Input/Output Behavior of Supercomputing Applications

Ethan L. Miller
Computer Science Division
Department of Electrical Engineering
and Computer Science
University of California, Berkeley
Berkeley, CA 94720
December 14, 1990
Revised October 16, 1992

ABSTRACT:

This paper describes the collection and analysis of supercomputer I/O traces and their use in a collection of buffering and caching simulations. This serves two purposes. First, it gives a model of how individual applications running on supercomputers request file system I/O, allowing system designers to optimize I/O hardware and file system algorithms to that model. Second, the buffering simulations show what resources are needed to maximize the CPU utilization of a supercomputer given a very bursty I/O request rate. By using read-ahead and write-behind in a large solid-state disk, one or two applications were sufficient to fully utilize a Cray Y-MP CPU.

1. Introduction

Over the last few years, CPUs have seen tremendous gains in performance. I/O systems and memory systems, however, have not enjoyed the same rate of increase. As a result, supercomputer applications are generating more data, but I/O systems are becoming less able to cope with this huge volume of information. Multiprocessors are exacerbating this problem, as the number of disks and tape drives in the I/O system, and thus aggregate I/O bandwidth, increase. Bandwidth is not usually scaled up at the same rate as the aggregate processing speed, however. According to Amdahl's metric, each MIPS (million instructions per second) should be accompanied by one Mbit per second of I/O. Solving this problem requires correct matching of bandwidth capability to application bandwidth requirements, and using buffering to reduce the peak bandwidth that the I/O system must handle. To better determine the necessary hardware bandwidth and software buffer sizing and policies, the I/O patterns of applications running on such computers must first be analyzed. To do this requires I/O access traces from real supercomputing applications, which we have gathered for the analysis in this report.

This paper first examines general file system characteristics, especially those important to supercomputers. Next, the applications which were traced will be described, followed by some analysis of the traces. Finally, we present the results of simulations of various methods, such as read-ahead and write-behind, to reduce peak and overall I/O demand for supercomputer file access patterns.

2. Overview

2.1. Conventional File Systems

Caching, the most effective method for reducing I/O bandwidth requirements, has been widely used in conventional file systems. It succeeds because of the properties commonly exhibited by many workstation and minicomputer applications, such as locality of reference in time and space [5]. For example, with a 2 MB cache on a VAX, only 17.7% of the applications' requests had to be fetched from disk. Prefetching data into a cache also reduces the instantaneous demand on an I/O system by spreading out demand and by predicting I/O references. This has the effect of reducing the number of requests, though not necessarily the amount of data transferred. In [5], sequential reads and writes accounted for over 90% of the accesses to files which were either read or written, but not both, and about 67% of total data transferred.

Another method of reducing I/Os from cache to disk is delayed writes. Delayed writes require a write-behind cache policy, which allows a program to continue executing after writing data to the cache without waiting for the data to be written to disk. In Sprite [4], data is not written back to disk for 30 to 60 seconds. Every 30 seconds, all data in the cache that is older than 30 seconds is written to disk, allowing the operating system to group all the writes. This allows temporary files which exist for less than 30 seconds, such as those generated by compilers, to be deleted and thus never written to disk. The 30 second delay is itself a file system parameter, and balances the reduced bandwidth to disk against the risk of losing data by not writing it to disk immediately. By totally eliminating disk I/Os associated with such files, the required bandwidth from cache to disk is reduced further.

These methods can be applied to supercomputer file systems, but there must be some changes to reflect the differences between interactive and small batch jobs run on smaller computers and the large vectorized applications run on supercomputers.

2.2. Supercomputer Environment

The production supercomputer environment is different from a conventional workstation and minicomputer environment. It is characterized by a few very large processes that consume huge amounts of memory and CPU time. Jobs are not interactive; instead, they are submitted in batch and run whenever the scheduler can find memory space and CPU time for them. This difference allows the scheduler better to plan usage of memory and CPU resources, as it has a relatively static queue of jobs to run, as opposed to the rapidly changing queue of interactive jobs. Such resource scheduling is often necessary, however, since many jobs require hundreds of megabytes of memory and hours of CPU time.

An example of a large supercomputing environment is the Cray Y-MP 8/832 at NASA Ames, the computer on which we traced several of the applications. This computer has eight processors, each with a 6 ns cycle time. The system has a total of 128 MW (each word is eight bytes long) shared among the eight processors. The I/O system has many high-speed disks, each capable of sustaining 9.6 MB/sec, totalling 35.2 GB, a 256 MW solid state disk (SSD) acting as an operating-system managed cache for a single filesystem (not the entire collection of disks), and several terabytes of nearline and offline tape storage [1,3]. The tape storage is divided into two parts—a nearline storage facility called the Mass Storage System (MSS), which can automatically mount tapes with requested data, and the extensive offline tape library which requires operator intervention. The NASA Cray system already has the maximum configuration of Y-MP memory (128 MW), so I/O problems cannot be alleviated simply by adding more primary memory.

The UNICOS process scheduling mechanism at NASA also affects the way programmers choose to structure their implementations, and thus I/O demands. Batch jobs, which include any program requiring over 10 minutes of Cray CPU time, are queued according to two resource requirements—CPU time and memory space. As the Cray Y-MP does not have virtual memory, all of a program's memory must be contiguously allocated when the program starts up, and cannot be released until the program finishes. To simplify memory allocation, each queue is given a fixed memory space. A job ready to run and residing in memory is run on any of the eight processors that is available. It runs until it must wait for a disk I/O, at which time it is suspended. This program remains in memory, and another program that is ready to run is given to the that processor. Since there are eight processors, there must be at least eight jobs in memory and ready to run to keep all of the processors busy. In practice, $n+1$ jobs resident in main memory will keep n processors busy, given a typical supercomputer workload [8]. To get these $n+1$ jobs, a single queue, especially one with low memory requirements, may have multiple applications in memory at the same time. Thus, for a given amount of CPU time required by an application, turnaround time is shortest for the application which requires the least main memory. Programmers take advantage of this by structuring their program to use smaller in-memory data structures while staging data to/from SSD or disk.

2.3. Supercomputer Applications

Because of their high-speed vector processing ability, supercomputers are ideally suited to problems that require manipulations of large arrays of data such as computational chemistry, computational fluid dynamics, structural dynamics, and seismology, to name a few. These problems all require large numbers of floating-point computations, which are usually vectorizable, over large data sets: from hundreds of megabytes up to tens or hundreds of gigabytes for some seismic computations. In most cases, the application performs multiple iterations over the data set, as when simulating a model through time.

3. Applications Traced

The first part of the study was an analysis of the I/O patterns of actual applications. We gathered traces from a variety of applications running on Cray computers, usually Y-MPs. We chose to trace applications with high I/O rates, both in megabytes per second and I/Os per second. While many supercomputer applications do not perform large amounts of I/O [8], we decided to concentrate on applications that do a lot of I/O. I/O-intensive applications stress the I/O system more, revealing performance bottlenecks. Those that perform little I/O are easy to characterize, as will be shown with the two traces that had low levels of I/O.

The traces fell into several categories. Most were computational fluid dynamics (CFD) problems, which are concerned with modeling the flows of fluids, such as water and air. However, each modeled different physical objects and made use of different algorithms. Several of the programs were climate models, while others modeled vortices around a moving blade. One program solved a structural dynamics problem, and one did polynomial factorization. In Table 1, we summarize

Application	**Running time (sec)**	**Total data size (MB)**	**Total I/O done (MB)**	**Number of I/Os**	**Avg. I/O size (MB)**	**MB/sec**	**IOs/sec**
bvi (CFD)	1258	171	22,835	1,380,457	0.016	18.2	1097
ccm (climate)	205	11.6	1,812	54,125	0.031	8.8	264
forma (structural)	206	30.0	15,155	475,826	0.030	73.6	2310
gcm (climate)	1897	229	266.2	7,953	0.031	0.14	4.2
les (large eddy)	146	224	7,803	22,384	0.317	53.4	153
upw (polynomial)	596		61.5	1,840	0.445	0.10	3.1
venus (climate)	379	55.2	16,712	34,904	0.032	44.1	92

Table 1. Characteristics of the traced applications

some basic information about the applications. **Running time** is the amount of CPU time each program required. All of the other numbers are relative to this time, not elapsed wall clock time. **Total I/O done** is the total amount of data the program read and wrote, and **number of I/Os** is the number of read and write calls the program made to the file system. The total size of the data set, which was the sum of the sizes of all the files the program accessed, is listed under **total data size**.

The first group is the climate models. These included **gcm** (Global Climate Model), **ccm** (Community Climate Model), and **venus** (a simulation of Venus' atmosphere). These were all CFD models which simulated atmospheres.

The major differences between the atmosphere models were the sizes of the data arrays in the simulations, the methods used to actually implement the algorithms, and the tradeoff each algorithm made between main memory size and I/O system usage. **Gcm** was primarily an in-memory simulation—the only data that went through the operating system were final results. The data fit into a main memory array, obviating the need to stage data from disk. As a result, the program did few I/Os. The **venus** code went to the other extreme. To get into a shorter job queue, the program's implementor decided to use a very small in-memory array. Thus, the program accessed the file system frequently to stage the required data to and from memory. **Ccm** took the intermediate point between the two, requiring fewer megabytes per second of program execution than **venus** but far more than **gcm**, probably because its in-memory data array was intermediate in size between the other two programs'.

The **bvi** (blade-vortex interaction) program was also a CFD program, but it simulated the interactions of a helicopter blade with the air around it. It was the only one of the programs traced explicitly designed for use with the SSD (solid state disk) on the Cray. Since the SSD has zero seek time and a very high transfer rate, the program did not suffer a major performance loss from the many small I/Os it made. I/Os to and from the SSD are done without suspending the process requesting the I/O, because the data is retrieved quickly. However, as will be discussed later, the file system overhead may have slowed the program down by using more operating system time. This added a sizable penalty, more than would be incurred for a large request replacing several small ones.

The **les** (large eddy simulation) application used the Navier-Stokes method with turbulence. This algorithm only calculates large-scale effects from the Navier-Stokes equations and directly models the small-scale effects. A more complete description of the algorithm is beyond this paper, but one can be found in [6].

Upw (approximate polynomial factorization) did the least I/O of any application traced. This program read a small input file, computed for ten CPU minutes, and wrote out an answer. It is an important program, however, since this is a representative I/O pattern for some applications. The program infrequently requests a few large I/Os.

The last program traced was called **forma**. This program was originally written for a Cray 1, with its small memory, and uses sparse matrices to solve structural dynamics problems. In this program, I/O serves a secondary purpose beyond simply staging data in and out. By breaking up the data array into blocks, empty blocks can be easily identified and created in memory instead of being staged in. Thus, there is a secondary tradeoff between I/O size and required bandwidth. A larger block would allow more efficient I/O requests, but it also might require more I/O bandwidth, since a non-empty subarray of size 2N x 2N might contain three empty N x N arrays. The 2N x 2N subarrays would then require four times the data rate of the N x N subarrays. The programmer seems to have chosen a relatively large access size despite the possible advantages of using a smaller one.

4. Tracing Methods

4.1. Information Traced

The traces gathered included two types of information. First, they recorded file and disk reference information, so the pattern of references to the file system (for logical-level traces) and physical disk sectors (for physical-level traces) could be reconstructed. File identifiers corresponded to file opens; if the same file was opened twice by a program, it received two different identifiers. Second, timestamps were taken for each I/O. There were three timestamps for each I/O event. The first timestamp was total elapsed wall time, which was obtained from a timer register in the CPU. This value was in units most convenient to the system; for the Cray Y-MP, it was in 6 ns clock ticks, as there is a counter in the CPU which is incremented every clock cycle. For traces in our standard format, this value was converted to 10 s units, as we believed this was sufficient time resolution for I/O traces. The second timestamp measured the wall clock time between when the I/O request was made by the application and when the completion status was returned. This timestamp might have been affected by the scheduler, since a program that waits for I/O is not guaranteed to be restarted immediately when the I/O completes. The third timestamp was process elapsed time, indicating the amount of CPU time the particular process had been running. Thus, the effects of multiprogramming could be filtered, as the process elapsed time between I/O events would be constant no matter how often the process was swapped out.

4.2. Trace Format

The I/O accesses the applications made were all recorded in a standard trace format that was designed to be used for both logical and physical I/O traces. The format was also designed with trace compression in mind, as mentioned in [7]. This section gives a high-level of the format; for details, see the appendix.

Figure 1 shows a sample trace record. Compression techniques worked especially well for supercomputer traces for two reasons—file accesses were highly sequential, and a very large majority of the accesses went to only a small number of files. Both of these characteristics will be discussed in more detail later.

Flags	Compression	Offset	Size	Start Time	Elapsed Time	File ID	Op ID	Proc. ID
FE	0	0	10	5AF8	32C	1D	89	743E

Flags - logical/physical I/O, read/write, synchronous/asynchronous
Compression - information about which fields can be calculated from previous records
Offset, size - where in the file (disk) the I/O took place
Start time - start of this I/O relative to the start of the previous I/O
Elapsed time - duration until completion of this I/O
File ID - identify file (disk) the I/O occured on
Op ID - unique for each call to the file system
Proc ID - process identifier

Figure 1. Information in a trace record.

To save disk space and trace-gathering time, the traces were compressed in two ways. First, some fields could be specified relative to the record immediately preceding it. These fields included the timestamp fields and the file identifier field. Instead of recording a full 8 or 9 digit time, only the elapsed time since the last record was recorded. A bit in the compression field was set if the file identifier was the same as in the previous record. The second method of compressing data was to record I/O sizes, I/O lengths, and process identifiers relative to the last I/O made to that file. Again, a few bits in the compression field could indicate that the I/O was sequential with the last I/O to that file, or that the I/O was the same size as that file's last I/O. In this way, the trace of a program which made interleaved accesses to several files, such as **venus**, was still compressed efficiently.

While we only collected logical-level trace data on the Cray, we included provisions for our trace format to include physical I/Os as well.

4.3. Trace Gathering Methods on the Cray Y-MP

All of the data on the Cray Y-MP is logical-level traces. This data included logical file numbers, file offsets, request sizes, and wall clock and process clock timestamps. Because no collected data was internal to the operating system (as physical disk block numbers would be), all the data could be collected by code running at user level. Thus, no modifications to the operating system were necessary. This was a distinct advantage on the Cray, since it would have been very difficult to obtain the amount of dedicated time necessary to debug changes to the operating system.

Instead of modifying the operating system, we changed the user libraries dealing with I/O. Cray provides data collection hooks in standard system libraries shipped with Unicos 5.0. These hooks merely provide aggregate data on I/O, such as the total number of bytes a process requested and the average and maximum times to do an I/O. In addition, major events such as file opens, closes and process forks are traced by the standard Cray software, though we did not use the data in these trace packets except to check some of our results. Trace packets are sent to a process on the Cray called **procstat**. The **procstat** process collects these trace records, which include an 8 word header and whatever data is necessary for the system call being recorded, and writes the records to a trace file for later analysis. A diagram of the path trace information takes is shown in Figure 2.

Merely modifying the libraries to produce one trace record per **read** and **write** call would have produced far too much data. The trace record headers are large compared to the amount of data recorded per call, between three and five words. Operations on each file were sent in batches, so one header served for hundreds of I/O calls and the header overhead was amortized over many calls. In addition, trace packets were forced out every hundred thousand I/Os. This was done since each packet recorded data for just one file, and a file with little I/O, such as a parameter file, might have two I/Os separated by hundreds of thousands of I/Os to a data file. Reconstructing a single stream of all the accesses from the file of packets requires buffering all the I/Os between flushes, since a

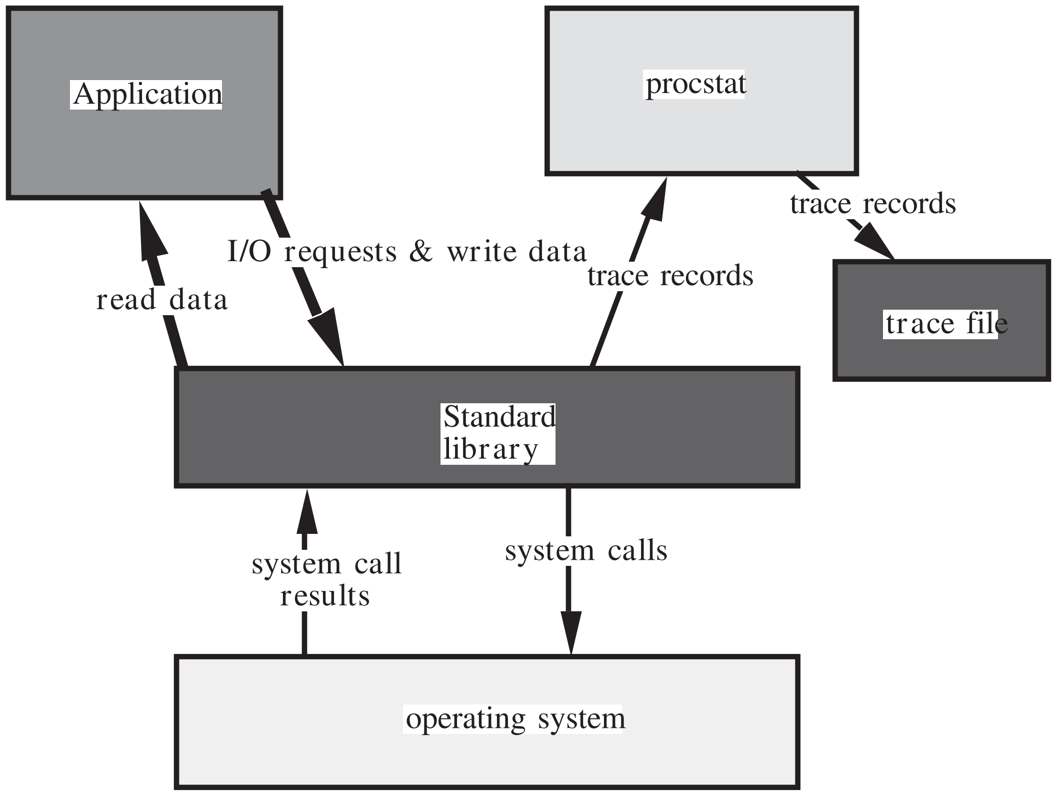

Figure 2. Trace information flow between software modules.

packet written during the flush might contain an I/O access from much earlier in the program's execution.

There was very little CPU overhead required to collect traces in this manner. There was no overhead during non-I/O operations because the tracing mechanism was only active during an I/O call to the operating system. Also, the amount of code executed per I/O was small compared to the code normally executed by the operating system to handle an I/O request. Overheads were less than 20% of I/O system call time, and total overhead was completely dependent on the amount of I/O done by the application.

While this trace collection method is standard on UNICOS, and vendor-supplied software contains most of the code necessary for tracing, the same method could easily be used to instrument standard libraries on other computers and collect traces from applications running on them. The only major system requirement is an accurate clock. An operating system which supports Unix-style pipes would also make it easier to implement the trace collection software because we used pipes to pass trace data from applications to **procstat**.

5. I/O Pattern Analysis

There has been much analysis of overall supercomputer performance in both the I/O and CPU usage. However, the I/O usage studies have focused primarily on overall system performance over relatively long periods, ranging from many minutes to several weeks [8]. While these studies are very useful for analyzing current CPUs, inferences from current systems to future systems may be difficult because parameters change in different ways. For example, larger or smaller memory systems relative to CPU speed will certainly affect overall system performance, but an accurate picture requires examining individual applications and their interactions under new system parameters.

5.1. Types of Application I/O

All of the I/O accesses made by the programs can be divided into three types—required, checkpoint, and data swapping. Required I/Os are similar to hardware cache misses called *compulsory* in [2]. These consist of I/Os that must be made to read a program's initial state from the disk and write the final state back to disk when the program has finished. For example, a program might read a configuration file and perhaps an initial set of data points, and then write out the final set of data points along with graphical and textual representations of the results. These I/Os, however, do not contribute much to the overall I/O rate. For a program which runs for only 200 seconds, reading 50 MB of configuration and initialization data and writing 100 MB of output, the overall I/O rate is only .75 MB/sec. This rate is easily sustainable by most workstations, and certainly does not demand complex solutions. While the peak I/O rates at the start and end of the program will be high, they will only occupy a small fraction of the total running time of the program. **Upw** and **gcm** are examples of programs that only do compulsory I/O.

Checkpoints, the second type of I/O, are used to save the state of a computation in case of a hardware or software error which would require the simulation to be restarted. A checkpoint file generally consists of some subset, possibly complete, of the program's in-memory data. Checkpoints are generally made every few iterations, though making them too often slows the program down unnecessarily. The application writer balances the cost of writing the checkpoint against the cost of redoing lost iterations of the simulation. The likelihood of failure determines the number of iterations between checkpoints. Since checkpoints occur multiple times per program, they add more to the bandwidth requirement than required I/O, but they also do not place a continuous high demand on the I/O system. For a program that saves 40 MB of state every 20 CPU seconds, the average I/O rate is only 2 MB/sec, far less than the maximum rate most supercomputers provide. As with required I/Os, dealing with peak rates may present a problem, but since the I/Os occur relatively infrequently, it is easy to have another program ready to run (and not in the checkpoint stage itself) while the first program is waiting for checkpoint I/Os to complete.

The third type of I/Os are those done because the memory allocated to the problem is insufficient to hold the entire problem. These I/Os are the equivalent of paging under a paging virtual memory operating system, but they are generally done under program control because many supercomputers lack paging. Even when paging exists, the program is better able than the operating system to predict which data it will need. Unlike the other two types of I/O above, memory-limitation I/O must be done on every iteration of the algorithm. The entire data set is usually shuttled in and out of memory at least once, and perhaps more often. If each data point consists of 3 words and requires 200 floating-point operations, there must be 24 bytes of I/O for every 200 FLOPS (this is quite close to Amdahl's metric, which would require 200 bits, or 25 bytes of I/O for those 200 FLOPS) . For a 200 MFLOP processor, the average sustained rate will be almost 25 MB/sec, far more than either the compulsory I/O data rate or the checkpoint I/O data rate. Peak rates are higher still, and in fact are higher than 200 MB/sec of requests sustained over several CPU seconds.

5.2. I/O Access Characteristics

The I/O accesses that the applications make can be characterized in several ways. These included the total amount of I/O, the read/write ratio both overall and for given files, and the size of each individual I/O, again overall and for each file. In looking at these characteristics, however, only “large” files were considered. In most cases, these files were over a few megabytes long, and some were hundreds of megabytes long. While “small” files, which include parameter files and human-read text output, are important, they do not contribute much to the overall I/O that a supercomputer application must do, as their contribution is dwarfed by accesses to large machine-generated data files.

All of the programs, with the exception of **gcm** and **upw**, made many read and write accesses, and did many I/Os, as can be seen by Table 2. These numbers are per second of CPU time used by the process.

The only applications which had read/write ratios much under one were **gcm** and **upw**, as can be seen in Table 2. They were the programs that did not do much I/O in the first place, since they did

Program	**Reads (MB/sec)**	**Writes (MB/sec)**	**Reads (IOs/sec)**	**Writes (IOs/sec)**	**Avg I/O size (KB)**	**Read/Write ratio (data)**
bvi	12.3	5.34	913	185	16.1	2.31
ccm	4.25	3.96	135	128	31.9	1.07
forma	62.2	5.68	1990	300	30.4	11.0
gcm	0.0107	0.12	0.34	3.85	31.9	0.089
les	24.0	25.2	74	81	325	0.95
upw	0.0012	0.0100	0.037	3.05	32.7	0.12
venus	26.4	14.7	60	32	456	1.80

Table 2. I/O request rates and data rates of the traced applications.

little I/O other than compulsory writes. The programs that did higher amounts of I/O had higher read/write ratios because, for those programs, the disk was used to hold large parts of the array. For each cycle of the algorithm, each section of the data is written once. However, that data may be read more than once so it can be used in the computation in different places. This pattern will remain no matter how large the memory of the system gets, since a larger memory will simply encourage larger problems, which will keep the same patterns. A file cache will not greatly change the read/write ratio to disk. The files are usually so large that they will not fit into the cache. Since the entire file is both read and written each iteration, there are no “hot” blocks that can remain in the cache between iterations. A cache might, however, decrease the read/write ratio to disk slightly because "paging" the data array might show spatial locality for reads.

Access size varied between programs, but was relatively constant within programs. The access size was completely under the programmer's control, so it varied according to how the algorithm was implemented. As seen in Table 1, accesses on the large files ranged from 32 KB to 512 KB. The notable exception was **bvi**, which used the SSD for most of its "disk" accesses. There was no seek penalty for the SSD, so the small I/O penalty was much less than it would be for a normal disk. A SSD access still paid operating system overhead and transfer time, but it did not incur any latency as a disk access would.

5.3. Cycles in Program I/O

Since all of the programs implemented iterative algorithms, the programs' I/O patterns followed cycles that matched the iterations of the program. Often, the data in the files would be read in the same sequence and with the same I/O request size each cycle. Even when the sequence was not the same between cycles, each program had a typical I/O request size which stayed constant throughout the program. Times of high data request rates also followed a pattern; request rate peaks were generally evenly spaced through the program's execution.

I/O was bursty, as expected, but the bursts came in cycles. The demand patterns for all of the cycles in a single application were remarkably similar, as Figures 3 and 4 show.

File reference patterns also followed cycles. This was especially true for algorithms that operated on an unchanging array that was larger than the program's memory size. For such applications, the reference patterns were essentially identical from cycle to cycle. For other applications, the array might change between iterations of the algorithm. For example, a common method in a CFD problem is to create more data points in areas of interest for detailed examination. Since these areas cannot be predicted in advance, the program itself identifies the areas and creates more points,

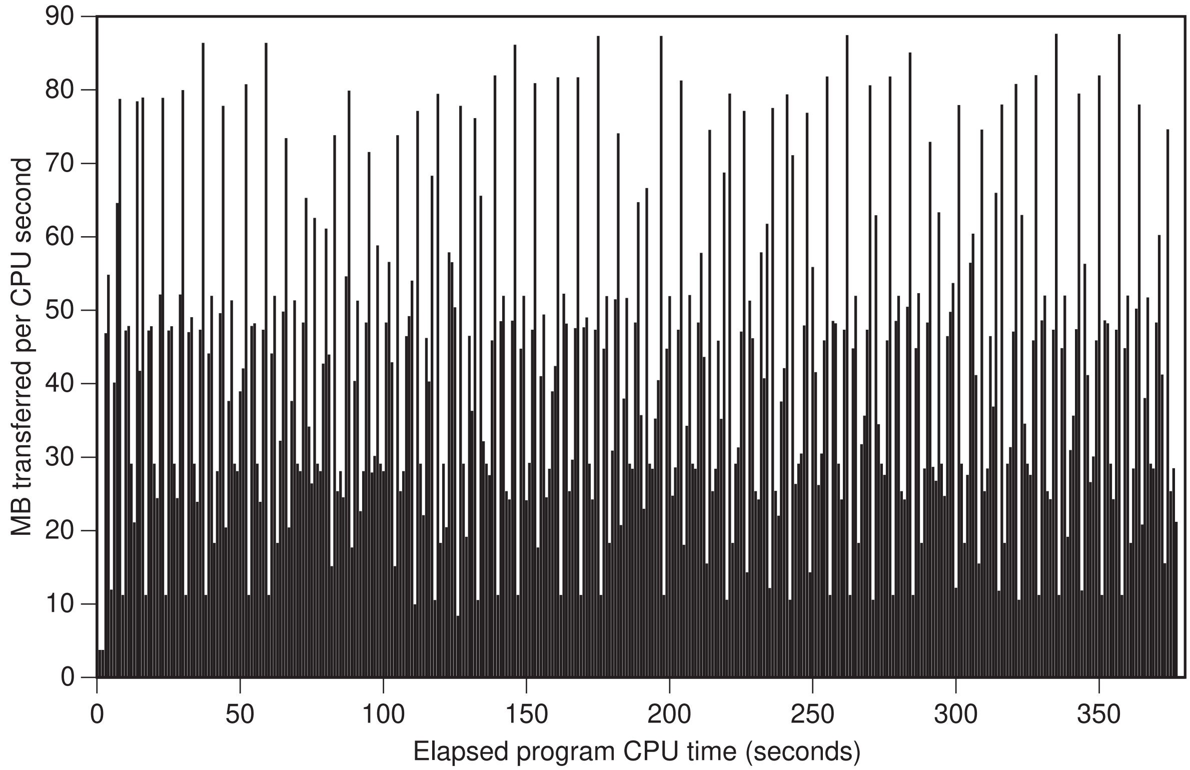

Figure 3. Data rate over time for **venus**.

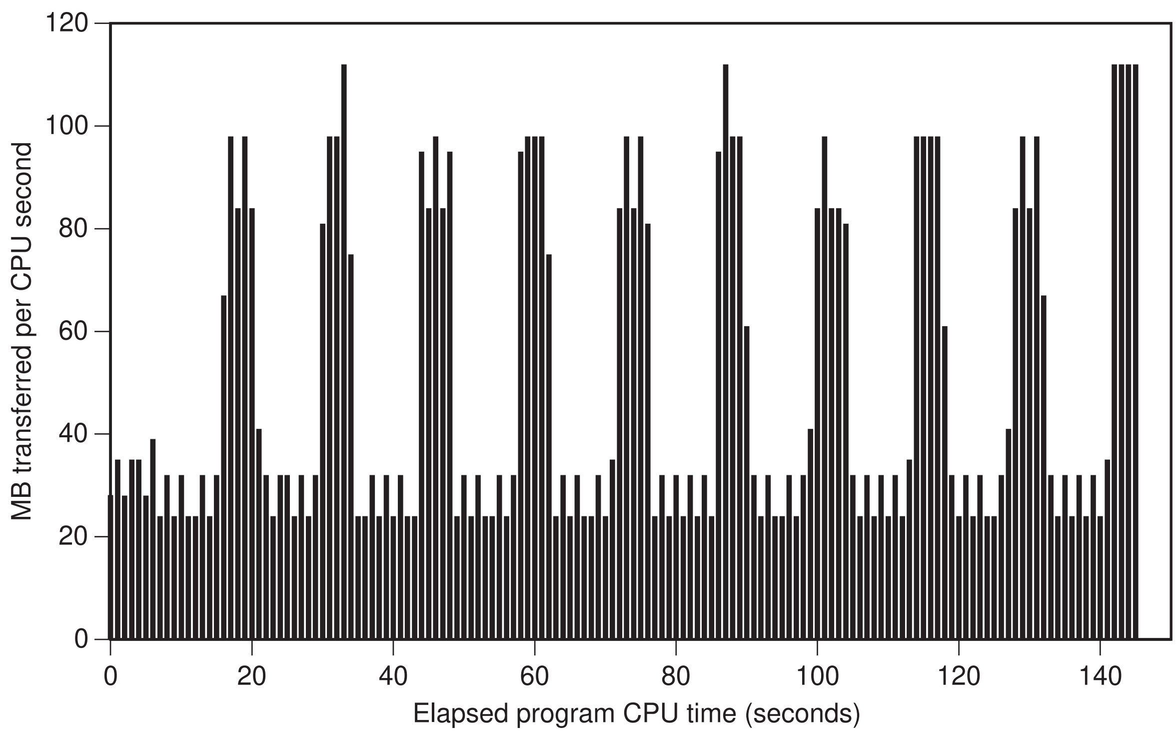

Figure 4. Data rate over time for **les**.

changing the data array and the disk reference patterns.

6. Caching Simulations

The traces collected from the applications can, by themselves, show the behavior of an individual program. However, a supercomputer rarely runs one job per processor even when in batch mode. CPU cycles would go unused if there were no additional programs to run because an application often must wait for a disk access to complete, and all programs do some I/O. On a typical Cray Y-MP system there are usually few enough I/O requests that $n+1$ programs are sufficient to avoid wasted cycles on n processors. This rule of thumb requires that programs fit their entire data array in memory, since any program that must use the disk to store its data will do large amounts of I/O each cycle. If all currently in-memory programs make many I/O requests, it is likely that more than one will be awaiting I/O all the time.

6.1. Cache Simulator

We constructed a cache simulator that models the behavior of a single CPU with multiple processes making I/O requests. For each process, there is an input trace in our format, which determines the size of each I/O and the elapsed time between it and the next I/O. Using this information, a simple scheduler built into the simulator, and a simple disk model, the overall sequence of I/Os that the processes will make can be simulated. The position of our buffer simulator relative to the entire file system is shown in Figure 5.

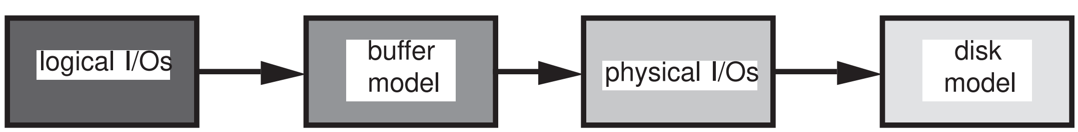

Figure 5. Path an I/O takes from application request to disk.

The simulator uses a simple round-robin scheduler with a quantum that can be specified each time it is run. The process-switching overhead, file system code overhead, and interrupt service time are also parameters that can be set in the simulator.

The disk model, like the scheduler, is a simple one. Since ours were logical traces and we did not model the file system, we could not use physical block numbers. Thus, seek times could only be approximated. There was no queueing at the disks, so the completion time of a specific I/O was dependent only on the location of the I/O and how “close” the I/O was to the previous I/O. This simplification significantly affected our results, as will be shown later.

6.2. Main Memory Buffering

The first set of simulation runs involved file caches that were small enough to fit in a Y-MP's main memory. On a standard Cray, the file system cache is shared among all the processors; however, we were only modeling one processor. To avoid modeling the dynamic division of the file cache between the processors, we restricted the cache size to a fraction of the memory “allocated” to a single processor. For example, in a system with 128 MW of memory, the file system cache might take up between 4 MW and 16 MW of memory—5% to 12%. This would be distributed among eight processors, though, so each processor could only use 1/8th of the available cache, assuming all were running I/O-intensive jobs. In this example, each processor would be limited to 0.5 to 2 MW of file cache space.

Very few of the applications traced had I/O that fit into such a small cache. This, combined with the sequential nature of the programs’ I/O, meant that most logical I/Os resulted in disk accesses. This is in contrast to the study in [5], that reported that up to 80% or more of the I/Os could be satisfied in a file cache. In a supercomputer, a main-memory file system cache is thus used more as a speed-matching and load-averaging buffer than it is to exploit access locality.

We used two techniques to decrease idle time for a given set of processes, thus increasing CPU utilization. The first method was prefetching data from disk. Its success was not unexpected, as [5] showed that prefetching was extremely useful. Because supercomputer I/O is both regular and sequential, it was easy to predict the next data bytes the program would request. In several of the programs, including **les**, an I/O request was not only sequential with the previous I/O, but was also the same size. Thus, prefetching the amount of data just read allowed the application to continue without waiting, but did not fill the cache with data that would be unused for some time.

The second method used was write-behind. While Sprite's delayed writes [4] would have been difficult to implement in the simulator, it was easy to allow a process to continue executing while written data had not yet gone to disk. This would also be easier to implement on a supercomputer. A separate process would be required to check all cache data and decide which of it goes to disk, but the per-process overhead is high on many supercomputer operating systems because of the large state which must be saved on process context switches. In Unix-like workloads, delayed writes can often result in temporary files being deleted from the cache before they must be written to disk [4,5]. However, most data written to a supercomputer's main memory file cache must go to disk because iterations take hundreds of seconds and files are hundreds of megabytes long. There is therefore little advantage to waiting a short time to see if data is deleted.

The main goal of using write-behind and prefetching is to reduce CPU idle time, given a set of processes executing and requesting I/O. Ideally, there should be no idle CPU cycles, and several of our simulations approached that with just one or two I/O intensive programs running at the same time. The program that came closest to fully utilizing a CPU while doing large amounts of I/O was **les**, since it was the only program that used asynchronous reads and writes explicitly. Clearly, its designer spent much time optimizing it for the Cray Y-MP system. **Venus** was another program benefiting from write-behind, though not as much from prefetching. In this program, the short cycles of reading and writing several relatively small files required over 40 MB/s of bandwidth to disk. While the disks were certainly capable of this rate, the seeks required by interleaving accesses to six different data files inserted extra delays. With write-behind, the delays did not affect the programs' running time as much. For example, writebehind reduced idle time from 211 seconds to 1 second for a simulation of two identical copies of **venus** running with a 128 MB cache.

Read-ahead and write-behind did not have all the effects we expected. We had expected that the peak demands on the disks would decrease, and the I/O request rate would remain relatively constant over the execution time of the program. As can be seen from Figure 6, this did not happen for several reasons. First, the simulator did not slow down disk access times when the disks had many outstanding requests, as would happen in a real system from queueing delay. Because the requests were logical file requests, it was impossible to map requests to individual disks for queue-

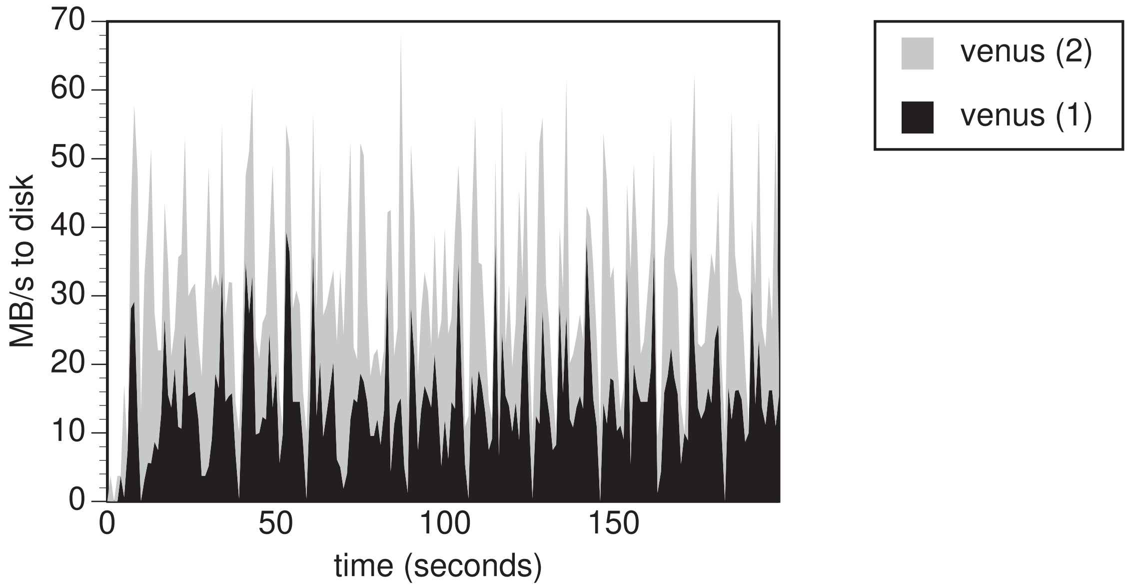

Figure 6.
Data rate for 2 simultaneously running copies of **venus** with a 32 MB cache.
(first 200 seconds of wall time)

ing, so we used a constant access time distribution that did not depend on the number of currently outstanding I/Os in the disk system. The second reason the request rate was not smoothed out was bunching at times of high I/O request rates. Ideally, the programs should have their periods of high I/O rate arranged in such a way that the high I/O rate period of one program comes during the computation phase of another program. Often, though, the two programs would both wait for I/O at the same time, such as when one program stops to request a large transfer of uncached data. Such a transfer might take as long as 15 ms (the Cray Y-MP disks seek relatively slowly). The other program might then make a similar request, requiring 15 ms as well. Both requests would finish at approximately the same time, and the process would repeat. In this way, the large seek and rotational delays might not be covered by computations in other processes, and the requests would be unevenly spread out over time.

Another problem that occurred when two high-I/O programs ran simultaneously is that one of the programs grabbed most of the buffers. This denied the other program a chance to do much I/O and use the CPU while the first program was waiting. A limit on the number of buffers a process could own did not provide relieve the problem, and actually worsened CPU utilization in several cases. The disadvantage of artificially slowing down the process that was doing large amounts of I/O did not outweigh the advantage of allowing multiple processes to run .

6.3. SSD Buffering

While supercomputers do some caching in their main memory, there is far more space available on the SSD, which cannot be directly used as program memory. In addition, SSDs are built from less expensive DRAMs, instead of the expensive SRAM used in the Cray Y-MP's memory. Currently, UNICOS 5.0 allows two options for using the SSD—system-managed buffers or user-managed buffers. The advantage of the latter is that the user has more knowledge of which data to stage from disk. However, managing that staging is a programming problem which many supercomputer application programmers do not want to undertake. A system which uses this approach may therefore find the SSD underutilized. While system-managed buffers in the SSD are less efficient than user-managed buffers might be, they are considerably easier for the average user, as they require no extra programming effort.

To simulate the SSD on the Cray Y-MP, we treated it as a huge main-memory cache, and added per-block penalties for cache hits. These were approximately 1 s per kilobyte transferred (at 1 GB/sec), with some additional overhead to set up the transfer. These times were relatively small compared to the time required to execute a system call.

Several of our traces had small enough data sets that they fit into the SSD entirely. For these programs, there was little or no idle time, as data was read from disk once and written back while the program continued executing. Figure 8 shows an example of this, with two identical **venus** programs running on the same CPU and not sharing data sets. **Venus**, **bvi**, and **ccm** all ran with low idle times in the SSD. **Gcm** and **upw** had low idle times in all of our simulations because they did so few I/Os—even in an 8 MB cache, **gcm** had only 1 second of idle time. Since **les** ran with little idle time on both the SSD and main-memory cache (because of explicit asynchronous I/O), all but one of the applications nearly completely utilized a Cray Y-MP CPU by itself when using a 32 MW SSD cache by itself. The Cray Y-MP at NASA has a 256 MW SSD, so each processor's share is 32 MW. Almost all of the read requests were satisfied by the SSD, so there were very few disk

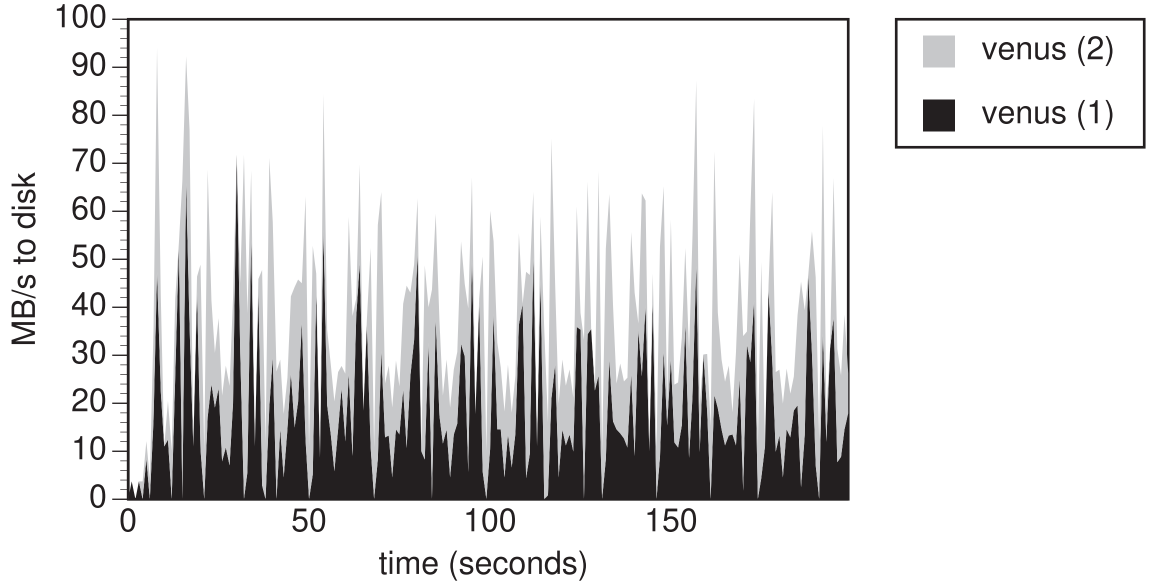

Figure 7.
Data rate for 2 simultaneously running copies of **venus** with a 128 MB cache.
(first 200 seconds of wall time)

read requests. However, as can be seen from Figure 7, the writes from cache to disk still did not come evenly; instead, they were bursty in the same way that the requests to cache were bursty.

6.4. I/O System Configuration

The best configuration for an I/O system, according to our simulations, is to provide as much SSD storage as possible, and maintain a smaller main memory cache. The largest main memory cache we believed would be reasonable, a 4 MW cache in a processor's allotment of 16 MW, still did not allow most I/O-intensive programs to execute without waiting for I/O, even with read-ahead and write-behind. The cache did not have enough buffer space to allow full read-ahead and write-behind to relatively slow disks. Figure 8 shows the effects of cache size on the total execution time of two simultaneously running **venus** programs.

SSD, on the other, appears to be a much better solution. In a 32 MW SSD, all of our programs except one utilized the CPU over 99%. SSD is more likely than main memory to scale with processor speed, since the constraints on an SSD memory's speed, physical size, and distance from the CPU are less likely to be affected by designing for a faster CPU. An SSD is appropriate for a

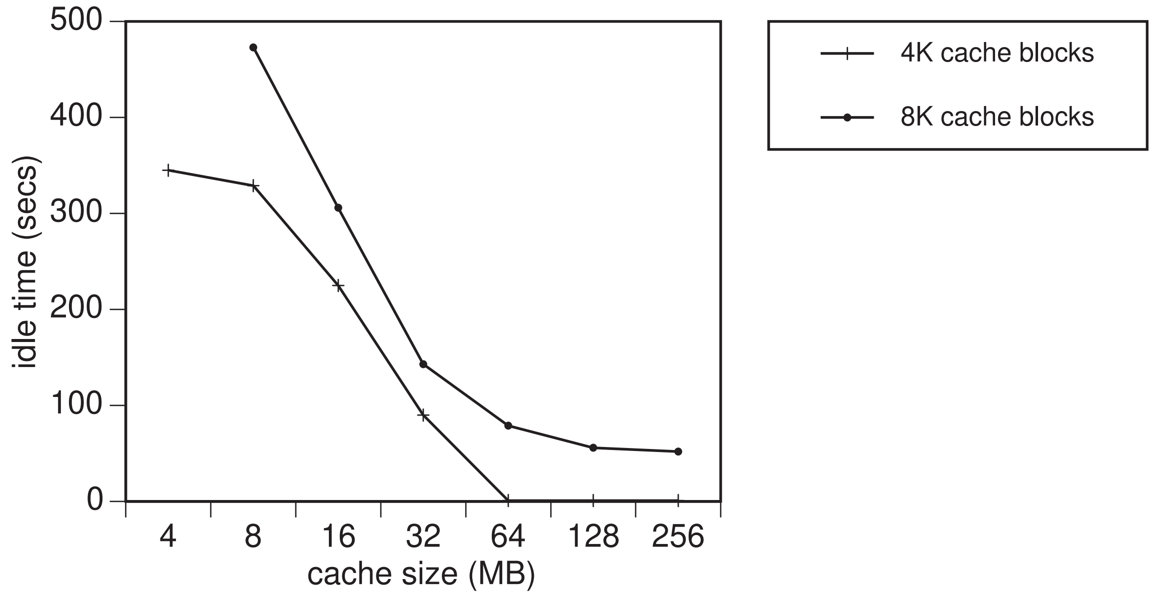

Figure 8.
Idle time while running two instances of **venus** with varying cache sizes. Execution time would be 761 seconds if there were no idle time.

multiprocessing environment as well; since the SSD communicates like a disk, multiple processors can access it in file-block-sized chunks instead of word by word, as main memory is accessed. Instead of low latency channels required for main memory, higher latency channels like those used for network communications can be used.

7. Conclusions

While much attention has been given to CPU performance in supercomputers, the I/O system, which includes the file cache, SSD, disks, and tape storage, will play an increasingly larger role in utilizing the CPU efficiently. We have examined several high-I/O demand supercomputer applications and shown that they are highly sequential and very regular in their access patterns. This information can be used to better design a supercomputer I/O system to fully utilize a supercomputer CPU, as our buffering simulations show. With a large SSD, only one or two processes per processor are needed to keep the CPU fully utilized. While main memory sizes may not scale as fast as processor speed, SSD sizes may scale more closely, since the constraints on physical size, distance from the CPU, and small access speed are not as stringent for an SSD. By implementing read-ahead and write-behind in a supercomputer's file system and using a solid-state disk, a few very large processes staging data to and from secondary storage can keep supercomputer CPUs busy.

References

[1] Bonifas, C. Searching For a Unix Mass Storage System For a Supercomputer Environment. In *Tenth IEEE Symposium on Mass Storage Systems,* 1990, pp. 129-133.

[2] Hill, M.D. *Aspects of Cache Memory and Instruction Buffer Performance*, Ph.D. dissertation, (published as Technical Report UCB/CSD 87/381), , November 1987.

[3] *NAS User Guide, Version 5.0*, NAS990.

[4] Nelson, M.N., Welch, B.B., and Ousterhout, J.K. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, Vol. 6, No. 1 (February 1988).

[5] Ousterhout, J.K., Da Costa, H., Harrison, D., Kunze, J.A., Kupfer, M., and Thompson, J.G. A Trace-Driven analysis of the UNIX 4.2 BSD File System. *ACM Operating Systems Review*, Vol. 19, No. 5 (1985), pp. 15-24.

[6] Peterson, V.L., Kim, J., Holst, T.L., Deiwert, G.S., Cooper, D.M., Watson, A.B., and Bailey, F.R. Supercomputer Requirements for Selected Disciplines Important to Aerospace. *Proceedings of the IEEE*, Vol. 77, No. 7 (July 1989), pp. 1038-1054.

[7] Samples, A.D. Mache: No-Loss Trace Compaction. Tech. Rept. UCB/CSD 88/446 University of California, Berkeley, 1988.

[8] Williams, E., Myers, C.T., and Koskela, R. The Characterization of Two Scientific Workloads Using the Cray X-MP Performance Monitor. In *Supercomputing '90,* 1990.

Appendix

This appendix gives a detailed description of our trace format. All of our traces were in ASCII instead of binary format. Surprisingly, text traces were shorter than binary traces. This savings occurred by converting integers which took 4 bytes in binary format into variable-length printed ASCII. Since many values were only 1 or 2 printed characters, this conversion saved space. Text traces have the added advantage of being machine-independent; in particular, there are no problems with byte order within words or word length.

While our permanent traces are in our format, traces should be gathered in whatever way is most convenient and converted to our format later. This has two advantages. First, there may already be some tracing facilities in place on the system being traced. It is easier to make minor modifications to gather a little more data than it is to rewrite all of the tracing code to generate output in our format. Second, converting traces from the “natural” format to our format will require CPU cycles, especially to convert time values from the local system clock to 10 s ticks. This post-processing should be done after the trace has been collected so it doesn’t affect whatever is being traced.

Each trace record consists of up to 10 fields. Five of these are present in every trace record—**recordType**, **compression**, **startTime**, **deltaTime**, and **processTime**. The **compression** field determines which of the remaining five fields are present, as explained in the “include” file which follows.

ProcessId is a unique identifier for the process which requested the I/O which the trace record represents. Usually, it is simplest to use whatever identifier the operating system assigns to the process, as that is guaranteed to be unique for some time around each process’ execution.

The **fileId** field uniquely identifies all records of I/Os on a single file. If a file is opened several times, it must be assigned a different **fileId** for each *open*. The identifier must be unique within each process, and should be unique across the trace, if possible. While uniqueness across the trace is not necessary, it is useful for tracking down bugs in the trace analysis software. Note that for physical records, **fileId** is an identifier for the disk written to. Because each physical disk is only “opened” once—by the operating system—all physical records for the same disk should use the same **fileId**.

The **operationId** field identifies all records associated with a single call to *read* or *write*. The logical record for that system call (there is only one logical record per *read* or *write*) can then be associated with all of the physical I/Os it generated. This shows the translation from a logical file position to physical disk blocks for an I/O. Like the **fileId** field, it must be unique within a process, though uniqueness across the entire trace is preferable.

The **offset** and **length** fields have different meanings for logical and physical records. The record type is determined by flags in **recordType**, as shown in the "include" file. For a logical record, **offset** is the byte offset into the file, and **length** is the length of the request. These numbers may be affected by flags in the **compression** field. In physical blocks, **offset** is the physical block accessed on disk, and **length** is the number of consecutive blocks, starting with **offset**, that are accessed. All physical block numbers are relative to **TRACE_BLOCK_SIZE**; thus, if the file system always does physical I/Os in 4 KB blocks, all block and length numbers in physical records will be multiples of 8, since **TRACE_BLOCK_SIZE** is 512.

The time fields are all treated as differences. **StartTime** is the difference between when this I/O started and when the I/O of the previous record started. **CompletionTime** is the difference between when this I/O started and when its completion was reported to the process. For physical records, this will likely be the time between when the request was sent to disk and when the interrupt occurred. For logical records, though, this time may be much longer, as a process is not always restarted immediately when the I/O it is awaiting finishes. Finally, **processTime** is the difference in process CPU time between when this I/O started and when this process's previous I/O started. All of these times are measured in units of 10 s.

Here, then, is the include file which we used for the code which generated and interpreted the traces:

```
/*----------------------------------------------------------------------
 *
 *  iotrace.h --
 *
 *  This file contains the declarations necessary to interpret the
 *  traces.  It includes a definition of a full trace record and
 *  all of the flags used in the trace records.
 *
 *----------------------------------------------------------------------
 */

/*
 * These are the fields in an individual trace record.  They are printed
 * in the order they appear in the structure.  The recordType, compression,
 * startTime, completionTime, and processTime fields are always present.
 * Flags in the compression field indicate whether the remaining fields
```

```
 * are present, or whether the field values are inferred from previous
 * records.
 */
struct traceRecord {
    unsigned short  recordType;       /* type of trace record (see below) */
    unsigned short  compression;      /* compression flags (see below) */
    unsigned int    offset;           /* offset in file (logical) OR
                                       * physical block */
    unsigned int    length;           /* length of access */
    unsigned long   startTime;        /* I/O start time (wall clock) */
    unsigned long   completionTime;   /* I/O completion time (wall clock) */
    unsigned int    operationId;      /* number to associate logical and
                                       * physical I/Os */
    unsigned int    fileId;           /* unique file identifier */
    unsigned int    processId;        /* process that made the request (for
                                       * logical I/Os only) */
    unsigned int    processTime;      /* time in 10  s ticks since this
                                       * process made its last I/O */
};



/*
 *  Flags used in the recordType field.
 */

/*
 * Describes the type of data accessed in this record.
 */
#define TRACE_FILE_DATA       0x0    /* file (user) data */
#define TRACE_META_DATA       0x1    /* metadata, such as indirect blocks */
#define TRACE_READAHEAD       0x2    /* readahead blocks requested by FS */
#define TRACE_VIRTUAL_MEM     0x3    /* blocks requested by VM paging */

/*
 * Is this a logical or physical record?  The value of this flag can affect
 * the interpretation of the rest of the record.
 */
#define TRACE_LOGICAL_RECORD  0x80
#define TRACE_PHYSICAL_RECORD 0x00

/*
 * What type of access was requested?
 */
#define TRACE_READ            0x00
#define TRACE_WRITE           0x40

/*
 * Was the request synchronous or asynchronous?
 */
#define TRACE_SYNC            0x00
#define TRACE_ASYNC           0x08

/*
 * This value for the recordType field is for a “comment” record.  It is
 * ignored by any program reading the trace, but it can be used
 * for human-readable comments.  I used it to record
 * correspondences between fileIds and actual file names, as well as
 * to identify each trace with information in the trace itself.
 */

#define TRACE_COMMENT         0xff
```

```c
/*
 * The next two flags are optional.  They are for data analysis purposes
 * only.  If the trace is being used for simulations, they will be irrelevant.
 *
 * Was the request satisfied in the cache or were disk blocks necessary?
 */
#define TRACE_CACHE_HIT       0x00
#define TRACE_CACHE_MISS      0x20

/*
 * If the block was in the cache, was it a readahead block?
 */
#define TRACE_RA_HIT          0x10
#define TRACE_RA_MISS         0x00




/*
 * The next set of flags are the compression flags.  These flags tell which
 * information has been left out of the trace record and how to generate
 * the missing information.
 */

/*
 * If these flags are set, multiply the relevant value by 512.  These
 * flags should only be set if the relevant information is actually in
 * the record.  Thus, if the offset is not in this record, the
 * TRACE_OFFSET_IN_BLOCKS flags should not be set.

#define TRACE_OFFSET_IN_BLOCKS        0x01
#define TRACE_LENGTH_IN_BLOCKS        0x02
#define TRACE_BLOCK_SIZE              512

/*
 * If any of these flags are set, the corresponding field is missing
 * from the trace record.  The information in them may be computed as
 * follows:
 *
 * processId: take from previous record in trace
 * fileId:take from previous record by this process
 * operationId:take from previous record of this file
 *                  (NOTE: for logical-only traces, this field is
 *                   useless and should be disregarded)
 * block:sequential with previous access to this file
 *                  (ie, previous record starting block + length)
 * length:take from previous record of this file
 *
 * The program reading the traces should keep track of 32 open files
 * for each process.  This is the maximum number most UNIX systems
 * allow open, and it’s unlikely that a process will actively access
 * more than that.  If it does, the traces will still be readable;
 * they’ll just be quite long.
 *
 * If either length or block is present, multiply by TRACE_BLOCK_SIZE
 * if the appropriate flag is set.
 *
 * Time values are always expressed as differences in time.  Start time
 * in the trace is startTime[cur] - startTime[cur-1].  Completion time
 * is completionTime[cur] - startTime[cur].  Process time is the
 * elapsed process time since the last I/O for this process was started.
```

```
 * All time values are in in 10  s units.  Since time values are
 * always compressed, there are no compression flags for them.
 */
#define TRACE_NO_LENGTH       0x04
#define TRACE_NO_BLOCK        0x40
#define TRACE_NO_PROCESSID    0x08
#define TRACE_NO_OPERATIONID  0x20
#define TRACE_NO_FILEID       0x80
```