SCALABLE SHARED MEMORY MULTIPROCESSORS

Edited by

Michel Dubois
University of Southern California

and

Shreekant Thakkar
Sequent Computer Systems

KLUWER ACADEMIC PUBLISHERS
BOSTON/DORDRECHT/LONDON

Univ. of Wisconsin, March 1989. Computer Sciences Technical Report #835.

[Gottlieb *et al.*, 1983] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The nyu ultracomputer–designing an mimd shared memory parallel computer. *IEEE Trans. on Computers*, C-32(2):175–189, February 1983.

[Graunke and Thakkar, 1990] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[Gustafson, 1988] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[Heath and Worley, 1989] Michael Heath and Patrick Worley. Once again, amdahl's law. *Communications of the ACM*, 32(2):262–264, February 1989.

[Karp and Flatt, 1990] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, May 1990.

[Snir, 1982] Marc Snir. On parallel search. In *Proc. Principals of Distributed Computing*, pages 242–253, August 1982.

[Zhou, 1988] Xiaofeng Zhou. Bridging the gap between amdahl's law and sandia laboratory's result. *Communications of the ACM*, 31(8):1014–1016, August 1988.

Measuring Process Migration Effects Using an MP Simulator

Andrew Ladd, Trevor Mudge, and Oyekunle Olukotun
Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan

Abstract

This chapter examines multiprocessors that are used in throughput mode to multiprogram a number of unrelated sequential programs. This is the most common use for multiprocessors. There is no data sharing between the programs. Each program forms a process that is scheduled to execute on a processor until it blocks because its time-slice expires or for I/O. The process can migrate among the processors when its turn to resume occurs. The degree of process migration can dramatically impact the behavior of caches and hence the throughput of the multiprocessor. This chapter investigates the behavior of cache misses that can result from different degrees of process migration. The degree of migration is varied by assigning each process an affinity for a particular processor. An efficient multiprocessor cache simulator is described that is used in the study. The study is restricted to shared-bus multiprocessors and two contrasting cache consistency protocols, write update and invalidate.

INTRODUCTION

In today's computing environment, multiprocessors are playing a greater role than ever before. Originally, multiprocessors were intended to accelerate single program execution through parallel processing. However, their most common use to date has been as "throughput" machines in multiprogramming environments. These systems typically consist of 2 to 32 processors connected by a common backplane bus. Memory units are also attached to this backplane such that all processors have equal access to memory. This is commonly referred to as a tightly-coupled shared-memory system.

Programs are selected to execute on the processors of the machine by the operating system scheduler. In a typical case, the scheduler takes a program (process) from a common run queue and sends it to one of the available processors. When the process' time-slice expires or it blocks on a system call (for example when waiting on I/O), it is removed from its processor and another process from the run queue takes its place. This is characteristic of a preemptive scheduler. When the process is ready to execute again it is placed back on the run queue and waits its turn for execution. In a straightforward round-robin type of scheduling, there is no guarantee which processor this process will execute on next. With respect to cache performance, the best processor for it to execute on next is most likely the one it executed on last. There is a much greater likelihood that the process will find more of its initial memory references in the cache of the processor it executed on last, thereby reducing the number of instructions and data that must initially come from memory[1]. This gives the process an affinity for the cache of the processor it last left [2]. On the other hand, if the process is scheduled on a processor it has never executed on, it will have no footprint (matching data and instruction lines) in that processor's cache. In this case it will have to bring in all the instructions and data it initially needs from memory, making an initially cold cache warm to the process [3]. Bouncing a process from processor to processor in this manner is termed process migration and can have a detrimental affect on the overall cache performance and bus traffic in a computer system. Process migration also increases the

interference of a migrating process' address references with the cache footprints of other processes. This interference will displace cache lines from previously executed processes, thus reducing the cache footprint of those processes.

An understanding of these process migration effects on system performance is important for the operating system designer. If better scheduling algorithms can send a process to a warm cache (one that the process left recently) then the cache hit rate is improved and bus traffic is lowered during the initial execution of the process. Transient or compulsory misses, initial misses that result from warming up a cache to a process, are reduced if the process is scheduled to a cache which is already warm to it. Steady state misses are cache misses that result from memory references displacing valid cache lines of the executing process [4]. Process migration tends to increase the transient miss behavior of caches in the system while leaving the steady state miss behavior unaffected. Other aspects of the design can also benefit from understanding process migration effects. For instance, cache and bus designers should consider the impact of how process migration effects could degrade system behavior.

A method to keep the caches consistent is required because processes migrate from processor to processor. This is especially important when actual data sharing occurs between processes because each processor needs to have equal access to the shared data. In tightly-coupled shared-memory systems there are two basic types of consistency algorithms: invalidate and update based. The invalidate protocol is based on eliminating shared data in other processor's caches when one processor modifies its copy of that data. The update protocol updates shared data in other caches upon each modification of the data in the expectation that it will be reused.

The choice between using physical caches or virtual caches has many tradeoffs. Virtual cacheing does not require the translation of the virtual address in the TLB (Translation Lookaside Buffer) before accessing the virtual cache. This significantly reduces the cache access time after the generation of the virtual address [3]. Cache line synonyms can cause problems in a virtual cache. Synonyms occur when two cache lines indexed by different virtual addresses refer to the same physical address.

A problem arises when a processor tries to modify one of the synonym lines because it is difficult to identify the other synonyms for the required modification. Virtual caches also create added complexity in the bus watching of the consistency protocols. A reverse TLB is required to map physical bus addresses into virtual addresses because the references on the system bus are physically addressed.

The trend in cache design for processors has been to increase cache size as processor speeds have increased. This is to compensate for the growing gap between main memory speeds and processor speeds. Main memory cycle times (and intrinsic bus delays) have not decreased at the pace of processor cycle times. This makes memory references that miss the local caches even more costly. The easiest method to reduce this cost s to make the local caches larger. In most designs the first level cache primary cache) is limited to the amount of chip area the CPU designers :an set aside. One solution is to create a secondary level of cacheing that :an alleviate the high cost of references missing the first level cache. The econdary cache is usually made up of standard SRAM parts assembled n the board in close proximity to the CPU and is typically slower than he primary cache. A secondary cache built this way is usually an order f magnitude larger than the primary cache. It is for these large cacheing ystems that longer address tracing is crucial. Effectively analyzing these ırge caches requires a large address stream to measure its steady state ehavior [5]. Address streams that are too short may only measure the ıitial warming of the cache (i.e., transient behavior).

The experiments conducted in this study examine cacheing behavior in multiprocessing environment. Only process migration effects are condered, therefore processes do not share data between other processes. his restriction was made to simplify the tools and to focus on "throughıt" environments. The tools used include an address trace generator d a multi-cache simulator. These tools provide fast and accurate mearements of cacheing behavior with respect to scheduling algorithms, nsistency algorithms, and cache characteristics.

Method	references	slow-down	abstraction
Analytical	>1M	-	Yes
Trace Driven	>1M	10X	No
RTL	~10K	1000X	No

Table 1: Comparisons of Cache Analysis Methods.

Previous Studies

Many studies have focused on the sharing between parallel processes. These studies [6, 7, 8, 9, 10] are interested, primarily, in exploring the effects of cache consistency or coherency where there is shared data. Although some of these studies account for process migration, the central theme they explore is somewhat different from characterizing process migration effects on multiprocessor caches.

Some of these studies [6, 7, 8] use analytical models to simulate address streams. While they can produce results very quickly and have the most flexibility, their results are an abstraction of real system behavior. At the other end of the spectrum are studies [10] which completely simulate each processor in some form of detailed register transfer level. While this method produces the most accurate results, it is far too slow to capture large multiprogrammed address traces. Trace driven simulations offer a compromise between the two. They take real address traces from targeted programs and run them through a high level cache simulator. Trace driven methods are considerably faster than detailed models but still simulate "real" system behavior. Faster methods are becoming increasingly desirable because cache sizes are growing. As cache sizes grow, the need for longer traces is crucial to accurately measure the behavior of these large caches. Table 1 illustrates a comparison of these methods.

This study uses a high level cache simulator driven by address traces extracted from benchmark programs. The size of the traces excluded register transfer simulation from consideration due to the time required for RTL simulations. Analytical models were not considered due to their

inaccuracies.

SCHEDULING

A key aspect to examining process migration effects is to set up a multiprogramming model that is sufficiently realistic yet deterministic enough to allow the resimulation of the same sequence of scheduling events through various cache configurations. When studying migration effects, it is important to identify which address stream is produced by which process (program). Without this identification it is difficult to isolate cacheing behavior affected by process migration.

Not only is process identification important but so is finding an accurate algorithm to interleave the multiple processes (scheduling). One approach is to let the host system schedule the multiple programs with its native scheduler and have each program send its results to the same cache simulator. The simulator then processes the address streams as they are received. In this way, the cache simulator analyzes the address traces generated in a manner consistent with the host machine's scheduler. This provides a straightforward method to utilize a real scheduling algorithm [5]. One drawback to having scheduling performed by the host is that it is sensitive to traced code side-effects, since the host scheduler cannot account for the annotated code. Another drawback is that the traced code would rarely schedule the same way between separate executions due to the uncertainty of system loading.

An alternative scheduling approach is to take a known scheduling algorithm, or statistical measurement of one, and apply that directly to the programs which are executed. This approach can compensate for some of the tracing side-effects by altering the algorithm appropriately. It also allows repeatable scheduling for different executions of the programs.

Closely related to program interleaving is processor scheduling. As with interleaving, an algorithm is needed to schedule each processor in the multiprocessor set. This problem is less severe since a new context can start execution on any idle processor or the first that becomes available. Since measuring the effects of process migration and affinity is the topic of this chapter, the scheduler must be flexible enough to study various scheduling algorithms and their impact on system performance.

CACHE COHERENCY

A method to keep the caches consistent is required because processes migrate from processor to processor. This is especially important when actual data sharing occurs between processes. Since this study does not examine data sharing, why should cache coherency matter at all? To answer this question, one only has to note that the context of a process must be consistent with past contexts of that process. For example: if process A was on processor 1 and referenced location X, then location X is valid in that processor's cache. Now assume process A is swapped out of processor 1 and is later scheduled on processor 2. While executing on processor 2 it writes to location X. Without a consistency protocol, processor 1's cache line for X will not be updated or invalidated. Now if process A is re-scheduled back to processor 1 and reads X it will receive a stale value.

Since cache coherency is an important consideration with process migration, valuable insight can be gained by measuring different coherency effects on caches. Cache coherency algorithms have been studied at great length. Many papers have been devoted to the comparison of coherency based on directory schemes and snoopy protocols [6, 7, 8, 9, 10]. Although these studies focus primarily upon the sharing of data, valuable conclusions are drawn as to when each scheme is beneficial over the others. Directory schemes are ideally suited for a distributed bus type of organization while snoopy schemes are more suited for centralized shared bus organizations. Distributed bus organizations, and thus directory based coherency, are more applicable with large numbers of processors where a centralized bus cannot support the necessary bandwidth for the communication traffic between processors and memory. Centralized buses on the other hand are simpler and less costly to implement and are therefore suited for a modest number of processors. Although the directory mechanisms tend to reduce the bus traffic bottleneck they usually cause a bottleneck at the memory modules because memory must maintain the coherency tables. Both the directory and snoopy based schemes can support invalidation or update protocols.

Finally, software approaches keep caches coherent without causing bus

or memory bottlenecks. This is done by flushing a processor's cache at each context switch. Their primary drawback is the reduction of usable cached data because a process is guaranteed to enter a cold cache at each context switch.

ADDRESS TRACE GENERATION

Before a cache analysis of "real" multiprocessing system can begin, accurate address traces must be generated. Ideally the address trace contains the complete set of address streams as seen by each processor in a multiprocessor system, or the equivalent information. This allows a cache simulator to process addresses exactly as they occurred. There are many factors that prevent the generation of an ideal set of address traces. Such factors include the side-effects of generating or extracting the traces, the performance degradation between the traced code and the actual code, and the difficulties of tracing certain kernel routines.

One major deviation from the ideal is that in many instances one cannot acquire address trace data from all instructions executing on a particular CPU. The particular method used may restrict tracing only to user programs and ignore the tracing of kernel code. This is particularly burdensome when trying to measure how context switching affects caches since various kernel procedures are likely to execute repeatedly on each processor.

Another deviation is the intrusion of the tracing method on the traced program. Tracing is usually done by inserting code between load, store, and program control instructions to record memory references. The program control instructions provide basic block boundaries which identify program flow changes. Whether the code is inserted into microcode [9] or inserted into program code [5, 11, 12], it still only executes when a program is traced. This extra code obviously creates side-effects and behavior that is never seen in the original untraced code. Memory management side-effects are examples of abnormal behavior which manifest themselves from the added code. The added calls to the inserted code to record addresses could cause page faults and displace TLB entries not normally seen by the original program. In addition, the address traces must be written somewhere, whether to a UNIX^{TM} pipe or a file, and are

bound to cause page faulting and TLB displacement. Reducing most of these side-effects and abnormal behavior is another concern in selecting a good trace generation method.

Closely related to the side-effect issue is the performance degradation of the code that is traced. If additional code is added to the original program in order to trace the address references, there is little chance that this added code will not degrade program performance. Performance degradation can cause several unfortunate by-products. First, system clocks and timers appear to run faster than as seen by the original program. Since fewer instructions of the original program are executing between timed events it appears to the executing program that clocks and timers run faster. Furthermore, I/O will appear faster, the program will appear to have a smaller time-slice before preemption, and kernel interactions will appear to execute faster if kernel code is not traced. For example, the Titan address generation scheme [5] slows down the execution of a program by as much as 12 times. ATUM [9] increases the time of execution by a factor of 10 and AE [11] induces a factor of 4 increase in execution time.

Finally, an enormous amount of data is created when generating address traces for large runs. Manipulating this data into a manageable form is our last consideration in tracing data. Two approaches are available to solve this problem. The first and most straightforward approach [5, 12] pipes the traces from the traced file directly into the program that analyzes them. This saves all the overhead of storing the traces into a file and, depending upon the disk cacheing method, could save time. In addition, disk space is no longer an issue when using pipes. The primary drawback with this method is that not all experiments are repeatable with the same set of traces, especially those involving kernel interaction. Each run will produce a slightly different behavior which could skew results when comparing different cache configurations to the same programs. The second approach is to save the traces in compressed form [13], or to save a smaller set of events that can be used to extract the real traces [11]. These approaches allow different experiments to run on the same set of data but suffer from the extra file system overhead that the piped version eliminated. They also require a great deal of file space.

The Mache compaction scheme used with UNIX^{TM} compress has been shown to reduce the traced data to approximately 2.5 bits per reference [13]. The AE method of saving compressed significant events have been shown to reduce the traced data close to 0.5 bits per reference [11].

TOOLS

For this study, an address tracing tool was employed which annotates the object code of the executable program. This choice was made because it could generate addresses from the original program as well as the library routines that were called (since the annotation is performed after linking). The address tracer also allowed the address stream to be sent to a UNIX^{TM} file descriptor which simplified piping multiple streams into the cache simulator.

The cache simulator reads in the address streams from the various executing programs via the UNIX^{TM} file descriptors. In addition, the simulator can sequence the reading of the file descriptors according to any particular process scheduling algorithm. As the addresses were read, the cache arrays were examined and updated in accordance with the cache consistency algorithms. In this study, only snoopy bus algorithms were examined. Both update and invalidate protocols were modeled to study their respective effects on cache behavior in the presence of process migration.

The caches in the simulator are direct-mapped with variable index and line sizes. Tag matches are based upon the non-index and non-line portion of the physical address and the pid (process identifier) of the process. Cache lines are allocated on read, as well as write, misses. Context switching is modeled by scheduling a process to execute for 3000 instructions on average[14]. A Poisson distribution with an expected value of 3000 is generated when a process is scheduled to execute. This value is set as the limit to how long the context of the current process will run. Context switch times thereby distribute around a Poisson mean of 3000 instructions.

The pid of each process is included in each address tag. This allows the disambiguation of virtual addresses of different processes. Since no sharing between processes exist this modified tag causes no unwanted

mismatches between different process' addresses. Since only virtual address traces were available, special care was required to distribute the references in the cache. In a physical cache, virtual pages are translated into physical pages which create a random distribution over each process' address space. A method to emulate this behavior is highly desirable. Sites and Agarwal proposed a method to hash the virtual addresses based on the pid of the process[15]. They found that caches fed with hashed virtual addresses provided similar performance to physically addressed caches. Virtual address hashing gives a pseudo-random distribution of each process' virtual address space inside the physical cache. If no adjustment to the virtual addresses were made the miss rate would be unrealistically high because many programs reside in the same range of virtual space. Physical page allocation tends to randomize pages in this space thus causing fewer cache line conflicts between processes.

A piping program linked the address trace generation program to the cache simulator in such a way that the cache simulator could read off each trace from a unique file descriptor. The piping program created a UNIX^{TM} pipe for each executing program and sequenced the file descriptor for each pipe starting at a predetermined base.

Scheduling Details

Two types of scheduling are examined, one that schedules globally and one that schedules locally. The global scheduler takes the processor which has been idle the longest, or if none, the first to become available, and schedules the process that has been waiting the longest to execute. Jobs are maintained in queues that hold processes which are waiting for execution. As a process' time-slice expires, it is placed at the rear of the queue and the process at the head is then chosen to run on the first available processor. Process priorities are not considered in the scheduling mechanism, so all processes have equal priority and equal access to each processor. In contrast, the local scheduler assigns each process to a unique processor corresponding to its initial context of execution. From then on that process only executes on its assigned processor. While this local scheduler is highly unrealistic, it does show how complete process affinity can affect cacheing behavior. For this reason it was chosen for

this study. Realistic schedulers usually schedule globally or provide a mixture of global and local scheduling. Local scheduling by itself does not allow for fair load sharing among the processors in the system. Refer to Figure 1 for details concerning the flow of the scheduler.

Particular attention must be paid to how each process stream interacts with the others. How much processing should be allowed on one stream before work begins on the others? How do the separate address streams synchronize with each other given the states of the caches of the concurrently executing processors?

A logical separation of address stream execution allows one processor to work on an address stream from the beginning of the context switch to the end. This works in all cases where no data sharing occurs between ;eparate processes and when processes run sequentially in the system. When a process exits a processor and is later rescheduled, the cache :oherency transactions for its address space are guaranteed to be resolved 'or any memory reference occurring before the process starts. Thus every)rocessor can run an address stream for a current process until it the tream switches to a new context. During execution of each context he address references are made consistent in the other caches such that vhen a process begins execution in another cache, its cache is guaranteed onsistent.

One problem with this approach is that even unrelated processes share ome amount of data in the kernel. Therefore, since coherency is not naintained on a cycle-by-cycle basis between all caches, some degree of rror is introduced into the results. This problem is difficult to overcome 'ithout expanding the simulator to provide cycle-by-cycle simulation. 'his would run a great deal slower due to the added overhead of the more etailed model. In addition, a method is not available to produce kernel 'aces. As noted, in this study the targeted user programs contain no arallelism or data sharing. Thus, a process can only invalidate or update iche lines in other caches that were allocated on previous contexts of iat process. In other words, process A cannot invalidate or update che lines that were allocated to process B since they do not share data ddress tags and pids will not match during snoopy look-up). The only iy for process A to affect the cache lines of process B is to displace

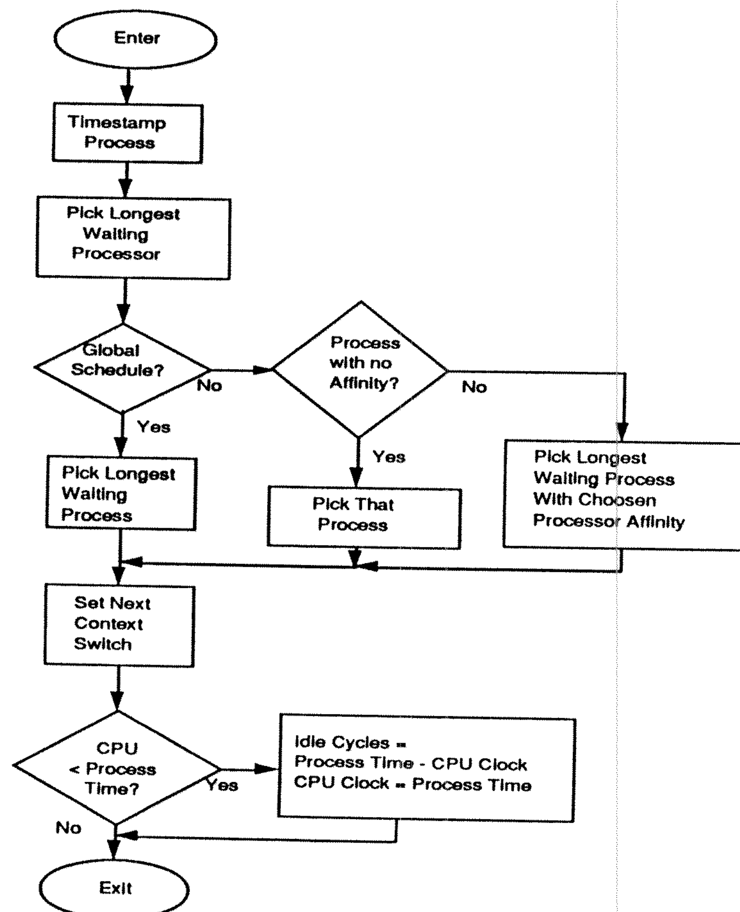

Figure 1: The Scheduler

them in the cache in which process A is currently executing.

Since a process is simulated in sequential order as dictated by the scheduling algorithm, we can guarantee that no execution of a process will precede (in the simulated context) a previous execution of that process. This is accomplished by time stamping a process when it is context switched out of a processor. Then if that process is scheduled to a processor whose clock (cycle count) is less than the time stamp, the processor's clock is set to the time stamp. This guarantees the sequential (with respect to cycle count) execution of the process. This argument when combined with the one from the paragraph above guarantees that misses due to invalidations can only occur when a previous instantiation of the process on another processor caused those invalidations.

Furthermore, we can detect when a miss was due to invalidation or displacement. This is accomplished by noting that even though simulation is not accomplished cycle by cycle, a processor's cache is consistent at the time a process is scheduled on it. Since a process cannot schedule itself on two processors at the same time there are no race conditions with respect to coherency. The question must be asked: Is there a race condition between invalidation of a line and another process' displacement of that line? This reduces to a don't care situation because upon completion of the time-slice the line will be displaced to the new value and pid whether the invalidation or displacement occurs first. The next process using the cache only knows that it missed due to displacement and not invalidation. For example, a cache line in processor 1 is valid from a previous instantiation of process A (tag = "pid + tag"). Now process B is executing on processor 1 and process A is executing on processor 2. If process A invalidates X and then process B allocates that line, the resulting state of the cache line is that of the allocation of process B. Conversely, if process B displaced X first then the invalidation from process A occurred, the invalidation would not match the new tag so the cache line would still contain the valid reference of process B. In either case the result is the same, a valid cache line for the reference of process B. For these reasons we can justify processes executing for their entire time-slice without cycle by cycle synchronization.

Coherency Details

This study examines snoopy based coherency algorithms since they are more flexible and are more prevalent in today's computers. The two basic types of snoopy protocols are invalidation and update protocols. We chose to simulate both and compare their effects on cache migration. The invalidation protocol is represented with a "Berkeley" type of coherency while the update protocol is represented with a "Firefly" type of protocol [8]. The Berkeley approach is a write-back protocol that maintains coherency on writes by invalidating matching lines in other caches. Digital Equipment Corporation's "Firefly" approach is also a write-back protocol but maintains coherency on writes by updating the matching cache lines in the other caches.

BENCHMARK PROGRAMS

Four benchmarks were chosen for simulation. Each of these was compiled and the object code annotated to generate address traces. The address traces of each program were fed into the simulator via a UNIX^{TM} pipe. The simulator then interleaved the traces corresponding to the type of scheduling selected. All programs ran in a UNIX^{TM} environment on a DECstation 3100^{TM}. Below is a description of each program; Table 2 provides characteristic data for each.

awk A UNIX^{TM} pattern scanning and processing language which scans a file for matching patterns and executes associated actions upon a match.

diff The UNIX^{TM} differential file comparator utility. This command compares the contents of two files and lists their differences.

dry2 The Drystone benchmark written in C. 50,000 iterations of this program were executed.

nroff A UNIX^{TM} text formatter. This command formats the text of a given file.

Benchmark	I-references	D-reads	D-writes
awk	32388725	4871777	3158560
diff	30939027	5870564	2582993
dry2	30254676	7060852	2858599
nroff	32036008	5983441	2701127
total	125618436	23786634	11301279

Table 2: Benchmark Programs

RESULTS

This section discusses the results of three experiments. In each experiment, simulations were conducted on a unified, direct-mapped cache with a line size of four words. Four pairs of scheduling and coherency combinations were modeled in each experiment: Global Berkeley, Local Berkeley, Global Firefly, and Local Firefly. As previously described, local scheduling refers to adding a processor affinity to each process and only allowing it to run on that processor. Global scheduling sends each process to the first available processor in a round-robin fashion. The cache coherency algorithms are modeled after the Berkeley and Firefly approaches described in a prior section. Each experiment was performed on models containing one to four processors. The measurements focused on three areas: total cache misses, transient cache misses, and misses due to invalidations. In addition, read and write miss measurements were recorded for the last two experiments.

The first experiment set the cache size to 32K words and applied the hashed virtual address traces of the four benchmarks to the simulator. Figures 2 through 4 illustrate these results. The second experiment increased the cache size to 128K words with everything else remaining the same. Figures 5 through 9 illustrate the cache performance resulting from the simulations of this larger cache. Finally, an experiment modeling a virtual 128K word cache was performed by applying the unhashed traces to the cache configuration of the second experiment. Its results are presented in Figures 10 through 14.

Cache Sizes

As expected, larger caches reduce the amount of cache misses across all scheduling and coherency algorithms. Larger reductions are seen with respect to the global scheduling and Firefly algorithms. As seen from Figure 5, the three processor 128K word cache used with the Global Firefly algorithm reduced cache misses by 46 percent over the same configuration with a 32K word cache (Figure 2). Large caches are effective in reducing miss rates in global schedulers since they greatly decrease the amount of transient misses. Recall that the causes of transient misses are interference by processes displacing each other's cache lines or initial cold start misses. Since larger caches tend to spread out the processes in the cache, less interference is seen. As the cache size grows, the number of transient misses due to global scheduling approaches the number of misses generated by local schedulers thereby alleviating some of the global scheduling penalties. Compare Figure 3 and Figure 8 to see how the number of transient misses due to global scheduling approach the number of transient misses due to local scheduling.

The Berkeley approach, although also benefiting from the reduction of transient misses, does not reduce its miss rate as drastically in larger caches. This is due to the additional cache misses from invalidations. In fact, by increasing the cache size, more misses are generated from invalidations from other processors. This is seen by comparing Figures 4 and 9. Larger caches are more likely to contain cache lines which match during an invalidation cycle thereby creating a miss if referenced later. The update approach generates no miss in this case since the cache line is kept active by updating the data.

Both the amounts of invalidates and updates increase with the larger caches during process migration. Invalidate traffic increases a modest amount due to the increase of shared-dirty states in the caches as processes migrate from cache to cache. The larger caches retain more data thereby keeping cache data around for longer periods of time. Update traffic increases by a slightly larger amount as the cache size increases. This is primarily due to the larger caches having a greater likelyhood of retaining data. Since the data has a greater chance of surviving in larger caches, more updates are required to keep the data coherent. Only when

no other cache contains a copy of the cache line may updating stop for that particular line.

Scheduling

Scheduling was shown to have the greatest impact on the total cache miss rate. It should be noted that when the processor count reaches four, even the global algorithms begin to schedule fairly locally. This s due to each process having a processor all to itself. When a process' context expires, no other process is waiting, so it gets scheduled back on the same processor immediately. This is true until one of the programs inishes execution, at which time two processors would be free upon each context switch. The global scheduler then schedules the switched out process to the "other" processor since it was waiting the longest (load haring).

As seen from Figures 2 and 5, local scheduling provides the greatest eduction of total misses especially with larger numbers of processors discounting the four processor case). In many cases, local scheduling educes the number of misses by one-half or more compared to the cor- esponding global scheduler. As the number of processors increases, the mounts of transient misses and misses due to invalidations increase for lobal scheduling (discounting the one and four processor case). The lobal scheduler sends a process to more and more processors between iccessive visits to the same cache. This tends to increase the number f transient misses resulting from processes interfering with each other y displacing one another's data and by increasing the amount of cold art misses (see Figures 3 and 8). It also increases the amount of in- lidations to each process' footprint since each process spends a smaller nount of time on each processor. This becomes more of a problem for nall caches since the contention for the smaller number of cache lines creases. Local scheduling is beneficial in systems with large numbers processors since it prevents this type of behavior. This is encouraging om a scheduler design perspective because there is the potential for a ge reduction in the number of cache misses.

The Berkeley protocol benefits from local scheduling the most. Since e Berkeley protocol uses invalidations to keep the multiple caches

coherent, reducing the amount of invalidations will help increase the amount of data retained in each cache for each process. Local scheduling eliminates invalidation of cache lines (not invalidate traffic) by keeping the process on one processor as shown by Figures 4 and 9. A data line only gets invalidated if another process writes to that address. Since sharing was not examined in this study, no other process would generate writes to that address. Therefore, the Berkeley protocol benefits from local scheduling by a reduction of transient misses and by an elimination of misses due to invalidations.

Local scheduling greatly reduces the invalidate traffic, sometimes by as much as 90 percent over global scheduling. This difference grows as the numbers of processors increase. In fact, as the number of processors increase, the number of invalidates will increase for global scheduling while they decrease for local scheduling. This is explained by the amount of hits to dirty data lines in the caches. While a process runs out of a particular cache, writes to dirty data lines require no invalidations. Once a process migrates to a new cache, writes to what were once dirty cache lines now require invalidations to purge the other caches of matching lines. Local schedulers, on the other hand, do not generate this added invalidation traffic due to migrating dirty lines. They do generate extra traffic when dirty lines are displaced by other processes if future writes are made to the displaced address. Even so, local schedulers were also seen to reduce displacement of data over their global counterparts.

Updates are completely eliminated in a local scheduler since no process ever migrates to another cache. Global scheduling does generate update traffic which increases as the processor count increases. As processors are added, there is a greater likelihood that a migrating process has left an active line in one of the caches since the only way to purge lines from past caches is for other processes to displace them. An update is only required if another cache contains the data line the current cache is writ- ing. Since increasing the number of processors increases the likelihood of a process' lines existing in other caches, the number of overall updates must increase.

Cache Coherency

The Firefly coherency protocol does the best with respect to reducing cache misses in the global scheduling approach. As seen from Figures 2 and 5, the Global Firefly approach reduces the amount of misses up to 54 percent over the Berkeley method. This is primarily due to Firefly generating no invalidations. Thus no cache misses are due to invalidates in the Firefly approach, while the Berkeley approach generates a sizable amount of misses due to invalidates. The amount of transient misses between the two approaches are practically identical as would be expected since transient misses are a function of the scheduling algorithm.

Local scheduling evened the differences between the miss rates of the Berkeley and Firefly protocols. Since the transient miss behavior is the same between the two (Figures 3 and 8), the only difference lies in misses due to invalidations. While Firefly never generates invalidations, local scheduling reduces misses due to invalidations in the Berkeley approach to zero. Therefore local scheduling causes both Berkeley and Firefly to generate the same amount of transient and invalidation misses resulting in the same overall miss behavior.

The miss rate of Firefly verses Berkeley is misleading since Firefly pays a large overhead to prevent invalidations. On every write to a word that is in multiple caches, an update must be performed. As a process migrates from processor to processor, the amount of duplicate cache lines increases between the multiple caches. Since multiple caches will retain each process' cache lines until they are displaced, updates will have to be performed while multiple copies exist. As the number of processors increases the probability that a cache line does not exist in multiple caches decreases. This gives rise to an abundance of update cycles to keep all the caches coherent. The Berkeley protocol, on the other hand, invalidates all the other duplicate cache lines after the first write of a process to the duplicate line. From then on, the cache line is dirty and resides only in the current processor. From a process migration point of view, this smaller number of invalidations verses updates gives the Berkeley protocol the advantage with respect to the number of bus transactions required to keep the caches coherent.

Local scheduling greatly reduces the amount of invalidations performed in the Berkeley protocol as the number of processors increase. This is primarily due to fewer processes executing on each processor. This reduction produces fewer displacements in each cache. This will increase the amount of write hits to dirty data lines and thereby reduce the amount of invalidate traffic. Local scheduling eliminates all the update traffic generated by Firefly since there is no way for duplicate lines to exist in multiple caches. Therefore, in a local scheduler, the Firefly approach would generate fewer bus consistency cycles compared with the Berkeley approach.

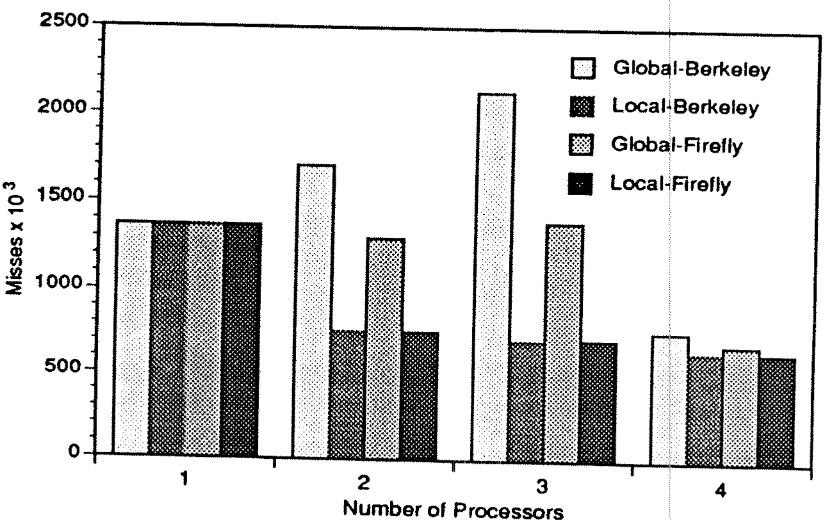

Figure 2: Total misses in a 32K physical cache.

Virtual Verses Physical Cacheing

A virtual cacheing strategy produced consistently more cache misses over the physical caches in every instance except for the four processor cases. This occurs because virtual addresses between processes conflict more than physical addresses. Physical addresses are spread out by the paging algorithm while virtual addresses tend to occupy the same page space between processes. This is especially true within the instruction space

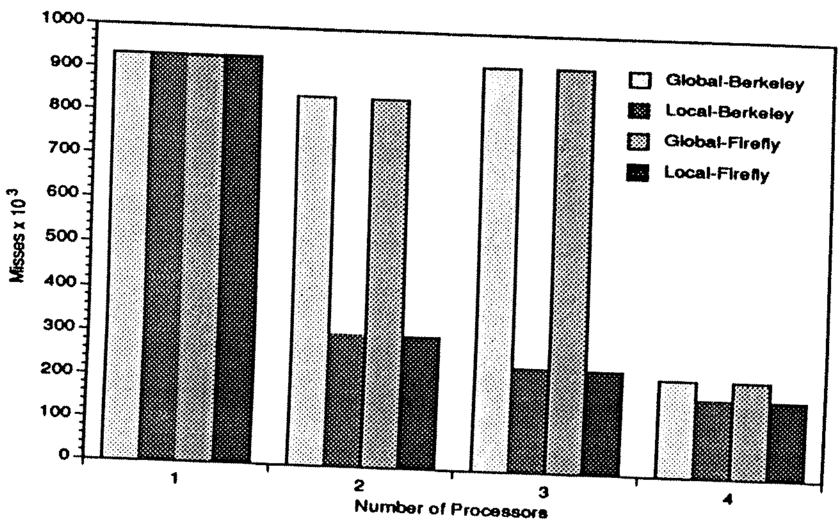

Figure 3: Transient misses in a 32K physical cache.

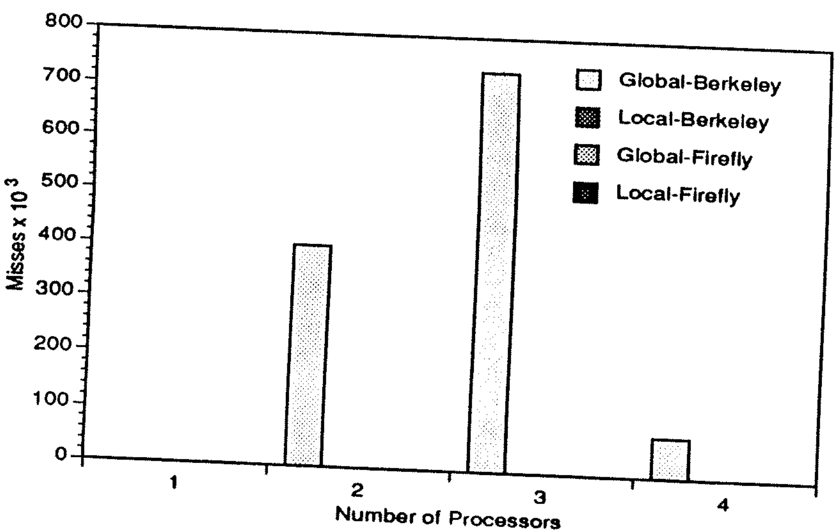

Figure 4: Invalidate misses in a 32K physical cache.

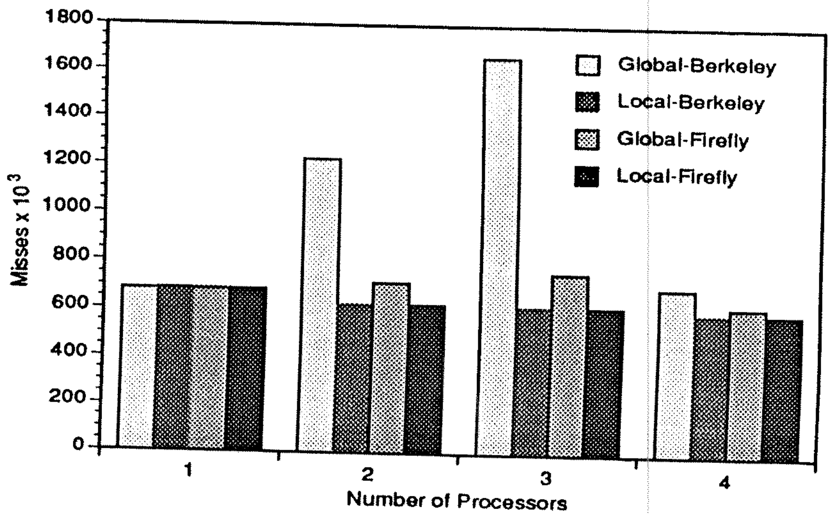

Figure 5: Total misses in a 128K physical cache.

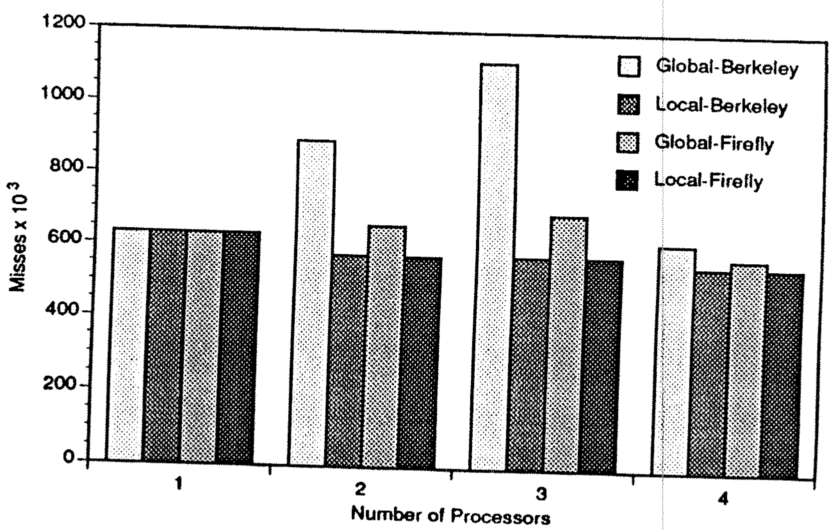

Figure 6: Read misses in a 128K physical cache.

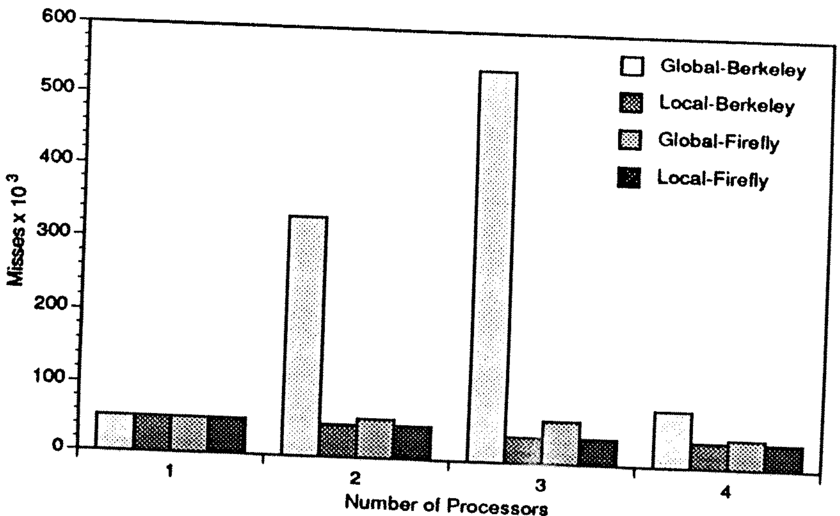

Figure 7: Write misses in a 128K physical cache.

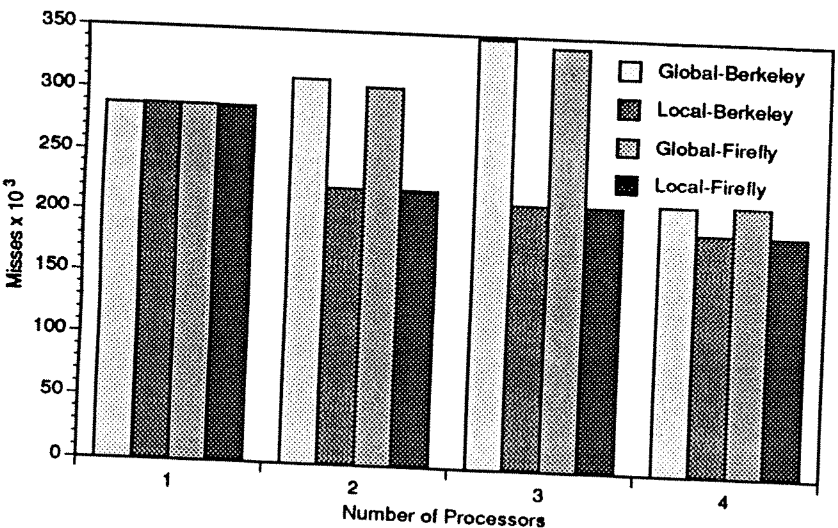

Figure 8: Transient misses in a 128K physical cache.

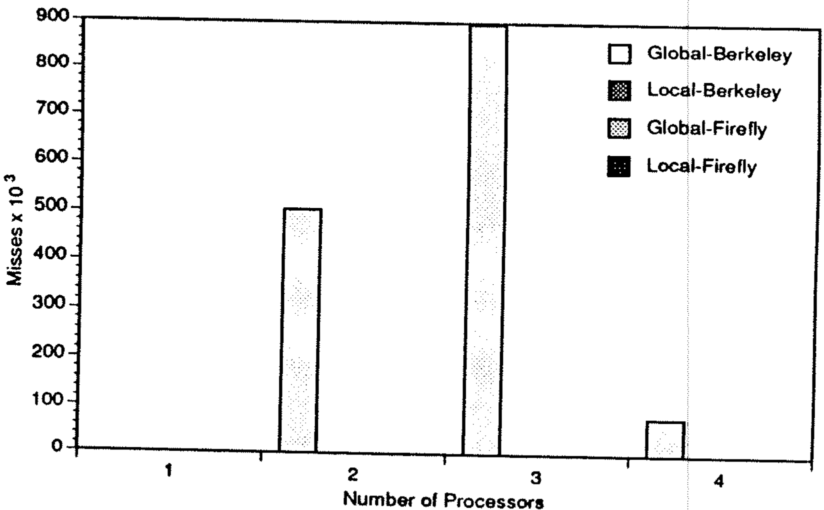

Figure 9: Invalidate misses in a 128K physical cache.

since most code is condensed at the beginning of each virtual address space. The read case in Figure 11 illustrates this well when compared to the write case in Figure 12 and the corresponding graphs in the physically addressed 128K word cache (Figures 6 and 7). The read case produced many more misses relative to the physically addressed cache while the write case produced a modest increase over its physical counterpart. This is primarily due to the interference of the virtual I-streams and the virtual stack regions of the processes. The virtual stack regions also start at the same location in virtual address space. The interference of virtual address spaces can clearly be seen by examining the transient misses of the virtual cache in Figure 13 and comparing them with the misses of the physical cache in Figure 8. The transient misses are much higher in the virtual cache with the exception of the four processor simulation.

One should expect that the global scheduling of a large number of processors would aggravate the difference between physical and virtual caches even more because virtual caches suffer from greater process in-

terference than physical caches. This is indeed true and can be seen by comparing the two graphs of total misses for virtual and physical caches, Figures 5 and 10. With three processors the global scheduler for the virtual cache produces twenty-five percent more misses than its physical counterpart while only five percent more with the local scheduler. Since Firefly generates no misses due to invalidates, its percentage difference is greater when compared with Berkeley (63 percent more misses in the virtual cache).

In the four processor case, each process had a processor to itself for most of its execution time. Only when one process terminated would migration start up again. Since very little displacement from competing processes occurred in the four processor case, the only significant displacement of cache lines was from the process itself. The virtual caches reduced the amount of self displacement by keeping the page addresses contiguous. Physical caches tend have a random distribution of page addresses and can cause greater amounts of displacements as pages are mapped so that they compete for the same cache line. While this is detrimental when there is little interference between processes, this miss rate penalty is more than offset when there is significant interference between competing processes as seen above.

Physical caches generate more update traffic than the corresponding virtual caches in multiprocessor environments. This can probably be attributed to the higher interference of virtual caches. Since it is less likely for data to remain in a cache (higher interference), it is less likely that an update is required.

CONCLUSIONS

This study examined process migration effects with respect to two scheduling algorithms and two cache coherency algorithms. Local and global scheduling were used to illustrate the extremes of scheduling algorithms and update (Firefly) and invalidate (Berkeley) based cache coherency schemes were chosen as the extremes of bus-based coherency.

The results illustrated that scheduling algorithms have a significant impact on the amount of cache misses in a multiprocessor system. As processes migrate they interfere with one another and increase the amount

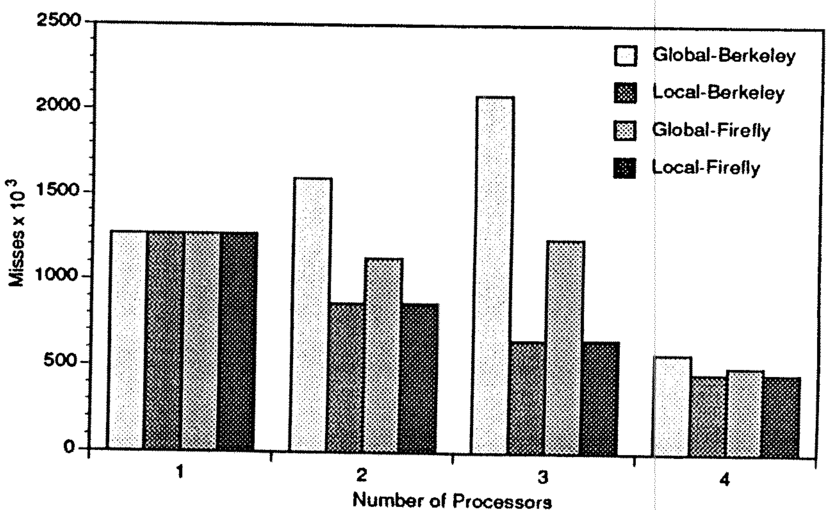

Figure 10: Total misses in a 128K virtual cache.

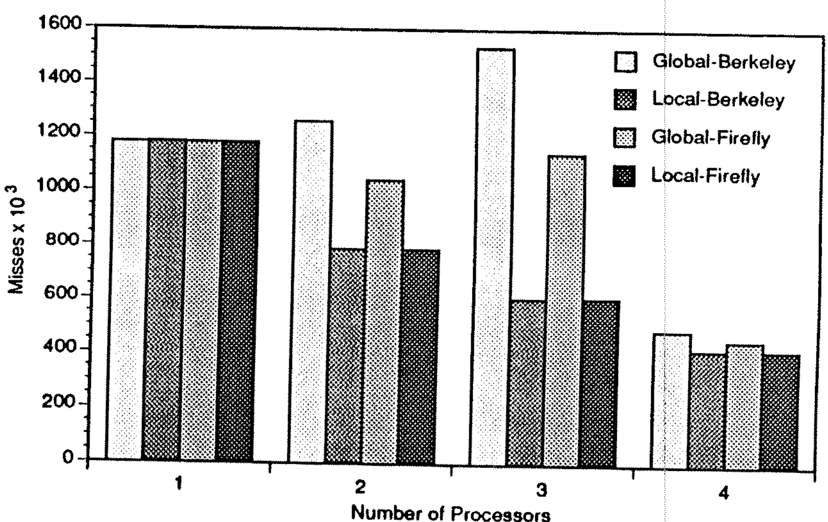

Figure 11: Read misses in 128K virtual cache.

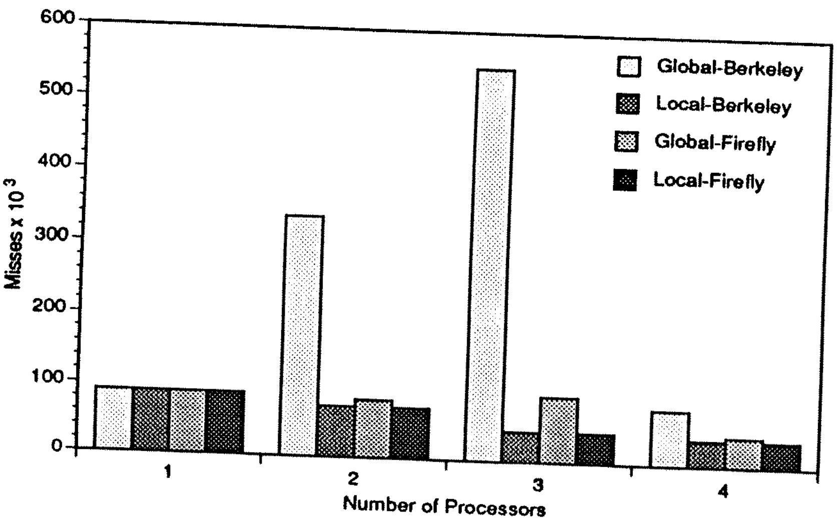

Figure 12: Write misses in 128K virtual cache.

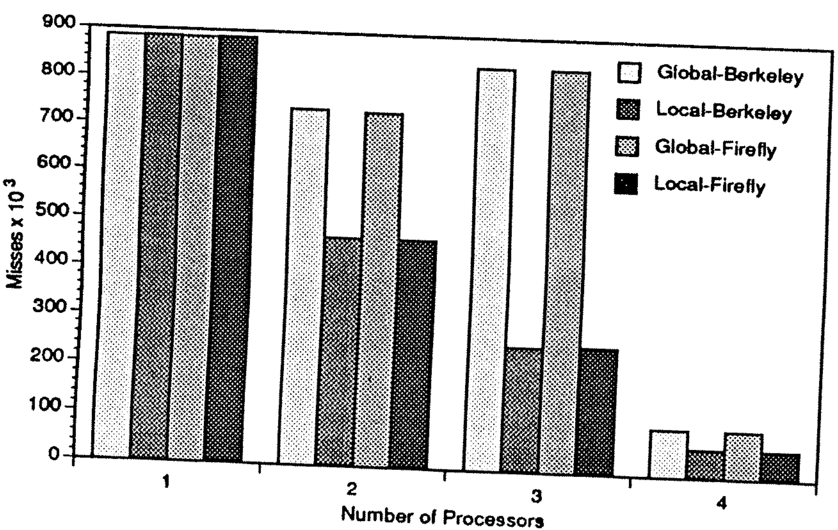

Figure 13: Transient misses in 128K virtual cache.

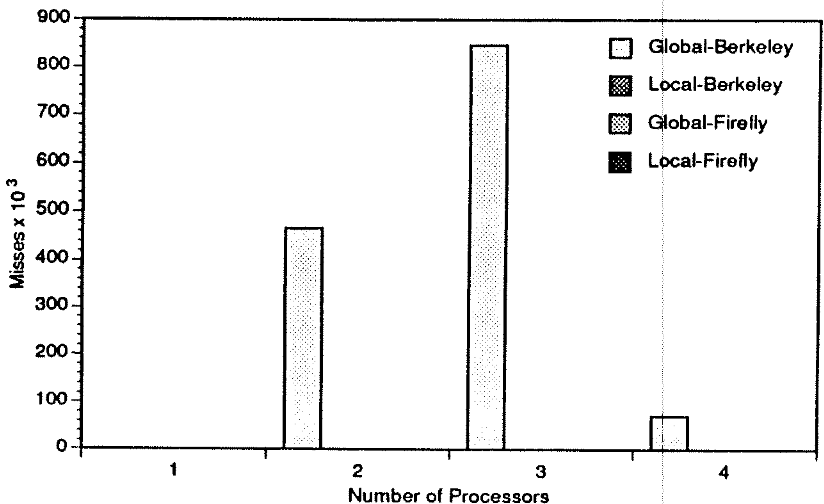

Figure 14: Invalidate misses in 128K virtual cache.

of transient misses per process. Local schedulers reduce these transient misses and eliminate the misses due to invalidations in write invalidate protocols. However, as cache sizes grow, the advantages of local scheduling are diminished because process interference is reduced as each process' addresses are spread across the larger caches. Since local scheduling eliminates misses due to invalidates, the advantages of update protocols over invalidate protocols are eliminated with respect to miss rates. On the other hand, local scheduling eliminates the update bus traffic while only reducing the invalidate traffic (the Berkeley algorithm does not determine when data is exclusively held by a cache).

Firefly provided the best miss rates for the global scheduling approaches since it generates no invalidations that could later force a miss. This is misleading though, since the elimination of invalidates requires update cycles to keep the cache lines consistent with each other. As seen from the global scheduling simulations, the amount of update cycles greatly exceeded the amount of invalidation cycles. With respect to local schedulers, both the Firefly and Berkeley protocols produced equal

miss rates. In this scenario, Firefly performed better since it generated no extra bus traffic to maintain bus coherency.

Virtual caches performed significantly worse when compared to the physical mapped (hashed) caches. Increased process interference was responsible for this and can be explained by the overlapping instruction and stack regions in the virtual address space. Physical mappings tend to randomly distribute page addresses in the caches which reduces process interference. Relative to physically addressed caches, virtual caches increase processor interference yet show no increase in misses due to invalidates. This tends to decreases the difference between update and invalidate coherency protocols because misses due to invalidations were responsible for this difference in the first place. Increased processor interference also makes local scheduling strategies even more attractive for virtual caches by providing processor affinity to each process.

FUTURE WORK

No tools were available to generate kernel traces in this study. Since nany kernel routines execute during each benchmark execution, the address references made by these routines are likely to remain in each proessor's cache, especially if the kernel address space is shared. A study f the multiprocessing behavior of programs with their corresponding kernel calls could provide a much greater understanding of the complete rocess migration effects on cacheing. Lower cache miss rates should be seen because more data in each cache is likely to be reused by each process.

Also, context switching was approximated using a Poisson distribution. In reality, context switches are produced by a combination of stem calls and time-slice expirations. A more accurate model of context switch behavior could identify which system calls generate a context switch. This model could isolate which addresses in the trace correspond these system calls and generate a context switch when such an address encountered. The model would still implement a context switch due time-slice expiration for the case when no change of context occurs fore the time-slice expires.

Another path of investigation corresponds to Stone's [1] assertion that

cache misses can be reduced if the processor stops allocating cache lines when a process is close to the end of its time-slice. This type of behavior would be fairly easy to investigate with the tools developed for this study.

Finally, the simulator used here could easily be modified to a particular instance of a scheduling algorithm. Operating system designers then can create their favorite scheduler and simulate it with these tools. The benefits or limitations of such an new algorithm can be compared with the more conventional algorithms.

ACKNOWLEDGEMENTS

The authors wish to thank Maureen Ladd for her help in preparing this chapter and the Digital Equipment Corporation, who supported Andrew Ladd on a GEEP scholarship.

References

[1] H. Stone. Cache allocation strategies for competing processes, February 1990. Presentation in Distinguished Lecture Series at University of Michigan.

[2] S. Thakkar and M. Sweiger. Performance of an OLTP application on the Symmetry multiprocessor system. In *Proceedings of the 17th International Symposium on Computer Architecture*, volume 1, pages 228–238. IEEE, 1990.

[3] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3), Sepetember 1982.

[4] M. Kobayashi and M. MacDougall. The stack growth function: Cache line reference models. *IEEE Transactions on Computers*, 38(6), June 1989.

[5] A. Borg, R. Kessler, G. Lazana, and D. Wall. Long address traces from RISC machines: Generation and analysis. WRL research report 89/14, Digital Equipment Western Research Laboritory, Palo Alto, California, September 1989.

[6] M. Papamarcos and J. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, volume 1, pages 348–354. IEEE, 1984.

[7] J. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In *Proceedings of the 11th International Symposium on Computer Architecture*, volume 1, pages 355–362. IEEE, 1984.

[8] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[9] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, volume 1, pages 280–289. IEEE, 1988.

[10] R. Clapp, T. Mudge, and J. Smith. Performance of parallel loops using alternative cache consistency protocols on a non-bus multiprocessor. In M. Dubois and S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*, pages 131–152. Kluwer, 1990.

[11] J. Larus. Abstract execution: A technique for efficiently tracing programs. Technical report, University of Wisconsin-Madison, February 1990.

[12] MIPS. *System Programmer's Package (SPP) Reference.* MIPS Corp., Boston, 1988.

[13] D. Samples. Mache: No-loss trace compaction. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, volume 1, pages 89–97. IEEE, 1989.

[14] S. Laha, J. Patel, and R. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11), November 1988.

[15] R. Sites and A. Agarwal. Multiprocessor cache analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, volume 1, pages 186–195. IEEE, 1988.