Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System

Josep Torrellas, Anoop Gupta, and John Hennessy
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Good cache memory performance is essential to achieving high CPU utilization in shared-memory multiprocessors. While the performance of caches is determined by both application and operating system (OS) references, most research has focused on the cache performance of applications alone. This is partially due to the difficulty of measuring OS activity and, as a result, the cache performance of the OS is largely unknown. In this paper, we characterize the cache performance of a commercial System V UNIX running on a four-CPU multiprocessor. The related issue of the performance impact of the OS synchronization activity is also studied. For our study, we use a hardware monitor that records the cache misses in the machine without perturbing it. We study three multiprocessor workloads: a parallel compile, a multiprogrammed load, and a commercial database. Our results show that OS misses occur frequently enough to stall CPUs for 17-21% of their non-idle time. Further, if we include application misses induced by OS interference in the cache, then the stall time reaches 25%. A detailed analysis reveals three major sources of OS misses: instruction fetches, process migration, and data accesses in block operations. As for synchronization behavior, we find that OS synchronization has low overhead if supported correctly and that OS locks show good locality and low contention.

1 Introduction

Cache-based shared-memory multiprocessors rely heavily on cache memories to bridge the difference in speed between processors and main memory. This crucial role of caches is well known to developers of parallel applications and writers of parallelizing compilers, who tune their algorithms for cache performance. In contrast, there is little published literature on how the references of a multiprocessor operating system (OS) affect cache performance. This lack of literature is partially due to the technical difficulty of reliably measuring OS activity. Indeed, because of the complexity and real time nature of the OS activity, these measurements cannot usually be taken from machine simulators; a real machine is required. In addition, to avoid perturbing the machine being measured, sophisticated hardware or software support is required. A second reason for the lack of data regarding multiprocessor OS references is that researchers have traditionally focused on the performance of compute-intensive applications, which entail negligible OS activity. Other common loads, however, like commercial and software-development loads, may require significant OS activity.

Several researchers have pointed out the importance of the cache performance of the OS in sequential machines. First, Clark [7] reports that the cache performance of the VAX-11/780 measured with a performance monitor is lower than the one predicted with application-only traces. Second, in a simulation using traces of VAX memory references, Agarwal *et al* [1] show that the OS can be responsible for over 50% of the cache miss rate. Related work by Ousterhout [14] and Anderson *et al* [2] suggests that OS activity is making an increasing impact on the performance of machines. This OS activity causes cache misses directly and also indirectly in the applications by displacing the state of the applications from the cache. In work based on application-only simulations, Mogul and Borg [13] and Gupta *et al* [9] show the importance of preserving and reusing the cache state of applications. Given all these previous observations, our goal in this paper is to provide an in-depth experimental characterization of the cache performance of a multiprocessor OS.

Our experiments characterize the cache performance of the IRIX OS running on the Silicon Graphics POWER Station 4D/340, a multiprocessor with four 33 MHz MIPS R3000 CPUs. In our experiments, we use a hardware monitor to capture all instruction and data cache misses for each CPU. By using a hardware monitor, we capture the complete system behavior without any measurement-induced perturbation. The experiments consist of running each of three parallel workloads for 1-2 minutes. Two of the workloads are common engineering workloads: one is a parallel compile of 56 files, the other a parallel numeric program running concurrently with the parallel compile and five screen edit sessions. The third workload is an Oracle database.

Our results show that cache misses in the OS stall CPUs for 17-21% of their non-idle time. This stall time reaches 25% if we add application misses induced by OS interference in the cache. In our analysis, we identify three main sets of OS misses: instruction misses, data misses due to process migration, and data misses in block operations. These misses stall CPUs for about 10%, up to 4%, and up to 6% of their time respectively. Finally, we also discover that OS synchronization displays good cache performance if supported correctly.

This paper is organized as follows. Sections 2 and 3 describe the experimental setup and the workloads evaluated respectively. Section 4 characterizes the cache performance of the OS. This section starts out with a high-level analysis of the behavior of OS misses in Subsection 4.1, then analyzes the sources of OS misses in Subsection 4.2, and finally considers application misses induced by the OS in Subsection 4.3. Next, Section 5 characterizes the synchronization performance of the OS. Section 6 discusses the implications of our results for larger machines. Finally, we conclude in Section 7.

2 Experimental Environment

In this section, we discuss the hardware and software infrastructure used in our experiments.

2.1 Hardware Setup

Our results are based on the analysis of address traces generated by a Silicon Graphics POWER Station 4D/340 [4], a bus-based cache-coherent multiprocessor with four CPUs. Each CPU is a MIPS R3000 with a 64 Kbyte instruction cache and a two level data cache: a 64 Kbyte first-level and a 256 Kbyte second-level cache. All caches are physically-addressed, direct-mapped, and have 16 byte blocks. The system is configured with 32 Mbytes of main memory.

We use a hardware monitor to record all bus activity without affecting the machine. A buffer in the monitor stores the physical address and ID of the originating processor for over 2 million bus transactions. Synchronization accesses are not stored since they are diverted to a special synchronization bus and are therefore invisible to the monitor. We discuss a scheme to address this problem later. Time is measured with a granularity of 60 ns using a counter in the monitor. For the machine's cache configuration, bus transactions fill the trace buffer in 0.5 to 4 seconds, depending on the miss rate of the workload.

To circumvent the constraint imposed by a trace buffer of limited size, we periodically suspend and restart the programs that form the workload being measured. A master process starts them and then goes to sleep. At regular intervals, the master wakes up and checks the trace buffer. If the fraction of the trace buffer that is empty is less than a threshold value, then the master suspends all processes, therefore sending the CPUs to the idle loop. Then, the master dumps the trace to disk and restarts the processes. The value of this threshold is chosen so that the buffer never overflows. With this approach, we can trace an unbounded continuous stretch of the workload instead of having to rely on samples interrupted when the trace buffer is being dumped.

This setup requires some extra support. First, the master is given real time priority, the highest priority possible. As a result, when the master wants to run, it is never held back by any other process. Second, in the original OS, the OS checks for suspend signals only in the code that schedules processes. To allow the CPUs to detect the suspend signal immediately after the master issues it, rather than on expiration of the time quantum, we modified the system call that sends the suspend signal to force all CPUs to reschedule. The resulting fast response to the suspend signal prevents the loss of traces.

Instead of dumping the trace onto a local disk, the master process sends the trace to a remotely-mounted disk. In the remote machine, another program postprocesses the trace in parallel while the next segment of the trace is being generated and transferred. With this setup, the activity of the postprocessing program does not pollute the caches and memory of the system under measure. In fact, in the measured system, only two sources of perturbation occur. One is the activity of the master process, which we optimized to require as few cache blocks as possible when checking the status of the trace buffer and when dumping the trace. The second is the activity of the network deamons while the data is being transferred across the network. These deamons partially destroy the I and D-cache state of the processor on which they run (processor 1 on the SGI 4D/340). The perturbations caused by these deamons are negligible, however, since the workload runs for 0.5-4 seconds once tracing is resumed, while filling completely empty I and D-caches takes only about 20 ms.

2.2 Software Support

In all of our experiments, we use release 3.2 of IRIX, the OS that is shipped with the multiprocessor. IRIX is in turn based on UNIX [3] System V and is fully multithreaded except for network functions, which run on CPU 1. All OS data is shared by all threads.

We instrument the OS code to record a variety of events:

- Entries and exits from the OS.
- ID of the processes that are currently running.
- Changes in the per-CPU TLBs, to be able to translate the physical address traces back to virtual addresses. The virtual addresses are required, for example, to determine whether an application reference is an access to the instruction or data cache.
- Entries and exits from interrupts.
- Cache flushing and other events that we need for particular experiments.

In addition to all this information, which is gathered at run time, we need to determine the state of the machine when tracing starts. To this end, a system call dumps the contents of the TLBs and some process state onto the trace buffer when tracing starts.

Because our hardware monitor stores addresses only, and not data, all the information listed above that we transfer to trace has to be encoded as accesses to addresses that the postprocessing program can distinguish. For our experiments, we devised an encoding that allows us to transfer any information to the trace as cheaply as one or more cache misses. This encoding is based on using two hardware features. First, the MIPS address space allows the OS to access physical addresses directly, bypassing the TLB. Second, the hardware allows certain accesses to bypass the two levels of caches. Therefore, we choose a range of physical addresses where only OS code ever lives and generate uncached byte reads to odd (i.e. not even) addresses in that range. These *escape references* cannot be confused with real code accesses because they read odd addresses.

While some of these escape references may be as simple as a single byte read – for example, reading the address that signals *Entering the OS* – others require more state. For instance, when we want to record that a new entry is added to the TLB, we send to the trace four pieces of information: the index of the TLB entry, the number of the virtual page added, the number of its corresponding physical page, and the ID of the process that owns the page. To transfer this information, we first read the location that signals *TLB entry change*. Then, one at a time, we take each of the four pieces of information to send, shift them left one bit, set the least significant bit to make the data odd, take the resulting value as an uncached physical address, and then byte-read from it. Although the resulting four addresses can be anywhere in the address space, the postprocessing program will easily distinguish each escape in this *escape sequence* by looking for four loads from odd addresses by this CPU in the trace following the *TLB entry change* access (remember that the CPU ID is included in the trace). All other misses generated by this CPU while sending the escapes are instruction misses necessarily and hence access even addresses only. To ensure that process rescheduling does not interrupt the escape sequence, we disable interrupts while dumping escape sequences. To summarize, we can transfer to the trace any amount of information as cheaply and non-intrusively as one or more cache misses.

Some extra instrumentation may be required when we want to determine what OS data structure or OS code sequence causes a given cache miss. In general, we compare the address missed on with the entries in the symbol table of the OS image. This approach, however, does not work for the data structures in the OS that are dynamically allocated. To solve this problem, when we want to measure the cache behavior of these data structures, we instrument all entries and exits in all OS subroutines. This approach provides the extra information of what subroutine was executing while a data miss occurred. With this information, we are usually able to determine what dynamic data structure was involved in the miss.

Finally, we can also measure synchronization activity with little intrusion. Since synchronizing accesses are diverted to a synchronization bus, they are invisible to our hardware monitor. To measure them, we do the following. First, we modified the OS to keep statistics on its synchronization activity while running. Second, we modified the OS to allow user processes to map the physical pages that contain these statistics into the processes' address space. As a result, if a user process maps these pages, it can, at any time, read the statistics that the OS keeps on synchronization. Therefore, to measure the OS synchronization performance of a workload, we run a special user process that maps these pages and compares the synchronization statistics before and after an uninterrupted execution of the workload.

3 Workloads Evaluated

The choice of what workloads to use is a major issue for any study of this type because of its impact on the results. We chose three parallel workloads. Two of them are common engineering workloads: one is a parallel compile, the other a parallel numeric program running concurrently with a parallel compile and some screen edit sessions. Our third choice is a typical commercial workload, namely an Oracle database. We now describe each workload in detail.

- *Pmake* is a parallel make of 56 C files with, on average, 480 lines of code each. The files are compiled such that, at the most, 8 jobs can run at once (*-J* flag is 8). While this workload has some compute-intensive periods when the optimizing phase of the compiler runs, it usually exhibits heavy I/O activity.
- *Multpgm* is a timesharing load composed of a numeric program plus *Pmake* and five screen edit sessions. All programs are started at the same time. The numeric program, called *Mp3d* [11], is a 3-D particle simulator used in aeronautics and run using four processes and 50000 particles. Each edit session is created as follows. An input file with *ed* commands is fed to a program that simulates a user typing at a terminal, and the resulting command piped to an *ed* invocation. The program that simulates the user simply sends the characters in the input file to the screen and to the pipe in bursts of 1-15 characters at a time. A call to *rand()* determines the number of characters to be sent at a given time. At the most, however, 25 characters can be sent every five seconds. The commands in the input file force the *ed* session to do character searches and text editing.
- *Oracle* is a scaled down instance of the TP1 database benchmark [8] running on an Oracle database. We do not run the standard-sized benchmark because we have limited memory and disk resources and I/O would be a heavy bottleneck. Instead, we reduce the size of the benchmark so that it fits in main memory. The resulting benchmark is not standard: it has 10 branches, 100 tellers, 10,000 accounts, and achieves 59 transactions per second (TPS).

  To see if the size of the database affects the cache performance of the OS, we ran a subset of the experiments using a standard-sized benchmark. We show in [18] that the characteristics of the OS misses in the standard benchmark are qualitatively the same as the ones in *Oracle*.

3.1 Overview of the Cache Behavior

We traced each of the workloads for 1-2 minutes. Table 1 presents some of the workload characteristics that show the importance of the OS. First, in columns 2-4, we divide the execution time of the workloads into user, system, and idle time. From the table, we see that the OS accounts for as much as 32-47% of the non-idle execution time in these workloads. Next, in column 5, we consider the fraction of OS misses in the workloads. From the table, we observe that this fraction varies from 25 to 50%.

We now consider the performance impact of the cache misses. The last three columns of Table 1 show the fraction of time processors waste in stalls due to all application and OS misses, due to OS misses only, and due to OS plus OS-induced application misses respectively. OS-induced application misses result from the OS displacing the application from the caches. We compare the stall time against the non-idle execution time because the amount of time that CPUs spend in the idle loop is a function of the memory and disk resources in the machine. The stall time is estimated by assuming that each bus access stalls the CPU for 35 cycles, a number slightly over the zero-contention latency of a memory access in the machine. This estimate does not handle two situations that occur in the real machine. First, processors can potentially overlap a write miss with computation. Second, processors may be stalled on a miss and we may not be aware of it. This may happen when a CPU misses in the first level cache but the bus is not accessed because the miss hits in the second level data cache. In this situation the CPU could be stalled for about 15 cycles. If the first situation dominates, our results on miss stall times are excessive; if the second dominates, our results are conservative. These two effects, however, only apply to data misses, which account for 45% of the misses in the workloads. In addition, we will see that data misses in the OS are often caused by the OS sweeping through blocks of data (Section 4.2.2). This effect may cluster the misses in time, fill the write buffer, and prevent writes from being overlapped with other computation. For these reasons, we believe that our results do not overestimate the amount of stall time.

We observe from column 6 of Table 1 that cache misses stall CPUs for 40-60% of their time. Column 7 shows that the stall time due to OS misses is as large as 17-21%. In the following section, we analyze what causes this stall time. Finally, column 8 shows that the stall time caused by both OS and OS-induced application misses reaches 25%.

4 Characterization of the Cache Performance of the Operating System

We start this section on the cache performance of the OS with a high-level view of the behavior of OS misses. After that, the bulk of this section characterizes the major sources of OS misses. Finally, we consider the misses in the application caused by OS interference.

Table 1: Characteristics of the workloads.

Workload	Execution Time			OS Misses / Total Misses (%)	Appl. + OS Miss Stall Time / Non-Idle Time (%)	OS Miss Stall Time / Non-Idle Time (%)	OS + OS Induced Miss Stall Time / Non-Idle Time (%)
	User (%)	Sys. (%)	Idle (%)				
Pmake	49.4	31.1	19.5	52.6	39.9	21.0	25.8
Multpgm	53.2	46.7	0.1	46.3	46.5	21.5	24.9
Oracle	62.4	29.4	8.2	26.6	62.5	16.6	26.8

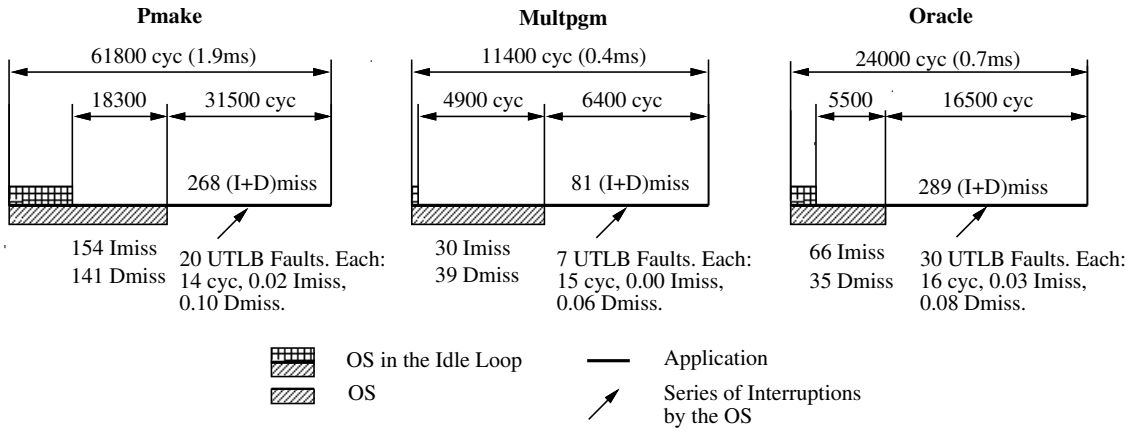

Figure 1: Average times and cache misses in the basic pattern that repeats throughout the trace. Time is measured in 30 ns processor cycles. Within each figure, distances are drawn to scale.

4.1 High-Level View of the Cache Activity of the Operating System

To understand the cache performance of the OS we first need to understand the interleaving of OS and application activity. In this section, we characterize the interleaving of OS and application invocations, the duration of these invocations, and the number of misses involved. This data is also useful to build analytic models of OS and application referencing activity.

Our measurements show that the OS interrupts the application in two ways. First, with frequent spikes of activity that are nearly miss-free; second, with relatively infrequent and long bursts of activity where most of the OS misses occur. The former are generated by TLB faults that only require copying a virtual to physical page association from an OS data structure to the TLB (UTLB faults). The latter we call OS invocations and are caused by system calls, interrupts, and other TLB faults.

Figure 1 shows the average characteristics of the basic pattern of execution that repeats throughout the trace. In the figure, the execution time is divided into *OS*, *OS in the Idle Loop*, and *Application*. In the upper part of the figure we show time duration in cycles. In the lower part we show numbers of misses. Finally, below the arrows, we show the number of UTLB faults that occur in an application invocation, as well as their cost in cycles and misses. In the figure, while the miss counts are exact, the cycle counts are distorted by our instrumentation: they are 1.5%, 5%, and 7% larger than they should be in *Pmake*, *Multpgm*, and *Oracle* respectively.

Several observations may be made from Figure 1. First, while the UTLB fault handler is invoked frequently, it causes very few cache misses and is very fast. On average, one invocation causes less than 0.1 misses. In addition, it can be shown that the distributions for the number of I- and D-misses generated in one invocation are strongly skewed towards very small values [18]. Overall, from the cost of UTLB faults shown in Figure 1, we compute that UTLB fault handling takes the equivalent of 1.5% of application cycles.

Second, the OS is invoked on average as frequently as once every 1.9 ms in *Pmake*, 0.4 ms in *Multpgm*, and 0.7 ms in *Oracle*. These intervals are an order of magnitude smaller than the 10 ms period of the OS clock. The small size of these intervals is due to the high frequency of several OS operations. For the case of *Multpgm*, Figure 2 shows the relative frequency of the operations executed by the OS without including UTLB faults. Although more than one operation can be performed per OS invocation, – for example two nested interrupts – we see that several types of operations occur more frequently than clock interrupts. We see that about 50% of the OS operations are system calls associated with synchronization (sginap system calls), about 20% are TLB faults, about 20% are I/O system calls, and only 5% are clock interrupts. The sginap system call is issued by the synchronization library after 20 unsuccessful attempts to acquire a lock. This call reschedules the CPU, in the hope of giving the process that holds the lock a chance to run and release the lock. As we will see in Section 4.2.3, the sginap system call is not common in the other workloads. As a result, the time between OS invocations is larger in the other workloads.

Finally, we note that an OS invocation replaces only a small fraction of the cache contents. For example, the average OS invocation causes 154 I- and 141 D-misses in *Pmake* (Figure 1), a small number compared to the number of blocks in the I- or D-caches. To see the complete picture, Figure 3-(a) and (b) show the distribution of the number of I-misses and D-misses respectively in *Pmake*'s OS invocations. In reality, the fraction of the cache that is replaced in an OS invocation is smaller than that suggested by Figure 3, since from 10% to 25% of the OS misses replace blocks that have already been missed on in the same OS invocation.

For completeness' sake, Figure 3-(c) shows the distribution of

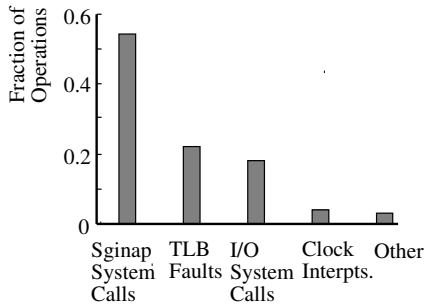

Figure 2: Frequency of the operations executed by the OS invocations in *Multpgm*.

the duration of *Pmake*'s OS invocations. Figure 3 can be used to build an analytic model of the OS activity in *Pmake*. The corresponding charts for *Multpgm* and *Oracle* are shown in [18]. They show that, as in *Pmake*, an individual OS invocation has a small impact on the cache contents. For completeness' sake, [18] also shows distributions for the number of misses and cycles in invocations of the application for *Pmake*, *Multpgm*, and *Oracle*.

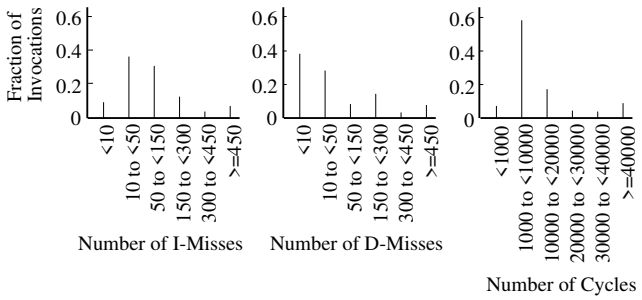

Figure 3: Characterization of the OS invocations in *Pmake*. From left to right, the charts show the distribution of the number of I-misses per invocation, D-misses per invocation, and cycles per invocation. The latter does not include the time in the idle loop.

4.2 Analysis of the Cache Misses in the Operating System

While the previous section showed the behavior of the OS misses, it did not explain the causes for these misses. We focus on the causes in this section. There are three main causes of cache misses in the OS, namely first-time references, displacement of the cache blocks from the cache by other OS references or application references, and coherence activity. Coherence misses in the data cache are the result of data sharing; coherence misses in the instruction cache result from the I-caches being invalidated when physical pages that contain code are reallocated. In addition, we also distinguish a subset of the misses that result from displacement by OS references: the misses where the application was not invoked between the displacing OS reference and the OS miss. These misses are interesting because they can be reduced by restructuring the OS code. This whole classification is summarized in Table 2. In the following, we first analyze the I-misses and then the D-misses.

4.2.1 Analysis of the Instruction Misses

The instruction misses in the OS are classified into their subcomponents in Figure 4-(a). In the figure, the total number of OS misses is normalized to 100. The most striking fact in Figure 4-(a) is that instruction misses constitute as much as 40-65% of all OS misses. Instruction accesses, therefore, are the first of the major sources of OS misses that we identify. Using the same method as in Table 1, we estimate that these OS misses stall CPUs for 10.9, 9.2, and 10.6% of their non-idle time in *Pmake*, *Multpgm*, and *Oracle* respectively. This large impact contrasts with the behavior of engineering or scientific applications, where instruction misses are less frequent than data misses and are often discounted in performance studies. This relatively poor I-cache performance of the OS is due to the scarcity of loops in the OS code.

Table 2: Classification of the cache misses in the OS from an architectural perspective. Except for *Dispossame* misses, a given miss belongs to only one class.

Class	Explanation
Cold	Misses generated when a processor accesses a physical memory block for the processor's first time.
Dispos	Misses that occur when the requested data used to be in the cache but has been displaced by an intervening OS reference.
Dispap	Misses that occur when the data requested used to be in the cache but has been displaced by an intervening application reference.
Sharing	D-cache misses resulting from OS data being shared or migrating among processors.
Inval	I-cache misses resulting from invalidation of the I-cache when physical pages that contained code are reallocated. The reallocation of data pages does not require invalidating the D-caches because the snooping hardware automatically updates them.
Uncached	OS accesses that bypass the caches.
Dispossame	*Dispos* misses that occur when the application was not invoked between the displacing OS reference and the OS miss.

A second observation is that the OS often interferes with itself in the I-cache. This is shown by the sizable contribution of *Dispos* misses in Figure 4-(a). To understand this self-interference better, we measure what parts of the OS code cause *Dispos* misses. The result of this experiment for *Pmake* is shown in Figure 5. The figure shows the number of *Dispos* misses as a function of the physical address of the OS routine where these misses occur. In the figure, instead of measuring the X-axis in bytes, we measure it in multiples of the I-cache size (64 Kbytes). We note that these self-interference misses are concentrated in short address ranges and therefore occur mostly in a few routines. We will consider the implications of this fact in the section that studies optimizations.

We note that this self-interference often occurs within the same OS invocation as opposed to across OS invocations. This makes this interference easier to understand and eliminate. The misses resulting from self-interference within the same OS invocation are called *Dispossame* misses. Figure 4-(b) shows the contribution of the *Dispossame* misses to the self-interference misses. We note that the magnitude of this contribution is related to the duration and frequency of the OS invocations. For example, *Dispossame* misses are a larger fraction of *Dispos* misses in *Pmake* than in *Multpgm* because *Pmake* has longer OS invocations than *Multpgm* (Figure 1).

Finally, the contributions of the remaining categories of I-misses in Figure 4-(a) depend on the behavior of the user applications

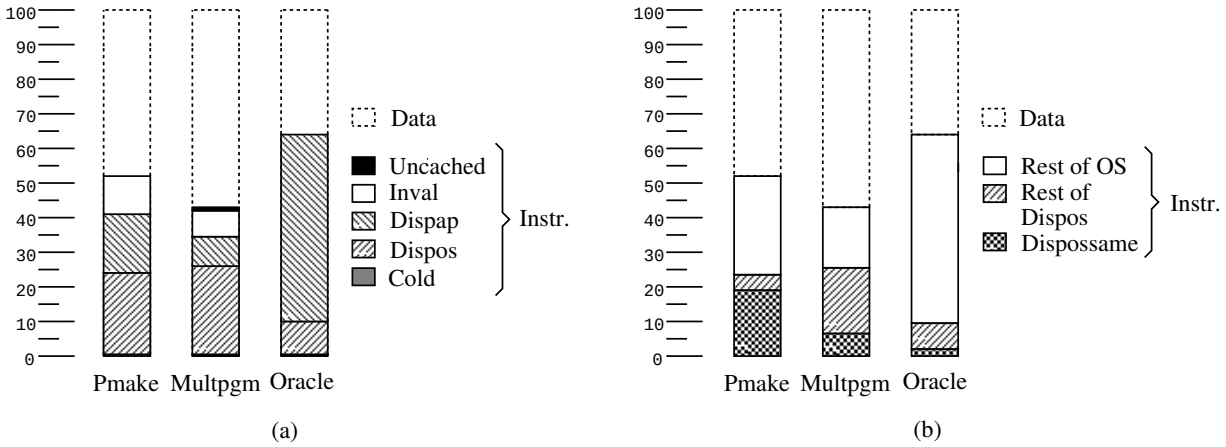

Figure 4: Classification of the instruction misses in the OS. Chart (a) shows the contribution of each class of instruction misses as a fraction of the total number of OS misses. Chart (b) shows the *Dispossame* component of the *Dispos* misses.

running. For example, *Dispap* misses dominate in *Oracle* because the working set of the database code is large and therefore the database interferes with the OS.

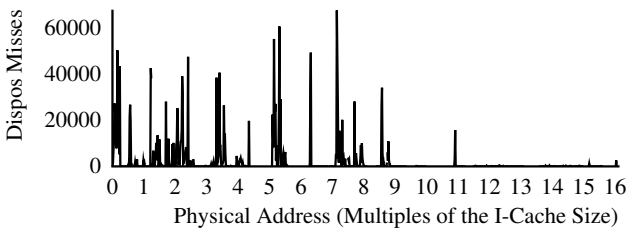

Figure 5: Number of self-interference (*Dispos*) misses in *Pmake*'s OS instructions as a function of the physical address of the OS routine where the misses occur. The X-axis is measured in multiples of 64 Kbytes, the size of the I-cache.

Removing Instruction Misses

One way to improve the hit rate of the OS instructions is to reduce the amount of self-interference. This can be accomplished by purposely laying out the basic blocks in the OS object code to avoid cache conflicts. The thin spikes in Figure 5 suggest that localized changes in this layout may achieve a reduction of misses. The techniques used to optimize the basic block layout, however, should be slightly different from the existing ones. Current techniques [12] are targeted towards code with frequent loop nests. They are based on identifying the most frequently executed loops and then placing the rest of the code to avoid interference with these loops. Techniques for the OS code, instead, should take into account that commonly-executed OS paths often contain a long series of loop-less operations. It is beyond the scope of this paper to consider these techniques.

A second way to reduce these self-interference misses and, in general, all displacement misses, is to increase the associativity of the I-cache. Unfortunately, set-associative caches are slower. Disregarding any speed considerations, however, we simulated the effect of set-associative and larger caches on the I-misses of the OS. The results of the simulation are shown in Figure 6. The figure plots the miss rate of the OS instructions for direct-mapped and two-way set-associative caches of different sizes relative to the miss rate in the machine measured. In our simulations, we use the references that miss in the caches of the real machine to simulate larger caches. For this reason, we cannot simulate a two-way set-associative cache of 64 Kbytes. Note that both application and OS instruction traces are simulated, although only OS misses are plotted in the figure.

We see in Figure 6 that increasing the associativity of the I-cache to two produces a noticeable reduction in OS misses. The effectiveness of set-associativity is not surprising given the amount of cache interference present in OS instructions.

A second observation from the plots is that, naturally, larger caches eliminate an increasing number of misses. The drop in the number of misses is particularly steep in *Oracle* – all the way to 1 Mbyte caches. This effect is due to the large instruction working set of the database. The curves for *Pmake* and *Multpgm*, on the other hand, saturate at 256 Kbytes. This behavior is caused mostly by cache invalidations, which create *Inval* misses. This is illustrated by Figure 6, which shows the effect of the misses caused by cache invalidations for direct-mapped caches. The dashed curve bounds the drop of the relative miss rate curve for direct-mapped caches. From the figure, we observe that both *Pmake* and *Multpgm* are seriously limited by these misses. Note, however, that the figure assumes that the algorithm used to invalidate caches does not change as caches increase in size.

4.2.2 Analysis of the Data Misses

The data misses in the OS are decomposed into their constituent classes in Figure 7-(a). As in Figure 4, the total number of OS misses is normalized to 100. From the figure, we note that the dominant class of data misses is *Sharing* misses. The remaining misses are caused by both cache displacement and cold references. A large fraction of these remaining misses – and a small amount of *Sharing* misses as well – occur when the OS executes block operations. In the following, we first analyze *Sharing* misses and then the misses caused by block operations.

Sharing Misses

To understand what causes *Sharing* misses, we start by dividing these misses into their contributing data structures. This is shown in Figure 8. From the figure, we note that *Sharing* misses are spread over a lot of different data structures. The size of these data struc-

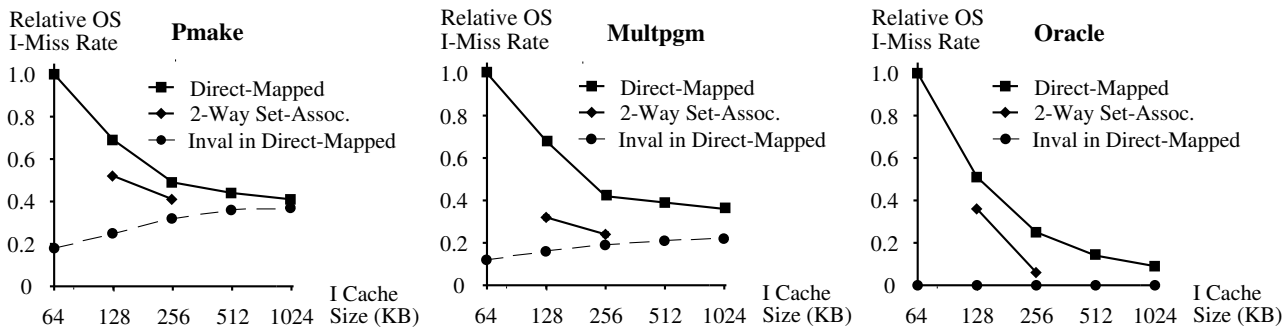

Figure 6: Effect of the size and associativity of the I-cache on the I-miss rate of the OS. For the direct-mapped caches, we also plot the effect of the misses caused by invalidations of the cache (*Inval* misses). The three curves have the same Y-axis.

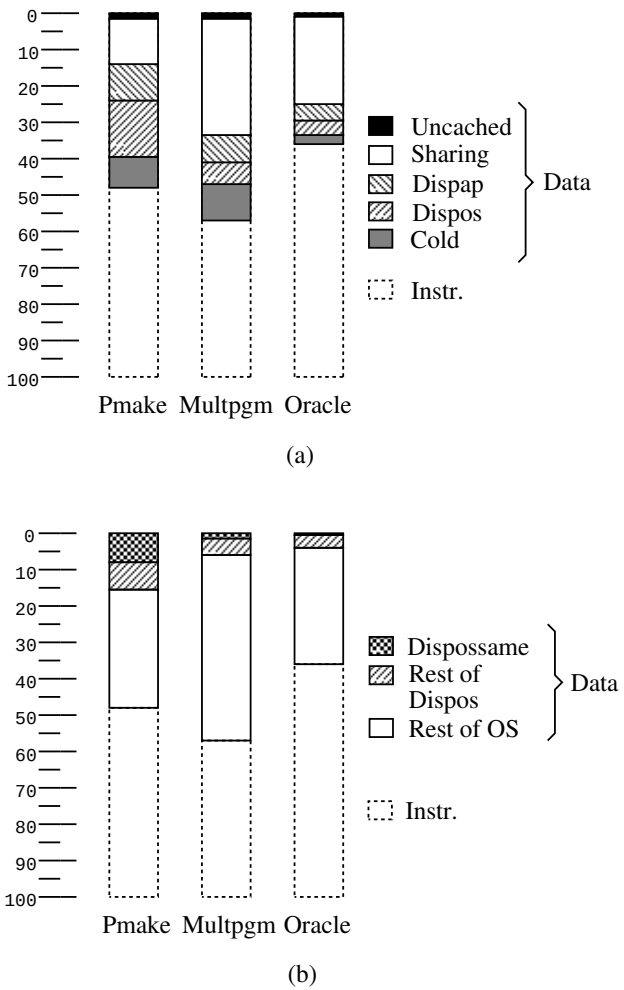

Figure 7: Classification of the data misses in the OS. Chart (a) shows the contribution of each class of data misses as a fraction of the total number of OS misses. For completeness' sake, Chart (b) shows the *Dispossame* component of the *Dispos* misses.

tures and a brief description of their function are shown in Table 3. From the table, we see that these data structures vary widely. For example, some of them are large and seemingly sparsely-shared, while others are small, frequently shared, and can potentially cause hot spots.

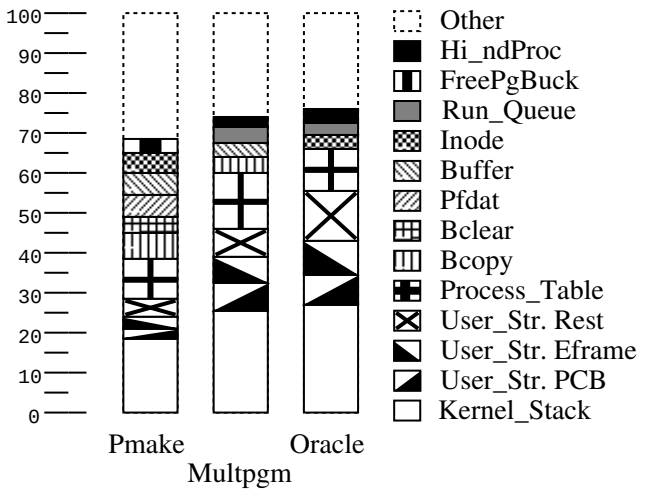

Figure 8: Classification of the *Sharing* misses in the OS according to the data structures that cause them. Each of the data structures under category *Other* accounts for less than the smallest category in the figure.

The major contributing data structures are those that mostly store per-process private state, namely the Kernel Stack, the three components of the User Structure, and the Process Table (See Table 3). Together, they account for 40-65% of the *Sharing* misses. Except for the Process Table in some cases, these data structures store per-process state that is accessed only by the CPU executing that process. If these data structures appear to be shared, therefore, it is because the process migrates among CPUs.

Process migration is our second major source of OS misses. Although process migration possibly contributes to the *Sharing* misses of most data structures of Figure 8 and also causes instruction misses, we conservatively assume that it only causes the *Sharing* misses in the three data structures considered. We call these misses migration misses. As shown in Table 4, these data misses account for 10-44% of the data misses in the OS and slow down our workloads by up to 4%.

Migration misses often occur when the OS manages the run queue, handles exceptions, or sets up read and write system calls.

Table 5 shows the individual contributions of these three operations to the number of migration misses. From the table, we see that these operations account for 25-50% of the migration misses. In the table, the *Management of the Run Queue* category includes the contributions of the seven routines that form the core of the run queue management. These routines save and restore the state of a process, put a process in the run queue, find a process to run, and manage the scheduler.

Table 3: Data structures that contribute the most to the *Sharing* misses in the OS.

Data Structure	Size (Bytes)	Function
Kernel Stack	4096	Stack used by the OS while executing in the context of the process.
PCB section of the User Structure	240	Place where the process registers are saved when a context switch occurs.
Eframe section of the User Structure	172	Place where the process registers are saved when the process gets an exception.
Rest of the User Structure	3684	Contains descriptors for files opened by the process and system buffers allocated for the process, maintains the return values of system calls, etc.
Process Table	46080	Contains the process state, priority, signals, scheduling parameters, etc.
Bcopy	—	Pages or fragments of pages accessed by the block copy routine.
Bclear	—	Pages or fragments of pages accessed by the block clear routine.
Pfdat	210944	Array of descriptors to the physical pages. Each descriptor contains the disk block number corresponding to the page, a pointer to the related inode, links to various lists of pages, etc.
Buffer	17408	Array of buffer headers for the buffer cache. Each buffer contains the disk block number whose data the buffer stores, a pointer to the actual data, links to several lists of buffers, etc.
Inode	68608	Table of memory-resident inodes. Each inode contains the file type, access state, current file position, etc.
Run Queue	24	Structure at the head of the run queue. It contains pointers to several queues and processes.
FreePgBuck	3072	Array of buckets that start hash lists to which free physical pages are tied.
Hi_ndproc	1	Flag used to make some decisions on priority scheduling.

Table 4: Conservative estimate of the data misses and stall time caused by process migration.

Workload	% of OS Data Misses				Proc. Migr. D-Miss Stall Time / Non-Idle Execution Time (%)
	Kernel Stack	User Struc.	Process Table	Total	
Pmake	4.8	2.5	2.6	9.9	1.0
Multpgm	14.4	11.6	7.8	33.8	4.2
Oracle	18.0	19.0	7.1	44.1	2.6

The second category in Table 5, namely *Low-Level Exception Handling*, groups the contributions of the initial and final stages of exception handling. These stages are coded in assembly for higher performance and perform low-level operations like determining what class of exception occurred or saving and restoring the registers. Exceptions include interrupts and TLB faults.

Table 5: Fraction of the migration misses accounted for by three common operations.

Operation	% of Migration Misses		
	Pmake	*Multpgm*	*Oracle*
Management of the Run Queue	11.5	20.5	14.3
Low-Level Exception Handling	7.3	12.9	14.5
Recognition and Setup of Read and Write System Calls	6.4	13.2	20.7
Total	25.2	46.6	49.5

Lastly, Table 5 shows the contribution of the recognition and setup of the read and write system calls. We see that these operations have a noticeable effect in workloads with frequent I/O activity.

As a final note, we observe from Figure 8 that a significant fraction of the migration misses occur while saving and restoring registers in context switches and exceptions (PCB and Eframe categories respectively). This implies that these two simple operations, namely register saving and restoring, have a noticeable performance impact.

Removing Sharing Misses

Larger data caches cannot eliminate *Sharing* misses. Consequently, since *Sharing* misses are the majority of data misses, larger data caches can only moderately increase the data cache performance of the OS.

One way to eliminate some of the *Sharing* misses is to restrict process migration. Process migration not only causes the data misses isolated above; it also causes other data and instruction misses in the OS as well as a (possibly larger) number of misses in the application. Process migration should not be completely eliminated, however, for it ensures load balance in the machine. Affinity scheduling [16, 19, 20] is one technique that removes misses by encouraging processes to remain in the same CPU while still tolerating process migration for load balance.

Misses in Block Operations

The OS often sweeps through large arrays of data, primarily in block copy and clear operations and when traversing the physical page descriptors (*Pfdat*s). As an example, a block copy occurs when a child process writes on a non-shared page that belongs to its parent. In that case, a copy of the page is made for the child. An example of block clear occurs when the OS allocates a page for data. The page has to be zeroed out before being used. Finally, a traversal of the array of page descriptors occurs when free memory is needed. In this case, the OS traverses the array to find out what pages need to be written out to disk.

These three block operations constitute our third major source of OS misses. Table 6 shows that they cause from 10 to 61% of the data misses. These misses are mostly displacement misses (*Dispap* or *Dispos*) or *Cold* misses, although Figure 8 showed that a few of them appear as *Sharing* misses. As seen in Table 6, they stall the CPUs by up to 6%.

These operations are more harmful than Table 6 suggests for two reasons. First, by accessing large data structures, these operations often wipe out a large fraction of the data cache. The data displaced from the cache may have to be fetched again later on. Second, the

Table 6: Data misses and stall time caused by the three block operations.

Workload	% of OS Data Misses				Block Ops. D-Miss Stall Time / Non-Idle Execution Time (%)
	Block Copy	Block Clear	Travers. of Descrip.	Total	
Pmake	17.6	23.7	19.7	61.0	6.2
Multpgm	15.1	7.2	15.7	38.0	4.7
Oracle	8.6	1.0	1.0	10.6	0.6

data fetched by these operations is often not reused. As an example, consider a page copy. The copy operation brings two pages into the cache; one of the pages will probably not be accessed anymore, and only a few words of the second page may be accessed in the near future.

Removing Misses in Block Operations

Block operations often access regular and relatively large blocks. This fact makes potential optimization easier. As an example, Table 7 characterizes the sizes of the blocks copied or cleared in *Pmake*. The table classifies the operations according to the size of the data operated on. For the operations in a given category, the table shows the relative frequency of invocation and one or more examples. From the table, we note that 50% and 70% of the invocations of block copy and clear respectively operate on a full 4 Kbyte page or a large, regular fraction of it.

Table 7: Characterization of the sizes of the blocks copied or cleared in *Pmake*.

Block Oper.	Size of Block	Freq. of Invoc. (%)	Example
Copy	Full Page	5	Update of a copy-on-write page.
	Regular Page Fragment (e.g. 1/4 of Page)	45	Transfer of data in/out of buffer cache.
	Irregular Chunk	50	Copy of strings or system call parameters.
Clear	Full Page	70	Allocation of a page for page table entries. First reference to a demand-zero page.
	Irregular Chunk	30	Initialization of structures allocated in the kernel heap or inode-related data structures.

One way to eliminate misses in block operations is to use special hardware and software support to prefetch data. In this way, if the data to be copied or cleared is prefetched in advance while other computation is in progress, the latency of the misses is hidden. A second technique is to bypass the cache when block transfer operations are performed. In this case, we still pay the cost of the cache miss latency, but do not wipe out other relevant state in the cache with this seldom-reused data. The data accessed with cache bypassing should not be fetched from memory one word at a time, but in blocks of contiguous data. This helps exploit the spatial locality of the reference stream and therefore reduces data transfer costs. Finally, more sophisticated support for block operations has been suggested by Cheriton *et al* [6].

4.2.3 Analysis of the Operating System Cache Misses from a Functional Perspective

For completeness' sake, we now briefly analyze the OS misses from a functional viewpoint. We classify them according to the high-level operation that the OS was executing when the misses occurred. The operations considered are expensive and cheap TLB faults, I/O system calls, the sginap system call, the remaining system calls, and interrupts. These operations are defined in Table 8, and the results of measuring both the data and instruction misses are shown in Figure 9. In the rest of this section, we first analyze the observations and then discuss the implications.

Table 8: High-level OS operations.

Operation	Explanation
Expensive TLB Faults	TLB faults that require the allocation of a physical page. They may involve simply grabbing a page from the list of free pages, sometimes performing a page copy or clear, or they may also require doing I/O to read or write pages to disk.
Cheap TLB Faults	TLB faults that require neither physical memory allocation nor I/O. Conceptually, the OS simply copies some information from global page tables to the TLB. They include UTLB faults.
I/O System Calls	System calls that involve file system reads or writes.
Sginap System Call	System call used by a user process to reschedule the CPU on which it is running. Sginap is called by the synchronization library after a process has been unsuccessfully spinning on a lock 20 times.
Other System Calls	Remaining system calls.
Interrupts	Any interrupt, such as disk and terminal I/O, inter-CPU, or clock interrupts.

Analysis of the Data and Instruction Misses

Let us begin with the data misses, shown in the leftmost chart of Figure 9. Clearly, the majority of them are caused by I/O system calls and TLB faults. The latter are mostly of the expensive type. In *Oracle*, the database requests allocation of pages itself and manages its own file activity. As a result, the expensive TLB fault activity is lumped into the I/O system call category.

The sginap system call is another source of data misses if, as in *Multpgm*, lock activity is common in the workload. The rationale under invoking sginap is that the process that currently holds the lock may not be running. With a CPU reschedule, that process may be picked up from the run queue and eventually release the lock. We note that each invocation of sginap produces only 25 data misses on average; it is the frequent invocation of sginap what makes these misses significant.

Turning to instruction misses in the rightmost chart of Figure 9, we see that I/O system calls are the largest contributors. We also note that, like I/O system calls, interrupts contribute more to the instruction misses than to the data misses in relative terms. The reason is that these two operations execute long stretches of code while referencing relatively few data items. In contrast, expensive TLB activity shows the opposite behavior. The reason is that it is composed of small kernels like *Copy* or *Clear* that reference large chunks of data.

Discussion of the Data and Instruction Misses

The numerous misses on the code executed in I/O system calls suggest that, should code layout optimization be attempted, this part of the OS should be studied first. Indeed, this part of the code

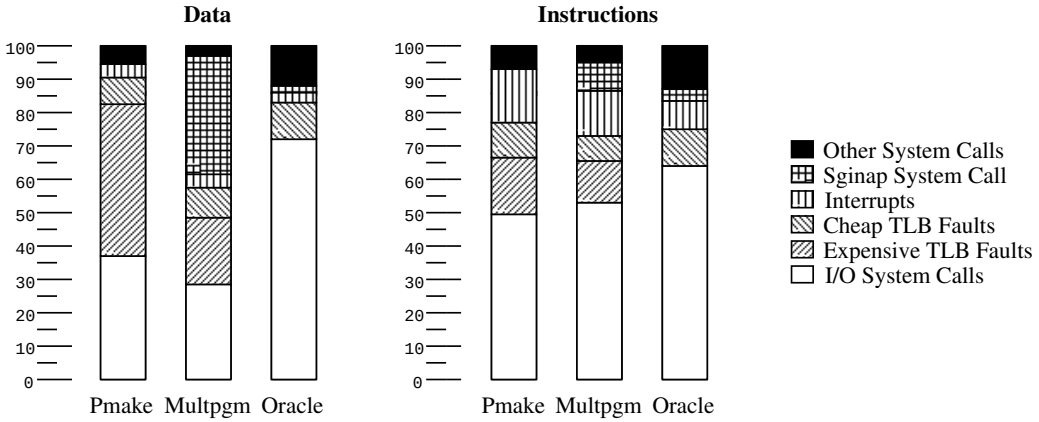

Figure 9: Classification of the OS cache misses according to the high-level operation performed by the OS when the misses occurred.

as it is now can quickly wipe out the whole cache. For example, some I/O drivers have a size comparable to the instruction cache.

A second observation is that cheap faults in our 64-entry fully-associative TLB are responsible for a small fraction of the cache misses only. This suggests that, even if we eliminated all entry conflicts in the TLB by using an infinite TLB, the savings in cache miss time would be small.

4.2.4 Summary

Table 9 consolidates the performance impact of the three major sets of OS misses that we identified, namely instruction misses, data misses due to process migration, and data misses in block operations. We now summarize the optimizations that we proposed. First, to reduce the number of instruction misses, we suggest examining associativity in the instruction cache or optimizing the layout of the OS code. Second, to eliminate migration-induced misses, we suggest exploring cache affinity scheduling. This technique optimizes the combined cache performance of application and OS. Finally, for misses in block operations, we suggest supporting data prefetching and/or selective cache bypassing in machines with a high cache miss penalty.

Table 9: Components of the stall time directly caused by OS misses.

Workload	OS Miss Stall Time / Non-Idle Execution Time (%)				
	Total OS Misses	Instr. Misses	Migration D-Misses	Block Oper. D-Misses	Rest of OS Misses
Pmake	21.0	10.9	1.0	6.2	2.9
Multpgm	21.5	9.2	4.2	4.7	3.4
Oracle	16.6	10.6	2.6	0.6	2.8
AVERAGE	19.7	10.2	2.6	3.8	3.0

4.3 Application Misses Caused by Operating System Interference

To finish this section on the cache performance of the OS, we now consider the misses that the OS induces in the application by displacing its state from the caches. These application misses we call *Ap_dispos* misses. Their contribution to the overall number of application misses is shown in Figure 10. In the figure, the total number of application misses in the workload is normalized to 100 and divided into data misses, labeled with *D*, and instruction misses, labeled with *I*. Overall, *Ap_dispos* misses account for 22-27% of all application misses.

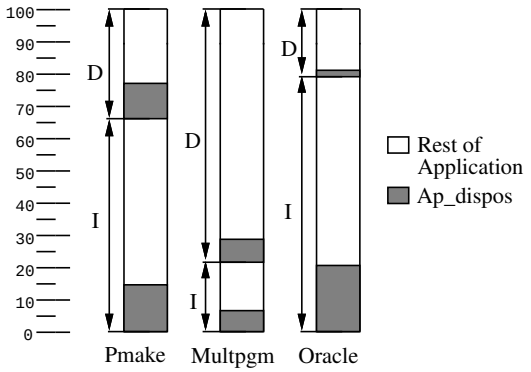

Figure 10: Fraction of application misses induced by OS interference (*Ap_dispos*). The total number of application misses is normalized to 100 and divided into data misses, labeled with D, and instruction misses, labeled with I.

5 Characterization of the Synchronization Performance of the Operating System

After analyzing the cache performance of the OS, we now consider its synchronization performance. We first present the CPU stall time caused by OS synchronization accesses and then analyze the patterns of these accesses.

5.1 CPU Stall Time Due to Operating System Synchronization

Since our hardware monitor does not capture the activity of the synchronization bus, the CPU stall times presented so far do not include the contribution of OS synchronization accesses. Using the technique detailed in Section 2.2, however, we measure that the stall time caused by OS synchronization accesses is 4.2-4.7% of the CPU time (column 'Current Machine' of Table 10). This overhead is largely the result of the protocol used in the synchronization bus, which suffers from the processor's lack of support for an atomic read-modify-write operation.

Later in this section, we show that OS locks exhibit good locality and low contention. These two properties imply that locks should perform well in a cache-based invalidation protocol like the one used for regular data in the machine. To illustrate this, we use traces of the lock accesses to simulate a machine where synchronization accesses use the main bus and the same cache coherence protocol as regular accesses. In the simulation, we assume support for atomically reading-modifying-writing lock variables using the load-linked and store-conditional instructions of the MIPS R4000 processor [15]. The resulting stall time due to OS synchronization misses is now only 0.7-1.0% of the CPU time (last column of Table 10). This indicates that, with efficient synchronization support, the stall time caused by OS synchronization accesses can be negligible.

Table 10: Stall time caused by OS synchronization accesses. The last column corresponds to a simulated scenario where the processor supports atomic read-modify-write (RMW) accesses to locks.

Workload	Stall Time Due to OS Synch. Accesses / Non-Idle Execution Time (%)	
	Current Machine	Atomic RMW Main Bus + Caches
Pmake	4.2	0.7
Multpgm	4.6	0.8
Oracle	4.7	1.1

5.2 Synchronization Access Patterns

A more detailed analysis of our measurements reveals characteristic patterns of access to OS locks. Due to lack of space, we do not present data for all of the workloads here. However, we show in [17] that they all exhibit similar behavior.

Overall, the OS synchronizes frequently. For example, the routines that acquire and release locks and semaphores in the OS are usually executed 3-5 times more frequently than the most popular non-synchronizing routines in the OS. In addition, we note that a large fraction of these acquire and release operations are directed to a few locks. Table 11 explains the function of the most important of these locks. To get an idea of the frequency of access to these locks, the second column of Table 12 shows the average number of cycles between two consecutive successful acquires for the most popular locks in *Pmake*. From the table, we observe that these locks are acquired once every 9000-36000 cycles. These cycles include CPU idle time.

While OS locks are frequently accessed, they show low contention. We expected low lock contention because, at the most, only four OS threads are active and therefore can pursue the same lock in our four CPU machine. To see the degree of lock contention, column 3 of Table 12 presents the fraction of attempts to acquire a lock that find it taken. To generate this data, of course, we ignore any spinning on the lock. Of all the locks studied, we see that the only one with significant contention with four processors is *Runqlk*. In addition to exhibiting low contention, these locks are not kept locked long enough to build up queues before their release. This is shown in column 4 of Table 12. This column presents the number of waiters when the lock is released if there is at least one waiter. From the table, we note that this parameter is usually very close to one.

Table 11: Functions performed by the most frequently-acquired locks. The postfix *_x* means that the lock is logically part of an array of locks where each one protects a similar data structure.

Lock	What the Lock Protects
Memlock	Data struct. that allocate/deallocate physical memory.
Runqlk	Scheduler's run queue.
Ifree	List of free inodes.
Dfbmaplk	Table of free blocks on the disk.
Bfreelock	List of free buffers for the buffer cache.
Calock	Table of outstanding actions like alarms or timeouts.
Shr_x	Per-process page tables and related structures.
Streams_x	Management of a character-oriented device.
Ino_x	Operations on a given inode, like read or write.
Semlock	Array of semaphores for the programmer to use.

Table 12: Characteristics of the most frequently acquired locks in *Pmake*.

Lock	# of Cycles between Acqu's (Thous.)	% of Failed Acqu's	# of Waiters if Any	% Two Acqu's by Same CPU w/o Interv.	Misses Cached / Misses Uncached (%)
Memlock	9.5	2.2	1.02	79.9	12
Runqlk	16.5	13.7	1.29	36.9	43
Ifree	16.7	0.8	1.00	91.4	5
Dfbmaplk	19.4	0.0	1.00	99.0	0
Bfreelock	22.5	1.5	1.00	72.6	15
Calock	35.1	0.3	1.00	11.4	45

Despite overall low lock contention with four CPUs, however, we observe a steady increase in lock contention as the number of CPUs increases. This effect is particularly obvious in *Runqlk*. For example, Figure 11 shows the number of failed acquires per millisecond for the locks with the highest contention in *Multpgm* as the number of CPUs increases. As before, we do not include the spins on a taken lock. This plot suggests that contention for *Runqlk* will be significant for machines with more CPUs.

Finally, OS lock accesses have high locality. As an example, the second to last column of Table 12 shows the fraction of successful lock acquires where the CPU that acquires the lock is the same as the one that last acquired it and no other CPU tried to access the lock in between. Except for *Calock* and, to a lesser extent, *Runqlk*, the numbers in the table are high, usually over 75%. This overall high locality makes caching locks advantageous. By caching we mean a cache coherence protocol where a CPU does not need an off-cache access when acquiring a lock that has not been accessed by anyone since this same CPU released it. For example, the load-linked and store-conditional MIPS R4000 instructions provide support for this protocol to work. The gains of caching are seen

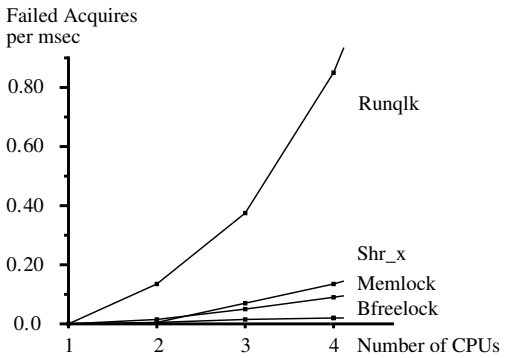

Figure 11: Lock contention as a function of the number of CPUs. This figure plots the number of failed acquires per millisecond for the locks with the highest contention in *Multpgm*. The Y-axis includes idle time.

in the last column of Table 12, which shows the ratio between the number of bus accesses in a machine that caches the locks and a machine that does not.

To conclude, the negligible stall time and low contention properties of OS locks imply that, if OS locks are cachable, then OS synchronization has a low performance cost in these small multiprocessors.

6 Implications for Larger Machines

So far we have focused on the performance of the OS on a small multiprocessor. In this section, we discuss the implications of our results for large shared-memory machines organized in clusters such as DASH [10], Paradigm [5], or Encore's Gigamax [21].

First, it may be appropriate to replicate the OS executable across clusters in these machines. This optimization is suggested by the numerous instruction misses in the OS. By storing a copy of the OS image in each cluster, instruction misses are serviced locally and therefore cache miss penalties are low. In addition, we avoid having a hot spot in the memory module where the OS code resides. Of course, this approach implies that a substantial chunk of memory is made unavailable for general use.

Second, the run queue should be distributed across clusters. As the number of CPUs in the machine increases, the potential for process migration increases too. One way to mitigate the effects of process migration is to distribute the run queue across clusters. Processes can then be encouraged to remain in the same run queue and therefore run mostly on the CPUs of one cluster. As a result, process migration will be less frequent and will cause mostly intracluster misses.

In general, shared data structures should be distributed across clusters and sharing limited to the CPUs within the cluster as much as possible. Except for the structures that store per-process private state, most of the structures in Figure 8 can be investigated for distribution.

Third, block transfers across clusters should be supported efficiently. For these machines, the frequent block operations studied in this paper are costly if performed across clusters. Therefore, memory should be allocated so that these operations access pages in the local cluster only. However, since this is not always possible, these machines benefit from support for efficient inter-cluster block transfers.

Finally, contention for the active locks will almost certainly increase with the number of CPUs. Table 11 showed the most active locks in the small machine characterized. To minimize the performance degradation, these locks should be distributed across clusters or new synchronization algorithms designed. The distribution does not apply, of course, to the locks whose name finishes with *_x*, for these belong to individual structures.

7 Summary

Understanding the cache performance of shared-memory multiprocessors is fundamental to continued success in speeding up these machines. One aspect of the cache performance of these machines that was not well understood is the cache performance of the OS. In this study, we use a hardware monitor to intercept the activity of the OS and are therefore able to characterize the cache performance of the OS.

The data shows that OS misses can slow down software-development and commercial workloads by 17-21%. The analysis of the data reveals three major sources of these misses, namely instructions fetches, process migration, and data accesses in block operations. Among these sources, the role of instruction fetches is larger than previously suspected. In addition, we also show that OS synchronization costs little if locks are cachable. In our analysis, we identify the most frequently acquired OS locks and their high locality and low contention properties.

One experience drawn from this study of the OS cache performance is that there is no single dominant issue that overwhelms the rest. While this is positive in that it suggests OS maturity, it also implies that there is no simple fix that will boost performance significantly.

Acknowledgments

This research is funded by DARPA contract N00039-91-C-0138. Josep Torrellas is supported by *La Caixa d'Estalvis per a la Vellesa i de Pensions* and the *Ministerio de Educacion y Ciencia*, both of Spain. Anoop Gupta is also supported by an NSF Presidential Young Investigator award. We gratefully acknowledge the help and encouragement provided by Luis Stevens and Ruby Lee. We also thank Daniel Lenoski, James Laudon, David Nakahira, Truman Joe, David Ofelt, and Jeff Kuskin for their help with the performance monitoring hardware.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[3] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.

[4] F. Baskett, T. Jermoluk, and D. Solomon. The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 Lighted Polygons per Second. In *Proceedings of the 33rd IEEE Computer Society International Conference - COMPCON 88*, pages 468–471, February 1988.

[5] D. Cheriton, H. Goosen, and P. Boyle. Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture. *IEEE Computer*, pages 33–46, February 1991.

[6] D. Cheriton, A. Gupta, P. Boyle, and H. Goosen. The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 410–421, May 1988.

[7] D. Clark. Cache Performance in the VAX-11/780. *ACM Transactions on Computer Systems*, 1(1):24–37, February 1983.

[8] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

[9] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, May 1991.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[11] J. D. McDonald and D. Baganoff. Vectorization of a Particle Simulation Method for Hypersonic Rarified Flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[12] S. McFarling. Program Optimization for Instruction Caches. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[13] J. Mogul and A. Borg. The Effect of Context Switches on Cache Performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, April 1991.

[14] J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[15] P. Ries. MIPS R4000 Caches and Coherence. In *Hot Chips III Symposium Record*, pages 6.1–6.5, August 1991.

[16] M. Squillante and E. Lazowska. Using Processor-Cache Affinity in Shared-Memory Multiprocessor Scheduling. Technical Report 89-060-01, Department of Computer Science, Univerity of Washington, June 1989.

[17] J. Torrellas. Multiprocessor Cache Memory Performance: Characterization and Optimization. Ph.D. dissertation, Stanford University, to appear, 1992.

[18] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Cache Performance and Synchronization Behavior of a Multiprocessor Operating System. Technical Report CSL-TR-92-512, Stanford University, January 1992.

[19] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Benefits of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. Technical report, Stanford University, July 1992.

[20] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 26–40, October 1991.

[21] A. W. Wilson. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 244–252, June 1987.