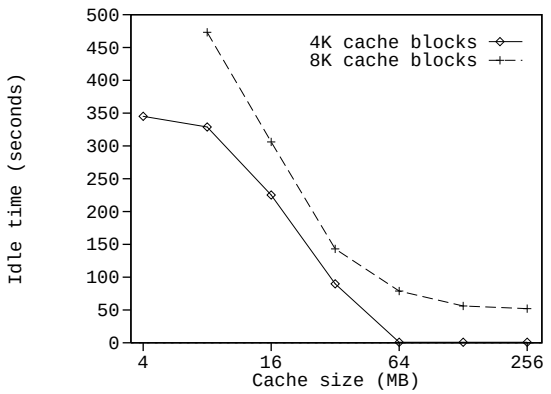

Figure 8: Idle time while running two instances of **venus** with varying cache sizes.

Execution time would be 761 seconds if there were no idle time.

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 as constraints on physical size, distance from the CPU, and access speed of short accesses 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] C. Bonifas. Searching for a Unix mass storage system for a supercomputer. In *Digest of Papers*, pages 129–133. Tenth IEEE Symposium on Mass Storage Systems, May 1990.

[2] R. L. Henderson and A. Poston. MSS II and RASH: A mainframe UNIX based mass storage system with a rapid access storage hierarchy file management system. In *USENIX — Winter '89*, pages 65–84, 1989.

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[4] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California at Berkeley, November 1987.

[5] NAS Systems Division, NASA Ames Research Center, Moffett Field, CA. *NAS User Guide*, January 1990.

[6] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[7] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 15–24, December 1985.

[8] V. L. Peterson, J. Kim, T. L. Holst, G. S. Deiwert, D. M. Cooper, A. B. Watson, and F. R. Bailey. Supercomputer requirements for selected disciplines important to aerospace. *Proceedings of the IEEE*, 77(7):1038–1054, July 1989.

[9] A. D. Samples. Mache: No-loss trace compaction. Technical Report UCB/CSD 88/446, University of California at Berkeley, September 1988.

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

[11] E. Williams, C. T. Myers, and R. Koskela. The characterization of two scientific workloads using the CRAY X-MP performance monitor. In *Proceedings of Supercomputing '90*, pages 142–152, 1990.

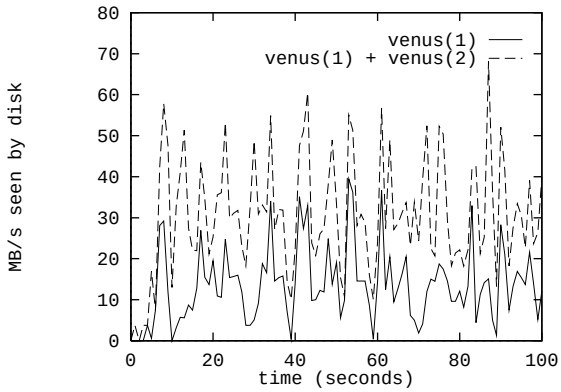

Figure 6: Data rate for 2 simultaneously running copies of **venus** with a 2 MW cache

(first 100 seconds of wall time)

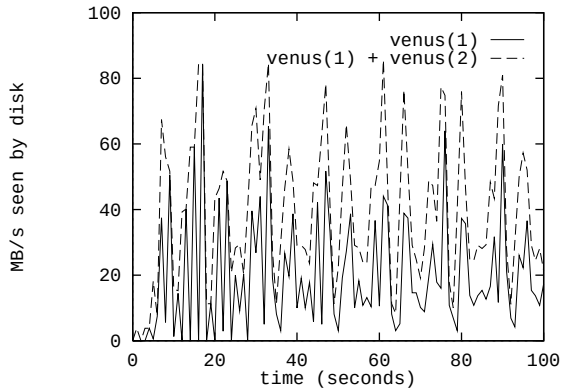

Figure 7: Data rate for 2 simultaneously running copies of **venus** with a 16 MW cache.

(first 100 seconds of wall time)

the SSD, which cannot be directly used as program memory. In addition, SSDs are built from less expensive DRAM, instead of the expensive SRAM used in the Cray Y-MP's memory. Currently, UNICOS 5.0 allows for 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. In addition, resource allocation becomes much more difficult, since the SSD must be allocated among multiple processors, each with programs that (presumably) would want as much SSD as possible. While system-managed buffers in the SSD are less efficient than optimally-managed user buffers, they are considerably easier to use and provide better utilization in a multiprocessor system.

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 $1\mu s$ 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 a 1 MW 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. 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 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 hand, appears to be a much better solution. In a 32 MW SSD, all of programs except one utilized the CPU over 99%. SSD is more likely than 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 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 memory, higher latency channels like those used for network communications can be used.

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 MW 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 [7] 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 process, thus increasing CPU utilization. The first method was prefetching data from disk. Its success was not unexpected as [7] and [10] showed that prefetching was 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 [6] 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. Delayed writes would require a separate process to check all cache data and decide which of it goes to disk, but the per-process overhead is high on many supercomputers because of the large state which must be saved on process context switches. A simple write-behind, on the other hand, merely requires that the operating system note when the write has finished and does not require a separate process. In Unix workstation workloads, delayed writes can often result in temporary files being deleted from the cache before they must be written to disk [6,7]. However, most data written to a supercomputer's main memory file cache must go to disk because iterations take many seconds and files are hundreds of megabytes long, and do not fit into the cache. 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 by itself 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 files inserted extra delays. Much of this delay may have been due to our disk model, which penalized programs for switching files often. With write-behind, the delays did not affect the programs' running time as much. For example, write-behind reduced idle time from 211 seconds to 1 second for a simulation of two identical copies of **venus** running with a 4 MW 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. There are several reasons for this. 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 queueing, so we used an unvarying access time distribution that did not depend on the number of currently outstanding I/Os in the disk system. Another 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 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 in

Application	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	0.011	0.12	0.3	3.8	31.9	0.09
les	24.0	25.2	74	81	325	0.95
venus	26.4	14.7	60	32	456	1.80
upw	0.001	0.01	< 0.1	3.1	32.7	0.12

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

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. As Figures 3 and 4 show, the demand patterns for all of the cycles in an application were remarkably similar.

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, 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 only 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 only one or two more processes than processors are sufficient to avoid wasted cycles. 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.

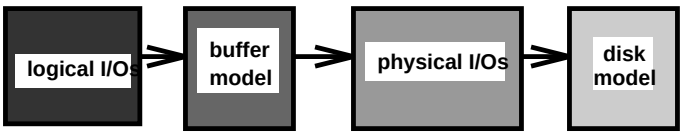

Figure 5: Path an I/O takes from application request to disk.

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 process will make can be simulated. The position of our buffer simulator relative to a simulation of the entire file system is shown in Figure 5.

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

data it will need. Unlike the other two types of I/O above, data staging 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 per iteration, 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—quite close to Amdahl's metric, which predicts 200 bits (25 bytes) of I/O for 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 include the total amount of I/O, the overall and per file read/write ratio, and the overall and per file average I/O size. 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. Their contribution to total I/O is dwarfed by accesses to large data files, such as those generated by computer or gathered by automatic sensors.

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

The only applications which had read/write ratios much less than 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 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 data 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, as a larger memory will simply encourage larger problems which will have the same access patterns.

A file cache will not greatly change the read/write ratio seen by the disk. The files are usually so large that they will not fit into the cache. Programmers already try to utilize locality in their algorithms, so there are few "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.

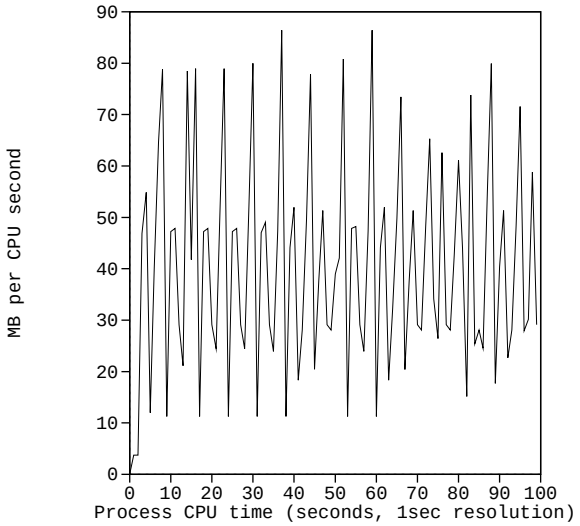

Figure 3: Data rate over time for **venus**.

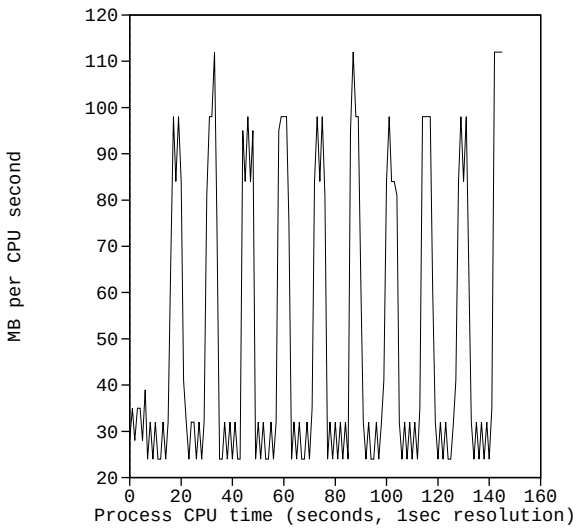

Figure 4: Data rate over time for **les**.

As seen in Table 2, 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 penalty for small I/Os was much less than it would have been for a normal disk. An SSD access still paid operating system overhead and transfer time (about $16\mu s$ for a 16 KB block), 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

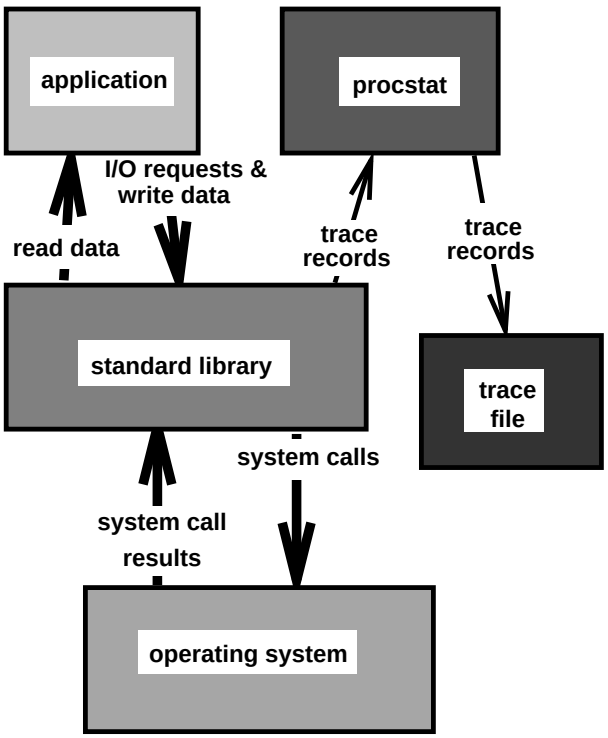

Figure 2: File data and trace record movement in the Cray Y-MP

tle CPU overhead. There was no overhead during non-I/O operations because the tracing mechanism was only active during an I/O system call. Also, the amount of tracing code executed per I/O was small relative to the code the operating system would execute anyway. Overhead per system call was less than 20% of the unmodified call. The total overhead per program was dependent on the number of I/O system calls.

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 on them. The only major requirement is an accurate system 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 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 [11]. 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 staging. Required I/Os are similar to hardware cache misses called *compulsory* in [4]. These consist of I/Os that must be made to read a program's initial state from disk and write the final state back 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 0.75 MB/sec. This rate is easily sustainable by most workstations, and certainly does not demand complex solutions. While the peak 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 program to be restarted. A checkpoint file generally consists of some subset, possibly complete, of the program's in-memory data. Checkpoints are usually 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 computation. 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. Since the I/Os occur relatively infrequently, however, it is easy to have another program ready to run (and not in the checkpoint stage itself) while the first program is writing its checkpoint out.

The third type of I/Os, *staging* I/Os, are 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

Field	Sample Value	Meaning
flags	0xFE	logical/physical I/O, read/write, synchronous/asynchronous
compression	0	which fields can be calculated from previous records?
offset	0	offset within file of this I/O
size	10	size of the I/O (in bytes)
start time	3590	wall time when this I/O started
elapsed time	129	wall time duration of this I/O
think time	10	process CPU time elapsed since the last I/O
file ID	22	which file the I/O occurred on
Op ID	138	unique for each call to the file system
Process ID	4891	process requesting this I/O

Figure 1: A sample trace record

converted to 10 μs units, as we believed this was the correct tradeoff between timer resolution and trace size. 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 when the I/O started. Thus, the effects of multiprogramming could be filtered, as the process elapsed time between I/O events would be dependent only on the application itself and not on 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 [9]. This section gives a high-level overview of the format.

Figure 1 shows a sample trace record. Compression techniques worked especially well on supercomputer I/O 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.

To save disk space and trace-gathering time, the traces were compressed in two ways. First, some fields could be specified relative to the immediately preceding record. These fields included the timestamp fields and the file identifier field. Instead of recording a full 8 or 9 digit time, only the difference between successive times were recorded. Also, 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 the trace was to record the size, length, and process identifier of an I/O relative to the last access made to that file. Again, a few bits in the compression field could indicate that the access was sequential with the last one to that file, or that the request was the same size as the previous one to that file. 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 collected on the Cray Y-MP were logical-level traces. This data included logical file numbers, file offsets, request sizes, and wall clock and process clock timestamps. Because none of the collected data was internal to the operating system (as physical block numbers would be), all the data could be collected by code running at user level. Thus, modifications to the operating system were unnecessary. 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 from a file and the average and maximum times to do an I/O. Major events such as file opens, file closes, and process forks are traced by the standard Cray software, but 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 packets, which include an 8 word header and whatever data is necessary for the system call being recorded, and writes them 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 packet per `read` and `write` call would have produced far too much data. The packet headers are large compared to the three to five words recorded per call. `Read` and `write` records for each file were sent in batches, so one header served for hundreds of I/O calls and the overhead was amortized over many trace records. 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 time-ordered stream of all the accesses from the trace packet file requires buffering all the I/Os between flushes, since a packet written during the flush might contain an I/O access from just after the previous flush. If the interflush interval was too high, this could require buffering many megabytes of trace data, which was impractical on the workstations we used to analyze the data.

Collecting traces in this manner required very lit-

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	7,953	0.031	0.1	4
les (large eddy)	146	224	7,803	22,384	0.317	53.4	153
upw (polynomial)	596	56	62	1,840	0.445	0.1	3
venus (climate)	379	55.2	16,712	34,904	0.032	44.1	92

Table 1: Characteristics of the traced applications.

climate models. These included **gcm** (General Circulation Model), **ccm** (Community Climate Model), and **venus** (a simulation of Venus' atmosphere). These were 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 motion of a helicopter blade through the surrounding air. It was the only one of the programs traced explicitly designed for use with the SSD 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** application used the Navier-Stokes equations to model turbulence. This algorithm only calculates large-scale effects and directly models the small-scale effects. A more complete description of the algorithm is beyond this paper, but one can be found in [8].

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 just staging data in and out. By breaking up the data array into blocks, empty blocks (corresponding to empty parts of the matrix) 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. A matrix block of size $2N \times 2N$, which consists of 4 subblocks of size $N \times N$, contains $4N^2$ elements, all of which must be read in. If three of the four subblocks are actually empty, though, only one subblock, with N^2 elements need be read in. This saves 75% of the I/O bandwidth. The program, however, seems to have chosen a relatively large access size despite the possible advantages of 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 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 recorded for each I/O. There were three timestamps for each I/O event. The first 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 format, this value was

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 to better plan usage of memory and CPU resources, as there is a relatively static queue of jobs to run and none of the jobs require fast response time. 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 8/8128 at NASA Ames, the computer on which we traced the applications. This computer has eight processors, each with a 6 ns cycle time. The system has a total of 128 MW of memory (each word is eight bytes long) shared among the eight processors. The I/O system has 35.2 GB on high-speed disks, each capable of sustaining 9.6 MB/sec, 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 [1,2,5]. 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 Y-MP system already has the maximum configuration of Y-MP memory (128 MW), so I/O problems cannot be alleviated simply by adding more main memory.

The UNICOS process scheduling mechanism at NASA affects the way programmers choose to structure their implementations, and thus I/O demands. Batch jobs, which include any program requiring more than ten 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, sufficient to run one or more jobs from the queue. 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 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, only a few more jobs than processors are required to keep all processors busy. Each queue may have sufficient space to run more than one job in; in particular, queues with low memory requirements may run several jobs at once. 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, 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, both to simulate a model through time and to gain higher accuracy for approximations.

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 user codes running on the Cray Y-MP. We chose to trace applications with high I/O rates, both in megabytes per second and accesses per second. While many supercomputer applications do not perform a lot of I/O [11], we decided to concentrate on codes that did require it. I/O-intensive applications stress the file system and I/O hardware more, revealing performance bottlenecks. Programs that perform few accesses 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 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

Input/Output Behavior of Supercomputing Applications

Ethan L. Miller
Randy H. Katz

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California at Berkeley
Berkeley, CA 94720

Abstract

This paper describes the collection and analysis of supercomputer I/O traces and their use in a set of buffering and caching simulations. Our analysis of these traces shows that program accesses are cyclical and bursty. We classify file references into three types, and show the general access pattern and intensity for each type. Simulations of a Cray Y-MP's file system and attached disks using these traces show what resources are needed to maximize the CPU utilization of a supercomputer given the very bursty I/O request pattern. 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 and consuming more data, but I/O systems are not becoming better able to cope with this huge volume of information. Multiprocessors are exacerbating this problem, as the number of disks and tape drives, and thus aggregate bandwidth required, increase. The actual bandwidth available is not usually scaled up at the same rate as the aggregate processing speed, however. According to Amdahl's metric [3], each MIPS (million instructions per second) should be accompanied by one megabit per second of I/O. Solving this problem requires correct matching of bandwidth capability to application 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, we must analyze the I/O patterns of applications running on such computers To do this requires I/O access traces from real supercomputing applications, which we have gathered for the analysis in this paper.

This paper first examines general file system characteristics, especially those important to supercomputers. Next, we describe and analyze the access patterns of the applications which we traced. 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 [7]. 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 references [10]. This may reduce the number of separate disk requests, but the amount of data transferred will not go down. In [7], 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 [6], 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 load on the disk system against the risk of losing data by not writing it to disk immediately. By totally eliminating disk I/Os associated with very short-lived