which was untraced from a job which 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 our instrumentation might affect the workload.

Our first concern was network contention. We expected users' jobs to generate a great many event records. Had we chosen to send a message to the data collector for each event record, we would certainly have created unreasonable congestion near the collector or perhaps in the overall machine. Since large messages on the iPSC are broken into 4 KB blocks, we chose to create a buffer of that size on each node to hold local event records. This buffer allowed us to reduce the number of messages sent by over 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 environment, it was imperative that the per-call over-

head 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 MB of data, our traces accounted for less than 1% of the total traffic.

Simple benchmarking of the instrumented library revealed that the overhead added by our instrumentation 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 [4]. After the instrumented library was put into production use, anecdotal evidence suggests that there was no noticeable performance loss.

3.2 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 KB 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 [14]. 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.

4 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. More detail may be found in [21].

4.1 Jobs

Figure 1 shows the amount of time the machine spent running a given number of jobs. For more than
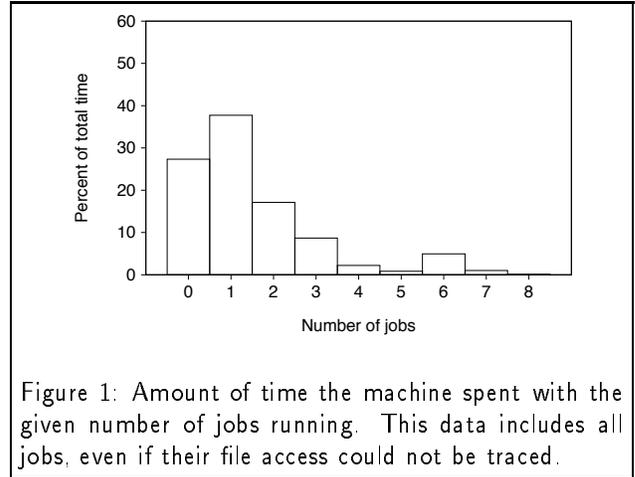

Figure 1: Amount of time the machine spent with the given number of jobs running. This data includes all jobs, even if their file access could not be traced.
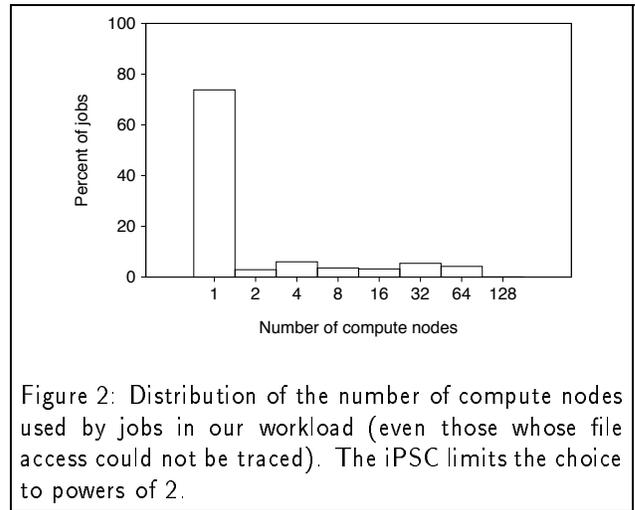

Figure 2: Distribution of the number of compute nodes used by jobs in our workload (even those whose file access could not be traced). The iPSC limits the choice to powers of 2.

a quarter of the traced period, the machine was idle (i.e., 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 multiprocessor file-systems research 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 the number of compute nodes used by each job. One-node jobs dominated the job population, although large parallel jobs dominated node usage. This dichotomy would be larger in new "self-hosting" parallel systems. 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.
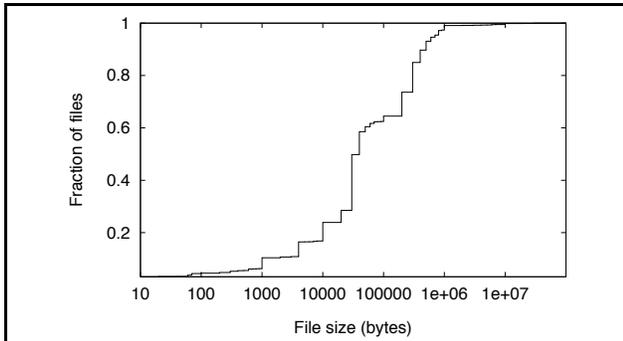
Figure 3: Cumulative distribution function (CDF) of the number of files of each size at close. For a file size x, CDF(x) represents the fraction of all files that had x or fewer bytes.

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

4.2 Files

During the 156 hours of tracing, almost 64,000 files were opened. Of those, 44,500 were only written to and 14,500 were only read from. The ratio of write-only files to read-only was surprising. It appears that the 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 MB) than average bytes read per file (3.3 MB). There were very few (less than 2300) files that were read and written in the same open. This behavior is also common in Unix file systems [12] and may be accentuated here by the difficulty in coordinating concurrent reads and writes to the same file (note the CFS file-access modes are of little help for read-write access).

Finally, there were nearly 2500 files which were opened but neither read nor written.

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

Number of Files	Number of Jobs
1	71
2	15
3	24
4	120
5+	240

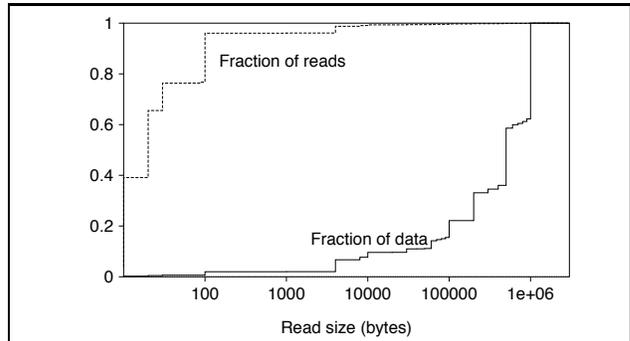Table 1 shows that most jobs opened only a few files over the course of their execution, although a few opened many files (the maximum was one job that opened 2217 files). Some of the jobs which opened a large number of files were opening one file per node. Although not all files were open concurrently, 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 (defined as 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 indicates that few applications chose to use files as an extension of memory for an "out of core" solution. Many of the Ames applications are computational fluid dynamics (CFD) codes, for which they have found that out-of-core methods are in general too slow.

Figure 3 shows that most of the files accessed were large (10 KB to 1 MB). It is important to note that each of the clusters of similarly sized files (e.g. at 25KB and 250KB) may be due to just one or two applications, so undue emphasis should not be placed on the specific numbers as opposed to the general tendency towards larger files. Although these files were larger than those in a general-purpose file system [1], they were smaller than we would expect to see in a scientific supercomputing environment [25]. We suspect that users limited their file sizes due to the small disk capacity (7.2 GB) and limited disk bandwidth (10 MB/s peak).

4.3 I/O request sizes

Figure 4 shows 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 than 4000 bytes, but those writes transferred only 3% of all data written (not shown). The number of small requests is surprising due to their poor performance in
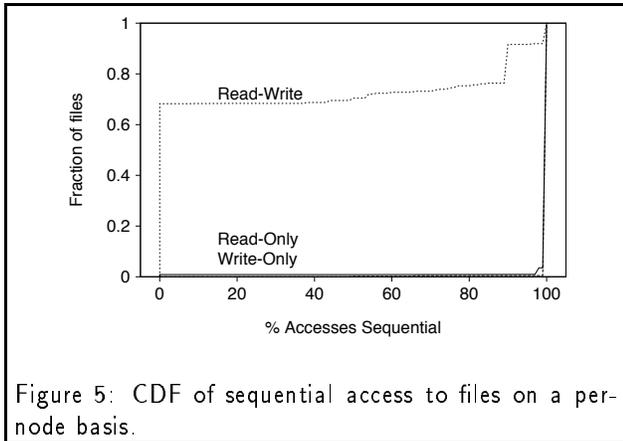
Figure 5: CDF of sequential access to files on a per-node basis.

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

CFS [27]. The small peak at 4 KB indicates that some users have optimized for the file-system block size, but it appears that most users prefer ease of programming over performance.

Figure 4 shows spikes in the number of small read requests as well as in the data transferred by 1 MB requests. While the spikes of small requests occurred throughout the tracing period, one trace alone (probably one job alone) contributed the spike at 1 MB. 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.

4.4 Sequentiality

A common characteristic of file workloads, particularly scientific workloads, is that files are accessed sequentially [28, 1, 25]. To grasp the notion of "sequential" access in a parallel application, 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 5 and 6 show the amount of sequential and consecutive access (on a per-node basis) 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 non-sequential. 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 (non-consecutive,
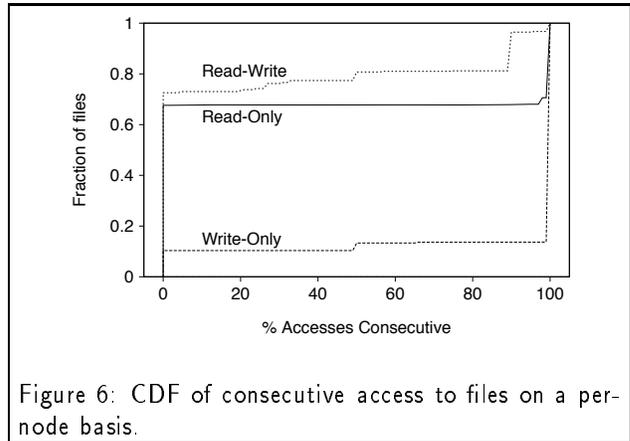
sequential read-only files) were the result of interleaved access, where successive records of the file are accessed by different nodes; from the perspective of an individual node, some bytes must be skipped between one request and the next.

4.5 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 2. A surprising number of files were read or written in one request per node (i.e., there were no intervals). Over 99% of the 1-interval-size files were consecutive accesses (i.e., the one interval size was 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.

Table 2: 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.

Number of different intervals	Number of files	Percent of total files
0	23291	36.5
1	37148	58.2
2	2561	4.0
3	105	0.2
4+	674	1.0

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 3. Over 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, presumably because much of the data was in matrix form.

Table 3: 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.

Number of different sizes	Number of files	Percent of total files
0	2480	3.9
1	25523	40.0
2	32779	51.4
3	2510	3.9
4+	487	0.8

4.6 Synchronization

Given the regular request sizes and interval sizes shown in Tables 2 and 3, Intel's "I/O modes" (see Section 2.4) 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 2 and 3 give one 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.

4.7 Sharing

A file is *shared* if more than one job or process opens it. It is *concurrently shared* if the opens overlap in time. 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 uncommon, and concurrent *write* sharing rare [1]. In a parallel file system, of course, 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 7 shows the percentage of files (which 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-
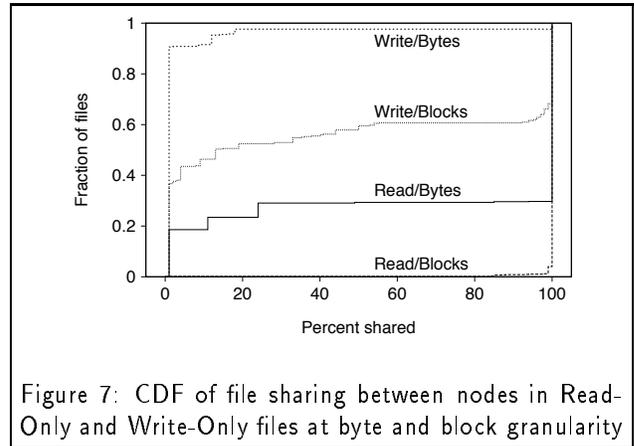

Figure 7: CDF of file sharing between nodes in Read-Only and Write-Only files at byte and block granularity

only files had no bytes shared at all. While a half of all read-write files were 100% byte-shared, 93% of them were 100% block-shared, which would stress a cache consistency protocol, if present. Overall, the amount of block sharing implies strong *interprocess* spatial locality, and suggests that caching may be successful.

4.8 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 RAID disk arrays commonly seen 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 Figure 4, 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 KB (one block), read-only buffers with LRU replacement are shown in Figure 8. We consider a *hit* to be any request that was *fully* satisfied from the local buffer (i.e., 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
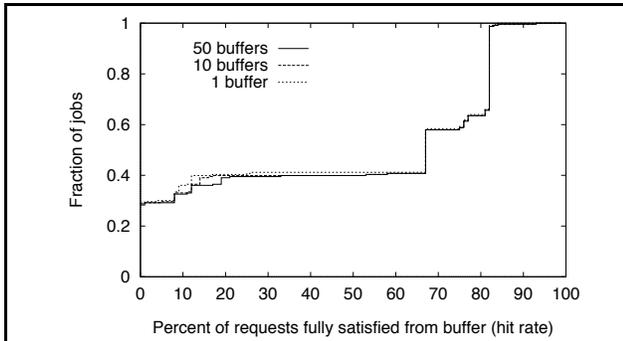
Figure 8: Results of compute-node caching simulation. Hit rates differed from job to job, with three distinct clumps, indicating that the cache either helped or did not. One buffer was as good as many buffers.
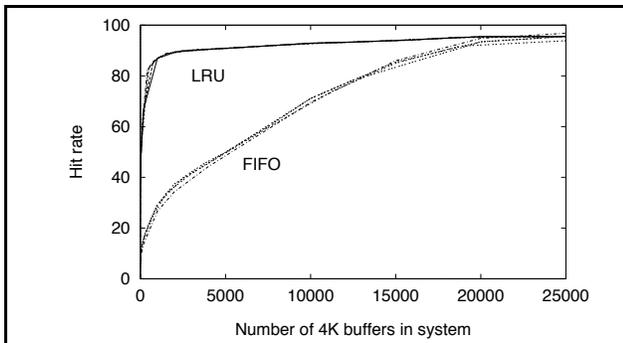

Figure 9: Results of I/O-node caching simulation. Each line represents a complete run of the simulation with a fixed number of I/O nodes ranging from 1 to 20.

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 Figure 4) 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 a very few jobs, apparently those which 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-KB buffers managed by either a LRU 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 as described in Section 3. We assumed the file was striped in a round-robin fashion at a one-block granularity. No compute-node cache was used. Figure 9 shows the results of the simulation. With LRU replacement, a small cache (4000 4-KB buffers over all I/O nodes) was sufficient to reach a 90% hit rate. With FIFO replacement, nearly 20000 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 I/O nodes (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 (Figure 8), 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 8 shows, the limited intraprocess locality was filtered out by the compute-node cache.

Note the contrast with Miller and Katz's tracing study [25], which found little benefit from caching. (They did notice 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.

5   Conclusions and recommendations

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 intra-job concurrent file sharing) and added some new characteristics (non-consecutive 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 partic-

ular, 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 RAIDs (which favor large transfers) rather than individual disks. Replacement policies other than LRU or FIFO should be developed (e.g., [19]), to optimize for interprocess locality rather than traditional spatial and temporal locality.

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, non-contiguous 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. Strided requests are available in some file-system interfaces [5, 9, 17]. For some applications, *collective I/O requests* can lead to even better performance [18].

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

Acknowledgements

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] M. L. Best, A. Greenberg, C. Stanfill, and L. W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 489–495, 1993.

[3] R. Bordawekar, A. Choudhary, and J. M. D. Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *International Conference on Supercomputing*, pages 367–376, 1993.

[4] R. Carter, B. Ciotti, S. Fineberg, and B. Nitzberg. NHT-1 I/O benchmarks. Technical Report RND-92-016, NAS Systems Division, NASA Ames, Nov. 1992.

[5] P. F. Corbett, D. G. Feitelson, J.-P. Prost, and S. J. Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, 1993.

[6] T. H. Cormen and D. Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Revised from Dartmouth PCS-TR93-188.

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

[8] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, 1993.

[9] E. DeBenedictis and J. M. del Rosario. nCUBE parallel I/O software. In *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, pages 0117–0124, Apr. 1992.

[10] J. M. del Rosario and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, Mar. 1994.

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

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

[13] R. A. Floyd and C. S. Ellis. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):238–247, June 1989.

[14] J. C. French. A global time reference for hypercube multiprocessors. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 217–220, 1989.

[15] J. C. French, T. W. Pratt, and M. Das. Performance measurement of the Concurrent File System of the Intel iPSC/2 hypercube. *Journal of Parallel and Distributed Computing*, 17(1–2):115–121, January and February 1993.

[16] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[17] D. Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[18] D. Kotz. Disk-directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994.

[19] D. Kotz and C. S. Ellis. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17(1–2):140–145, January and February 1993.

[20] D. Kotz and C. S. Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, Jan. 1993.

[21] D. Kotz and N. Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. Technical Report PCS-TR94-211, Dept. of Math and Computer Science, Dartmouth College, Apr. 1994. To appear in Supercomputing '94. Revised May 11, 1994.

[22] O. Krieger and M. Stumm. HFS: a flexible file system for large-scale multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[23] KSR1 technology background. Kendall Square Research, Jan. 1992.

[24] Parallel file I/O routines. MasPar Computer Corporation, 1992.

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

[26] NASA Ames Research Center, Moffet Field, CA. *NAS User Guide*, 6.1 edition, Mar. 1993.

[27] B. Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, Dec. 1992.

[28] 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.

[29] B. K. Pasquale and G. C. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of Supercomputing '93*, pages 388–397, 1993.

[30] 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.

[31] M. L. Powell. The DEMOS File System. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 33–42, Nov. 1977.

[32] 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.

[33] P. J. Roy. Unix file access and caching in a multicomputer environment. In *Proceedings of the Usenix Mach III Symposium*, pages 21–37, 1993.

[34] Connection Machine model CM-2 technical summary. Technical Report HA87-4, Thinking Machines, Apr. 1987.