Comprehensive Hardware and Software Support for Operating Systems to Exploit MP Memory Hierarchies

Chun Xia and Josep Torrellas, *Member*, *IEEE*

Abstract—High-performance multiprocessor workstations are becoming increasingly popular. Since many of the workloads running on these machines are operating-system intensive, we are interested in exploring the types of support for the operating system that the memory hierarchy of these machines should provide. In this paper, we evaluate a comprehensive set of hardware and software supports that minimize the performance losses for the operating system in a sophisticated cache hierarchy. These supports, selected from recent papers, are code layout optimization, guarded sequential instruction prefetching, instruction stream buffers, support for block operations, support for coherence activity, and software data prefetching. We evaluate these supports under a simulated environment. We show that they have a largely complementary impact and that, when combined, speed up the operating system by an average of 40 percent. Finally, a cost-performance comparison of these schemes suggests that the most cost-effective ones are code layout optimization and block operation support, while the least cost-effective one is software data prefetching.

Index Terms—Cache hierarchies, shared-memory multiprocessors, architectural support for operating system, prefetching, trace-driven simulations, performance, block operations.

1 Introduction

High-performance multiprocessor workstations are becoming increasingly popular. These machines have very fast processors that put heavy stress on the memory subsystem. As a result, to deliver high performance, these machines are equipped with sophisticated cache hierarchies that intercept most processor requests.

These multiprocessors often run workloads that involve a considerable use of the operating system. Examples of such workloads are large multiprogrammed mixes or commercial applications such as databases and multimedia codes. Given the importance of good cache performance in these machines, it is therefore necessary to ensure that the operating system uses the cache hierarchy effectively. This gives rise to an interesting question: What hardware and software support for the operating system should be provided by the memory hierarchy of these machines?

Unfortunately, previous work in the literature does not target this question directly or completely. A large group of researchers have examined the cache performance of the operating system without focusing much on evaluating optimizations [1], [2], [6], [7], [11], [12], [16], [17], [19]. There is some work specifically focused on evaluating code optimizations for the operating system (for example, [18], [20], [21]). However, it examines only part of the problem, for example, minimizing instruction cache misses with code layout optimization. The combined effect of all the optimizations proposed is unknown, especially for very advanced multiprocessor memory hierarchies. Finally, there is other work proposing optimizations to enhance the cache performance of applications (for example, [9], [10], [19]), but it is not clear how to apply the results to the operating system itself.

The goal of this paper is to perform a simulation-based, cost-effectiveness evaluation of a comprehensive set of hardware and software supports for the operating system to exploit sophisticated multiprocessor memory hierarchies. We start by classifying the operating system cache misses. Then, we select from recent papers a comprehensive set of hardware and software supports to minimize these misses. These supports are code layout optimization, guarded sequential instruction prefetching, instruction stream buffers, support for block operations, support for coherence activity, and software data prefetching. We estimate the relative cost-effectiveness of these supports under a uniform environment.

Our results show that these schemes tend to complement each other and that, when combined, speed up the operating system by an average of 40 percent. The most cost-effective schemes are code layout optimization and support for block operations, while the least cost-effective one is software data prefetching.

The rest of this paper is organized as follows: Section 2 discusses the simulation-based experimental setup used; Section 3 gives insight into how an operating system uses the memory hierarchy; Section 4 discusses and evaluates the different optimization schemes considered; and Section 5 compares the relative cost-effectiveness of the schemes.

- *C. Xia is with BrightInfo. E-mail: chun.xia@technologist.com.*
- *J. Torellas is with the Computer Science Department, University of Illinois, Urbana-Champaign, IL 61801. E-mail: torrella@cs.uiuc.edu.*

Manuscript received 2 Aug. 1996.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 102069.

0018-9340/99/$10.00 © 1999 IEEE

2 Experimental Setup

This work is based on a simulation analysis of complete address traces from a multiprocessor machine. In this section, we briefly discuss the hardware and software setup that we use, the workloads that we trace, and the architecture that we simulate.

2.1 Hardware and Software Setup

We gather the traces from a 4-processor bus-based Alliant FX/8 multiprocessor. The operating system running on the machine is Alliant's Concentrix 3.0. Concentrix is multithreaded, symmetric, and based on Unix BSD 4.2. We use a hardware performance monitor that gathers uninterrupted traces of application and operating system in real time [3]. The performance monitor has one probe connected to each of the four processors. The probes collect all the references issued by the processors except those that hit in the per-processor 16-Kbyte primary instruction caches. Each probe has a trace buffer that stores over one million references. When one of the four trace buffers nears filling, it sends a nonmaskable interrupt to all processors. Upon receiving the interrupt, all processors halt in less than 10 machine instructions. Then, a workstation connected to the performance monitor reads the trace and processes it on-the-fly. Once the buffers have been emptied, processors are restarted via another hardware interrupt. Using this approach, we can trace an unbounded continuous stretch of the workload.

To perform our analysis, we need to collect the addresses of all instruction and data references issued by the processors. However, the performance monitor cannot capture instruction accesses that hit in the primary cache. To get around this problem, we annotate every single basic block in the operating system and application codes. Specifically, we add to each basic block a single machine instruction that causes a data read from a unique address [8]. When the basic block is executed, the read is issued. The performance monitor captures these accesses and can then interpret their addresses according to an agreed-upon protocol. With this instrumentation, we can completely reconstruct the path that each process followed in the execution. More details can be found in [8].

This instrumentation increases the dynamic size of the code by about 30 percent on average. Consequently, when we perform the trace simulations, we "compact" the instruction addresses to their original value as if the extra load instructions were not there. This eliminates any address distortion. In addition, we also need to ensure that we are not perturbing the execution of the operating system noticeably with the bigger code size. In particular, we need to ensure that there is no significant increase in page faulting activity. For this reason, we compare the relative frequency of execution of the operating system routines with and without this instrumentation. We find that there is no noticeable difference. More details can be found in [8].

2.2 Workloads

In our experiments, we perform four tracing sessions, each with a different load in the system. Each day-long session corresponds to about 15 seconds of real time. The loads are as follows:

TRFD_4 is a mix of four runs of a hand-parallelized version of the *TRFD* Perfect Club code [5]. Each program runs with four processes. The code is composed of matrix multiplies and data interchanges. It is highly parallel yet synchronization intensive. The most important operating system activities present are page fault handling, process scheduling, cross-processor interrupts, processor synchronization, and other multiprocessor management functions.

TRFD+Make is a mix of one copy of the hand-parallelized *TRFD* and a set of runs of the second phase of the C compiler, which generates assembly code given the preprocessed C code. We run four compilations, each on a directory of 22 C files. The file size is about 60 C lines on average. This workload has a mix of parallel and serial applications that force frequent changes of regime in the machine and frequent cross-processor interrupts. There is also substantial memory management activity.

ARC2D+Fsck is a mix of four copies of *ARC2D* and one copy of *Fsck*. *ARC2D* is another hand-parallelized Perfect Club code. It is a 2D fluid dynamics code that runs with four processes. It causes operating system activity like the one caused by *TRFD*. *Fsck* is a file system consistency check and repair utility. We run it on one whole file system. It contains a wider variety of operating system I/O code than *Make*.

Shell is a shell script containing a number of popular shell commands including *find, ls, finger, time, who, rsh*, and *cp*. The shell script creates a heavy multiprogrammed load by placing 21 programs at a time in the background. This workload executes a variety of system calls that involve context switching, scheduler activity, virtual memory management, process creation and termination, and I/O- and network-related activity.

The decomposition of the execution time of these workloads into user, idle, and operating system time is shown in the last few rows of Table 1. From the table, we can see that the operating system accounts for 42-55 percent of the total execution time.

2.3 Architecture Simulated

We simulate a multiprocessor workstation with four processors cycling at 500 MHz. Each processor has three on-chip caches: two 8-Kbyte direct-mapped primary caches (one for instructions and one for data) and a 128-Kbyte 4-way set-associative unified secondary cache. Each processor module also has a 2-Mbyte direct-mapped off-chip tertiary cache. The line sizes in all caches are 32 bytes. The primary data caches are write-through, while the secondary and tertiary caches are write back. There is one 8-entry deep, cache-line wide write buffer between the primary data cache and the secondary cache, between the secondary and the tertiary cache, and between the tertiary cache and memory. Read accesses bypass the writes stored in the write buffers. We use the Illinois cache coherence protocol under release consistency for coherence. The bus is 8-byte wide, cycles at 100 MHz, and has split transactions.

The memory hierarchy timings in processor cycles and without any resource contention are as follows: A hit in the primary cache is serviced in one cycle, while a hit in the

TABLE 1
Characteristics of the Workloads Studied

Characteristic		Workload				Average
		TRFD_4	*TRFD +Make*	*ARC2D +Fsck*	*Shell*	
Type of OS Misses / Total OS Misses (%)	M1.I	59.9	51.9	62.6	52.5	56.7
	M2.I	2.4	4.5	4.0	5.0	4.0
	M3.I	1.1	7.8	4.4	3.7	4.1
	(M1+M2+M3).I	63.4	64.2	71.0	61.2	64.9
	M1.D	26.5	25.3	21.1	31.1	26.0
	M2.D	3.0	2.6	1.7	2.4	2.4
	M3.D	7.1	7.9	6.2	5.3	6.7
	(M1+M2+M3).D	36.6	35.8	29.0	38.8	35.1
	(M2+M3).I + (M2+M3).D	13.6	22.8	16.3	16.4	17.2
	(M1+M2+M3).I + (M1+M2+M3).D	100.0	100.0	100.0	100.0	100.0
Overall	User Time (%)	49.7	36.7	42.9	24.9	38.5
	Idle Time (%)	8.1	7.8	11.9	29.2	14.3
	OS Time (%)	42.2	55.5	45.2	45.9	47.2
	Memory Stall on OS / OS Time (%)	85.9	88.2	87.8	86.6	87.1

secondary cache is serviced in six cycles. If the secondary cache misses, the request reaches the tertiary cache 10 cycles after the processor issued the request. Then, six cycles are needed to check the tag. If there is a hit, 12 cycles are needed to refill the secondary cache and a further six cycles are required to provide the requested word to the processor. Overall, therefore, the whole round trip access to the tertiary cache takes 34 cycles. Of course, the access is pipelined.

If, after checking the tertiary cache tags, we detect a miss, 12 cycles are needed to reach the memory. This time includes the bus arbitration and the actual request transfer in the bus. Then, the memory is accessed. We simplistically model the memory as a fixed, nonpipelined latency of 30 cycles or 60 ns. After that, the line transfer through the bus and into the tertiary cache takes 32 cycles. As indicated above, at this point we need 12 plus six additional cycles for the data to reach the processor. Overall, the complete pipelined round trip access to the memory takes 108 cycles. Resource contention can add extra cycles.

The architecture is simulated cycle by cycle. Both instruction and data accesses of both applications and operating system are modeled. Contention is fully modeled. This architecture, which we call *Base*, will be modified later.

The processor model that we use in our simulations is a simple one. We use direct execution of the applications and model a single-issue processor with blocking reads. Writes are nonblocking. All instructions take one cycle. We use this model because simulating a complex out-of-order superscalar processor for long programs takes a long time. Unfortunately, the use of such a simplistic processor model necessarily introduces errors in the results. Consequently, we correct the results according to the analyses by Pai et al. [13] and Ranganathan et al. [15].

Pai et al. [13] discuss how to correct the simulations of a simple processor like the one we use (*Simple*) to approximate the results of a simulation of a 4-issue, dynamically-scheduled superscalar with nonblocking reads, speculative execution, register renaming, and a 64-entry instruction window (*ILP*). The first issue involves the number of cache misses. They found that the *miss factor*, which is defined as the number of L1 misses in *Simple* over the number of L1 misses in *ILP*, is usually close to 1. Moreover, this is especially the case for applications that have a poor overlap among read misses. Symbolic codes like the operating system, where processors read linked lists and often use complex data structures with indirections, tend to fall under this category. Consequently, we will use the number of misses found with the single-issue processor as a good approximation of those with *ILP*.

The second issue is the execution time. Pai et al. [13] found that the best approximation to the execution time of *ILP* is the execution time of what they call *Simple.4xP.1cL1*. *Simple.4xP.1cL1* is a system that includes *Simple* processor running with a frequency four times higher and, that keeps the absolute latencies of the differnt levels of the cache hierarchy unchanged. The exception is the primary cache and write buffer access times, which are sped up 4 times. Consequently, the execution time data that we present corresponds to *Simple.4xP.1cL1*. In any case, according to [13], *Simple.4xP.1cL1* tends to be conservative in terms of the relative fraction of the execution time that the processor is stalled waiting for the memory system.

The final issue is the relative effect of prefetching in *Simple* and *ILP*. This issue is addressed by Ranganathan et al. [15]. They show that the absolute number of prefetches that are on time (completely hide the latency of a miss) or are a bit late (partially hide it) is approximately the same in *Simple* and *ILP*. Consequently, we will use the number of misses partially or totally hidden by prefetches in the single-issue processor as a good indication of the number of misses partially or totally hidden by prefetches in *ILP*. For the execution time, we will use the same correction as in the previous paragraph.

3 How the Operating System Uses the Memory Hierarchy

To propose effective support for the operating system, we start by examining how it uses the memory hierarchy.

3.1 How the Operating System Uses the Memory Hierarchy

To gain insight into how the operating system uses the memory hierarchy, Table 1 classifies the types of read misses in the operating system. The misses are classified according to the different levels of the memory hierarchy. The notation used is as follows: *M1* is the number of operating system reads that miss in the primary cache and hit in the secondary cache; *M2* is the operating system reads that miss in the secondary cache and hit in the tertiary cache; and *M3* is the operating system reads that miss in the tertiary cache. Consequently, the total number of operating system reads that miss in the primary cache is *M1* + *M2* + *M3*. In the table, we separate the instruction and the data components of these misses by adding an *I* or a *D* suffix to the notation. For example, *M2.I* is the instruction component of the *M2* misses.

The table also shows the fraction of the execution time spent in the user, operating system, and idle modes. Finally, the table shows the fraction of the operating system time that the processor is stalled due to accesses to the memory hierarchy.

The miss data suggests how the operating system uses the memory hierarchy. In primary caches, there are about twice as many instruction misses (*(M1 + M2 + M3).I* = 64.9 percent on average) as data misses (*(M1 + M2 + M3).D* = 35.1 percent on average). However, as we move to secondary and tertiary caches, data misses become more frequent than instruction misses. For example, *M3.I* is 4.1 percent on average, while *M3.D* is 6.7 percent on average. The reason for this behavior is that, while instructions have a smaller working set than data, they suffer severe conflicts in small direct-mapped primary caches. For the larger caches, instructions tend to fit in the cache. For example, for the 2-Mbyte tertiary cache, the roughly 1-Mbyte kernel code suffers mostly cold misses. Data, instead, does not suffer as much self-interference as the instructions in the primary cache. However, it suffers some coherence and even conflict misses in the large tertiary caches. The presence of coherence misses can be deduced from the relatively small *M2.D* compared to *M3.D*.

The table shows that these workloads spend a good fraction of their time inside the operating system. In addition, the table shows that a significant fraction of the operating system time is spent waiting for the memory hierarchy. The goal of Section 4 will be to reduce this stall time.

It is not possible to deduce from our data the exact composition of this operating system stall time. However, we can gain some insights if we assume for a moment that each type of operating system read miss in Table 1 contributes to the operating system stall time with an amount that is proportional to its no-contention latency time. In that case, we would see the importance of the stall time due to off-chip cache misses. This is not obvious from the number of misses: off-chip misses (*(M2 + M3).I* + *(M2 + M3).D*) are only 17.2 percent of the misses. However, since off-chip misses have long latencies, under our assumption, about three quarters of the operating system stall time would come from off-chip accesses. Overall, therefore, it is crucial to reduce the number of off-chip misses.

Furthermore, if we use the same assumption, the stall due to data misses is larger than the stall due to instruction misses. Again, this is due to the larger contribution of the off-chip stall time. For the data stream, the estimated off-chip stall time is much larger than the on-chip stall time. Even for the instruction stream, the estimated on-chip stall time accounts for only about one third of the total stall time. Overall, therefore, to reduce the processor stall time in the operating system due to memory accesses, we will largely focus on techniques that eliminate off-chip misses.

3.2 Why Do These Misses Occur?

To understand why the operating system exercises the memory hierarchy in this way, we examine address traces of the operating system. In the following, we discuss the instruction and data access patterns observed. A more detailed discussion can be found in [18], [20].

3.2.1 Instruction Access Patterns

It is well-known that the instruction access patterns in the operating system are different from those in typical scientific applications. In the operating system, tight loops account for a relatively small fraction of the execution time. A close examination of operating system instruction traces reveals that they often contain repeated sequences of hundreds of instructions, which we call *instruction sequences*. An instruction sequence may span several routines and is not part of any obvious loop. Instruction sequences are the result of operating system functions that entail a series of fairly deterministic operations with little loop activity. Examples of popular sequences are the first stages of handling a page fault, processing an interrupt, or servicing a system call, or the core routines involved in context switching.

A large fraction of the references and misses in the operating system occur in a set of frequently executed instruction sequences. These instruction sequences involve interrupt handling, page fault handling, system call handling, and context switching, among other things. As a result, although the size of the operating system code is roughly 1 Mbyte, a large fraction of it is rarely executed. For example, our workloads only exercise 18 percent of the code.

For small primary caches like the ones considered, over 90 percent of the operating system misses are due to self-interference. Much of this self-interference is caused by these frequently-executed instruction sequences. Furthermore, interference with the application accounts for relatively little. Consequently, we can quite effectively optimize the operating system code independently of the application. To understand how to do it, we examine three types of locality, namely spatial, temporal, and loop locality.

Spatial Locality. Instruction sequences should be an obvious means of exploiting spatial locality. Unfortunately, the basic blocks in an instruction sequence rarely are laid out in contiguous memory locations. Instead, they are

mixed up with many other seldom-executed basic blocks that are bypassed by conditional branches almost always taken. These seldom-executed basic blocks appear all over the code because the operating system has to handle all possible situations. As a result, the operating system code has low spatial locality.

Such code suffers more cold misses than necessary. More importantly, it is vulnerable to frequent conflict misses. For example, when two basic blocks in a popular instruction sequence conflict for the same cache location, we will have conflict misses on every invocation of the sequence. Our measurements indicate that, in the small direct-mapped primary caches considered, many of the misses are conflicts within instruction sequences. These conflict misses may also propagate to the secondary cache.

To eliminate these misses, we need to position the basic blocks of an instruction sequence in contiguous memory locations. This would seem to be difficult given that an instruction sequence often spans several routines. Fortunately, many of the operating system routines have very few calling points. Specifically, about 75 percent of the static instances of routines have only one calling point. Consequently, it is possible to bring caller and callee routines together in a manner similar to inlining.

Temporal Locality. Temporal locality can be exploited in two parts of the operating system code, namely small hot routines and popular instruction sequences. The former are a few routines that are invoked much more frequently than the rest. In addition, they tend to have a very small size, especially if we consider only their frequently executed sections. Examples of such routines are those that perform lock handling, timer counter management, state save and restore in context switches and exceptions, TLB entry invalidation, and block zeroing. Unfortunately, between two consecutive invocations of one of these popular routines, the operating system tends to execute much loopless code and, therefore, displaces the routine from the cache. To exploit temporal locality, we need to ensure that the most important basic blocks of these hot routines remain in the cache at all times.

Multiple popular instruction sequences can also provide a means of exploiting temporal locality. Indeed, sequences like the interrupt handler, page fault handler, and the context switching code have a high execution frequency and displace each other from the cache. To exploit temporal locality and, therefore, reduce the number of misses, these sequences should be placed in memory so that they do not overlap each other in the primary cache.

Loop Locality. Loop locality is the combined effect of spatial locality and temporal locality in loops. Loops fall into two categories, depending on whether or not they call routines. Loops that do not call routines have a small weight. Indeed, they account for only 29-39 percent of all dynamic instructions in the operating system compared to 96 percent in *ARC2D*. Furthermore, they often execute few iterations per invocation (six or fewer). Loops that call routines, on the other hand, execute complex operations, often spanning several routines. Consider, for instance, the loop that is executed when a process dies and the memory that it allocated has to be freed up. The operating system loops over all the page table entries of the process, performing many checks and complex operations in each case. These loops also have very few iterations per invocation. Their size, however, including the size of the routines that they call, is huge.

To exploit loop locality, we need to place each loop in memory so that no two instructions in the loop or in the routines that it calls conflict with each other in the cache. If this is possible, misses will be limited to the first iteration of the loop.

3.2.2 Data Access Patterns

Data access patterns are generally more complicated than instruction ones. In our operating system, there are about 1,500 global variables, of which about 55 percent are dynamically allocated. In addition, there is a large number of stack variables. Overall, there are many cold and conflict misses spread all over the kernel data. Since there are no obvious miss patterns, we classify the misses into those that occur while the operating system is executing block operations, those that are due to coherence activity, and those that are due to random conflicts.

Block Operations. Block operations include block copying and block zeroing and account for about 40 percent of the operating system read misses in the primary data cache. Block operations cause four overheads: stall due to read misses while reading the source block, stall due to write buffer overflow while writing the destination block, stall due to future misses on data that is displaced from the cache during the block operation, and, finally, the overhead of the instructions executed during the operation.

Block operations have several characteristics. First, large blocks, like those that use a whole 4-Kbyte page, tend to wipe large contiguous sections of the smaller caches and create many misses. In addition, block operations cause interference misses with the instructions in unified secondary and tertiary caches. Finally, large fractions of the blocks are often not reused after the block operation. For example, only a small fraction of the stack page tends to be accessed after it is copied in a process fork. Similarly, while the operating system initializes all the lines in a user page from the BSS section, the application program may not access most of the variables in the page for a long while or ever.

Coherence Activity. Coherence misses induced by data sharing account for about 10 percent of the operating system read misses in the primary data cache. In the operating system considered, there are several sources of coherence activity. The first one is synchronization variables. These include barriers, often used to synchronize the different processors before scheduling a parallel program, and locks, like those associated with accounting, physical memory allocation, job scheduling, and the high resolution timer. Other sources of coherence activity are counter variables that can be privatized, variables that suffer false sharing, and variables that are truly frequently shared. Examples of the latter are pointers in the system resource table, which point to the processes that use a given resource. All these sources of coherence misses need to be addressed.

Random Conflict Misses. Frequently accessed dynamic kernel data structures like the kernel stack or the *u* area suffer many misses. The *u* or *user* area is a region of the

TABLE 2
Components of the Processor Stall Time Targeted by Each Optimization

Optimization	Stall Due to M1 Misses			Stall Due to M2 and M3 Misses			Stall Due to Write Buffer Overflow
	Cold	Conflict	Coherence	Cold	Conflict	Coherence	
Lopt		X		X			
Gpref		X					
Strm				X	X		
Blk	X	X	X	X	X	X	X
Cohr			X			X	
Dpref		X					

Only the major effects are shown.

operating system that contains some important per-process state. These misses are caused by conflicts with other data structures. However, for a given data structure, there is no single other data structure that causes most of the conflicts. Instead, the conflicts are caused by many other data structures. As a result, data relocation is not a reasonable scheme; a more general scheme is necessary.

4 Hardware and Software Support for the Operating System

Based on all these insights and the body of published literature, we now evaluate comprehensive hardware and software supports for the operating system to exploit multiprocessor memory hierarchies better. We select a set of supports that have been proposed separately elsewhere [10], [18], [20], [21] and that target the problems identified. These supports are code layout optimization, guarded sequential instruction prefetching, instruction stream buffers, support for block operations, support for coherence activity, and software data prefetching. In this section, we evaluate and compare them under a uniform environment, combining them in a single system. In the next section, we examine trade-offs among them.

4.1 Code Layout Optimization by the Compiler

The first support that we consider is optimizing the layout of the code in memory to expose spatial and temporal locality [14], [18]. We do not attempt to expose loop locality because it is too difficult to exploit effectively. The main goal of this optimization, which we call *Lopt*, is to reduce the stall due to conflicts in the primary instruction cache (Table 2). In addition, it will also eliminate some cold misses in the secondary and tertiary caches because, by packing data in cache lines better, cache lines are used more effectively. Secondary and tertiary caches are too large for the instructions to suffer many conflict misses.

To expose spatial and temporal locality, we profile the code to determine the frequency and sequence of basic block execution. Then, we identify *seeds*, which are basic blocks that start sets of instruction sequences. Seeds are usually operating system entry points. Starting from seeds, we then use a greedy algorithm to build the instruction sequences. Given a basic block, the algorithm follows the most frequently executed path out of it. This implies visiting the routine called by the basic block or, if there is no subroutine call, following the highest-probability path out of the basic block. We stop when all the successor basic blocks have already been visited, or they have an execution count smaller than a threshold (*ExecThresh*), or all the outgoing paths have less than a certain probability of being followed (*BranchThresh*). When we have to stop, we start again from the seed looking for the next acceptable basic block.

In the algorithm, we have a loop that repeatedly selects a pair of values for *ExecThresh* and *BranchThresh*, generates the several resulting instruction sequences for each of the seeds, and places them in contiguous memory locations so that spatial locality is exposed. In each iteration of this loop, we lower the values of *ExecThresh* and *BranchThresh*, therefore capturing more and more rarely executed segments of code for all the seeds. The overall result is that we place the code in memory in segments of decreasing frequency of execution. As a result, popular sequences are placed close to other equally popular ones and, therefore, cannot conflict with them in the cache. As indicated in Section 3.2.1, this effect exposes temporal locality.

To exploit temporal locality in small hot routines, we assign the most popular basic blocks of them to a small contiguous range of memory locations called the *SelfConfFree* area. The SelfConfFree area does not conflict in the cache with other important parts of the operating system code because we place only seldom-executed code in memory locations that can conflict with it in the primary cache.

The effect of this optimization is shown in Fig. 1. The figure shows, for each workload, the number of operating system read misses in the primary cache for different environments. At this point, we consider only the *Base* and *Lopt* bars. *Base* corresponds to the unoptimized machine and is normalized to 1, while *Lopt* is *Base* plus the code layout optimization. In the figure, each bar is broken down into misses in the primary caches that hit in the secondary cache (*M1*), misses in the secondary cache that hit in the tertiary cache (*M2*), and misses in the tertiary cache (*M3*). The figure shows that *Lopt* delivers a significant reduction of misses (7-33 percent). As expected from Table 2, the large majority of the misses eliminated belong to the *M1* category. The number of misses in the two largest caches changes little.

To gain insight into how these changes translate into execution time variations, we examine Fig. 2. The figure

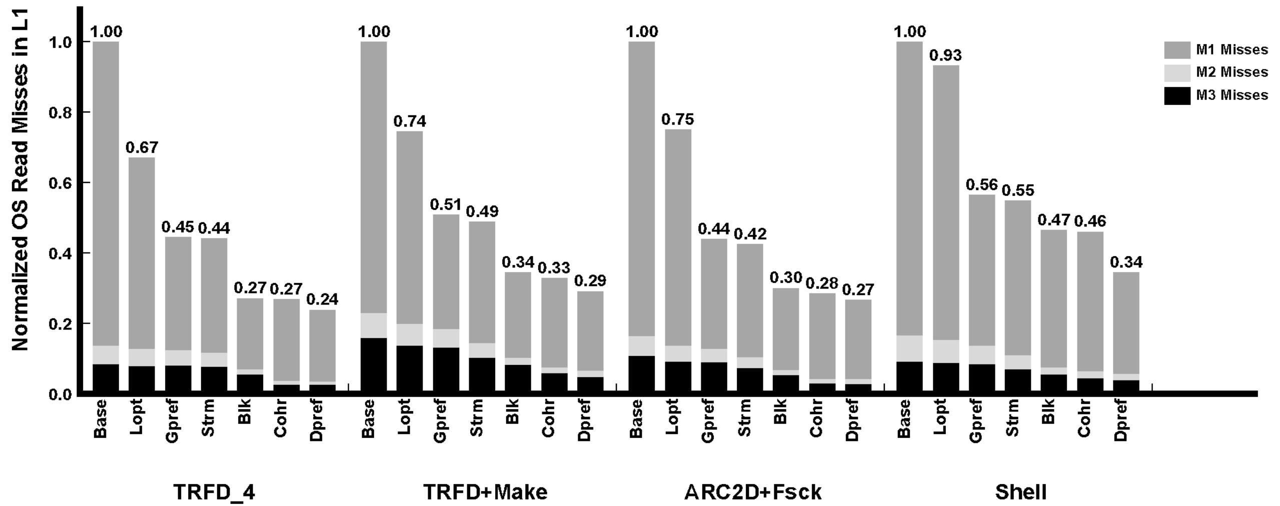

Fig. 1. Number of operating system read misses in the primary caches for several different environments. In each workload, the bars are normalized to *Base*. The impact of the optimizations is presented in a cumulative manner.

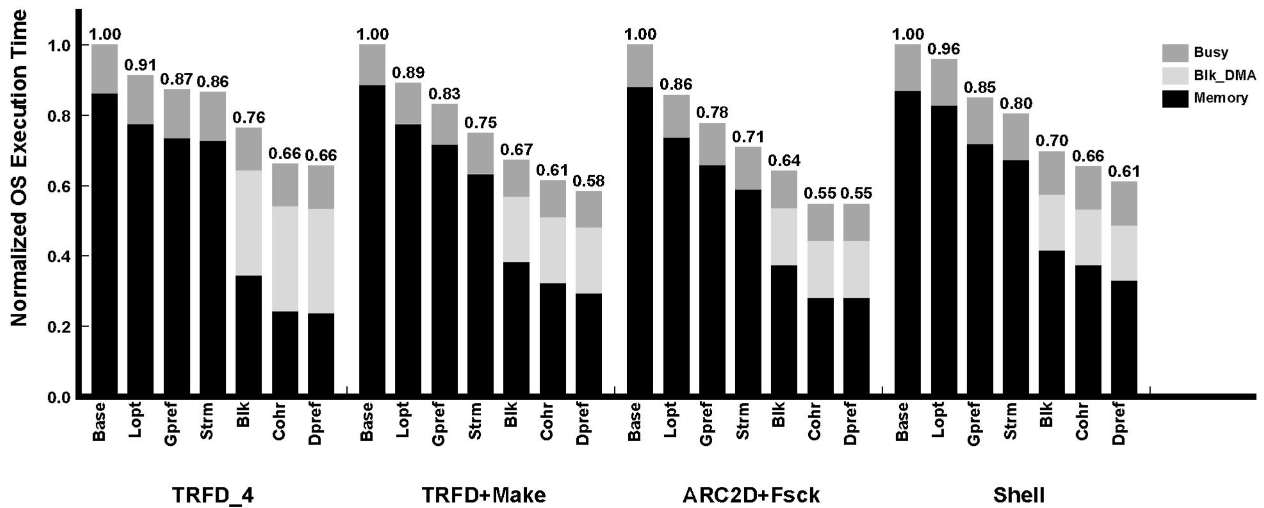

Fig. 2. Estimated execution time of the operating system for several different environments. In each workload, the bars are normalized to *Base*. The impact of the optimizations is presented in a cumulative manner.

shows, for each workload, the estimated execution time of the operating system for the same environments as in the previous figure. In each workload, the bars are normalized to *Base*. Each bar is broken down into the estimated busy time (*Busy*), the stall time while a block DMA occurs (*Blk_DMA*, to be described later), and the stall time due to memory accesses (*Memory*). *Busy* includes synchronization time.

From the figure, we can see that this optimization removes a good fraction of the *Memory* stall time. The resulting speedup, however, is not as large as the reduction in misses in Fig. 1 would initially suggest. The reason is that most of the eliminated misses belong to the relatively inexpensive class of primary cache misses satisfied by the secondary caches. Only a few of the more expensive *M2* and *M3* misses are eliminated. The results are consistent with what we expected in Table 2. Overall, the operating system runs about 10 percent faster on average.

Table 3 repeats the data in Figs. 1 and 2, breaking down the instruction and data miss contributions. For space reasons, the table shows only the average of the four workloads.

4.2 Hardware Instruction Prefetching

Table 3 shows that after *Lopt* is applied, *M1.I*, *M2.I*, and *M3.I* are still large. The on-chip cache misses that remain tend to be spread out along the code in a somewhat uniform manner and are hard to eliminate. Similarly, some off-chip instruction misses remain and are likely to cause a significant amount of stall time. Therefore, we now try to hide the latency of these two types of misses with an on-chip and an off-chip prefetching scheme.

4.2.1 Guarded Sequential Instruction Prefetching

Guarded sequential instruction prefetching [21] is an on-chip hardware prefetching scheme designed to be combined with the code layout optimization described above. In the previous section, we saw that the code is organized in instruction sequences and then laid out in memory. Once a processor starts executing an instruction sequence, it is very

TABLE 3
Data in Figs. 1 and 2, Breaking Down the Instruction and Data Cache Miss Contributions

Parameter	Optimization						
	Base	*Lopt*	*Gpref*	*Strm*	*Blk*	*Cohr*	*Dpref*
M1.I	56.7	36.4	8.6	9.7	10.0	10.0	10.0
M1.D	26.0	26.2	26.1	26.0	16.8	18.1	13.7
M2.I	4.0	3.1	5.2	1.4	0.8	0.8	0.8
M2.D	2.5	2.4	2.5	2.5	1.1	0.8	0.7
M3.I	4.2	3.3	2.9	1.4	1.3	1.3	1.3
M3.D	6.7	6.5	6.5	6.5	4.6	2.5	2.0
OS Misses	100.0	77.4	48.9	47.6	34.5	33.5	28.5
Memory	87.1	77.5	70.4	65.2	37.9	30.4	28.5
Blk_DMA	0.0	0.0	0.0	0.0	19.9	19.9	19.9
Busy	12.9	12.9	12.9	12.9	11.5	11.5	11.5
OS Time	100.0	90.4	83.3	78.1	69.3	61.8	59.9

Only the average of the four workloads is shown. The impact of the optimizations is presented in a cumulative manner.

likely that it will continue executing it until the end of the sequence. At that point, the processor is likely to branch off. Consequently, we can encode one bit in the last instruction of each sequence and set the bit to 1. This bit is called the *Guard Bit*. When a processor starts executing an instruction sequence, the prefetcher will race ahead prefetching in sequence all the memory lines until it stops when it finds a guard bit set. We call the scheme *Gpref*.

This scheme is better than ordinary sequential prefetching schemes that prefetch a fixed number of lines on reference, or on miss. Indeed, such schemes may prefetch past the end of sequence and, therefore, pollute the primary cache. Alternatively, they may stop prefetching before the end of sequence and, therefore, not hide as many misses. In addition, guarded sequential prefetching is reasonably cheap to implement. We need to make three modifications. First, we need to encode the guard bit in control transfer instructions, which are the ones that can be the end of sequences. Second, we need to make minor extensions to existing next-line sequential prefetchers. Finally, we need a bit in each TLB entry to turn the prefetcher on and off at run time based on whether or not the page contains optimized code.

As Table 2 shows, *Gpref* tries to eliminate stall due to the remaining conflict-induced *M1.I* misses. We expect little impact on off-chip stall. To see how successful this scheme is, we first look at the estimated miss reduction (Fig. 1). The columns labeled *Gpref* correspond to the *Lopt* environment plus guarded sequential prefetching. From the figure, we see that the scheme hides to some degree about 30-40 percent of the remaining misses. This impact is largely due to hiding *M1* misses; it is too hard to hide the more expensive off-chip misses (Table 3). The estimated impact of the optimization on the execution time of the operating system is shown in Fig. 2. The figure shows that the scheme removes some more stall time, making the operating system run 7.8 percent faster on average. As we can see from Table 3, there are now few remaining *M1* instruction misses.

4.2.2 Instruction Stream Buffers

Table 3 shows that the application of *Gpref* has not had much impact on the off-chip instruction misses (*M2.I* and *M3.I*). These misses often occur in contiguous addresses and tend to be clustered. This is especially the case for cold misses. Consequently, we use instruction stream buffers between secondary caches and the memory bus to complement the guarded sequential prefetcher. Stream buffers prefetch in hardware with a fixed stride after they recognize such a stride in the off-chip accesses [10]. In our experiments, we use two off-chip 8-entry direct-access stream buffers per processor. The buffers are allowed to look up the tertiary cache, thus avoiding unnecessary bus accesses when the line is in the tertiary cache. We select two buffers to be able to intercept references to the caller and callee procedures in a procedure call. We call the scheme *Strm*.

As Table 2 shows, *Strm* targets the stall due to cold and conflict-induced *M2.I* and *M3.I* misses. The estimated impact of this scheme on the number of misses is shown in Fig. 1. The columns labeled *Strm* correspond to the *Gpref* environment plus the stream buffers. From the figure, we see that the scheme hides only 2-5 percent of the remaining misses. However, nearly all of the hidden misses are off-chip misses (*M2* and *M3*). As a result, the scheme is estimated to have a noticeable impact on the execution time. This is shown in the *Strm* bars of Fig. 2. The bars show that stream buffers reduce the execution time of the operating system by an average of 6.2 percent. This reduction is achieved by decreasing off-chip stall time. Overall, after this optimization, there are few instruction misses, as shown in column *Strm* of Table 3. In that column, *M2.I* + *M3.I* is 2.8 instead of 8.1 in *Gpref*. Experiments with deeper buffers or more buffers did not result in much further gain.

4.3 Off-Chip Hardware Support for Block Operations

After focusing on instruction misses, we now attempt to minimize the stall produced by data accesses. Since, according to Section 3.2.2, block operations account for about 40 percent of operating system misses in the primary data cache, we address this problem first. To optimize block operations, we use an off-chip block transfer engine [20]. This engine is a smart controller that performs block operations in a memory-to-memory DMA-like fashion while holding the bus for the duration of the block transfer. In the meantime, the processor that was supposed to perform the block operation waits. Caches are not polluted with either the source or the destination of the block operated upon. However, like in a cache-coherent DMA for I/O, the tertiary caches of all the processors are read or updated if they contain the source or destination cache line respectively. If a cache is updated, the update propagates all the way to the on-chip caches. At peak rate, the hardware transfers 8 bytes in the bus from source to destination memory location every two bus cycles. Of course, the bus transfer may have to slow down every time that caches have to be read or updated. This scheme is attractive because it does not require modifying the processor chip and delivers high performance [20]. We call the scheme *Blk*.

As Table 2 shows, *Blk* tries to eliminate all types of stall times. Indeed, block operations can create any type of read miss in any of the three caches, including cold, conflict and coherence misses. Our scheme eliminates cold and conflict misses by bypassing the caches and coherence misses by

updating the caches if they have a copy of the data. Furthermore, our scheme reduces the stall due to write buffer overflow by not using the write buffers: The data is transferred from a memory location to another memory location without involving the processor.

The impact of this scheme on the number of misses is shown in Fig. 1. The columns labeled *Blk* correspond to the *Strm* environment plus the support for block operations. From the figure, we see that the scheme eliminates 15-40 percent of the remaining misses. Furthermore, it reduces the three types of misses. As a result, the estimated impact on execution time is considerable. This is shown in the *Blk* bars of Fig. 2. The bars show that the scheme is estimated to reduce the execution time of the operating system by an average of 11.3 percent. This reduction is accomplished by decreasing the number of *M1*, *M2*, and *M3* data misses (Table 3), by lowering the *Busy* time thanks to not having to execute the instructions of the block operations, and by reducing stall due to write buffer overflow. Unfortunately, we now have some *Blk_DMA* stall time because the processor is stalled while the block transfer engine takes over (Fig. 2).

4.4 Support for Coherence Activity

While Section 3.2.2 indicated that coherence activity causes only about 10 percent of the operating system read misses in the primary data cache, these are important misses. Indeed, they miss in the three levels of caches and are, therefore, expensive. To remove most of these misses, we use a hybrid cache coherence protocol: update-based for a small set of heavily-shared variables and invalidate-based for the rest [20]. The set of variables under the update-based protocol includes barriers, active locks, and variables that often exhibit a producer-consumer behavior. All these variables together use about 380 bytes and are placed in a single page. For this page only, we use the Firefly [4] update protocol. Note that this optimization can be supported with off-the-shelf processors. For example, the MIPS R4000 processor supports update/invalidate protocol selection for each individual page. The selection is done with a bit in each TLB entry. We note, however, that the operating system designer needs to have sophisticated tools to identify the right variables to use updates on. Clearly, applying an update protocol to all operating system variables creates too much traffic.

Other optimizations included are the privatization of variables that can be privatized, the relocation of variables that exhibit false sharing into different cache lines, and the relocation of variables that are accessed in sequence into the same cache line. All these other optimizations require software-only support and currently rely on programmer input [20]. We call the combined scheme *Cohr*.

As Table 2 shows, the major effect of *Cohr* is to reduce stall time due to on-chip and off-chip coherence misses. The actual impact of the scheme on the number of misses is shown in Fig. 1. The columns labeled *Cohr* correspond to the *Blk* environment plus the coherence optimization. From the figure, we see that the scheme eliminates only 1-7 percent of the remaining misses. However, the large majority of the misses eliminated belong to the most expensive class, namely *M3*. This is because coherence misses miss in the three levels of caches. The effect seen in the figure on other types of misses is due to interference. As a result, the estimated impact on execution time is considerable, as shown in the *Cohr* bars of Fig. 2. The bars show that the coherence optimization reduces the execution time of the operating system by an average of 10.8 percent. Overall, after this optimization, there are few remaining off-chip data misses. This is shown in column *Cohr* of Table 3. In that column, *M2.D + M3.D* is 3.3 instead of 5.7 in *Blk*.

4.5 Software Data Prefetching

Most of the data misses remaining in the operating system are caused by data structures conflicting in the cache with usually many other data structures. Intuitively, we could use prefetching to hide these misses. However, given that the operating system generates irregular data reference patterns, hardware prefetching is unlikely to be successful. Instead, hand-inserted software prefetching seems to be the only alternative [20].

To determine where to insert data prefetches, we measure the number of data misses suffered by each basic block of the operating system code. For the few basic blocks with the highest number of misses, we determine the source code statements that cause the misses. These statements constitute miss hot spots. Miss hot spots tend to be similar in different workloads. For this experiment, we select the 12 most-active miss hot spots and apply prefetching by hand. These hot spots are five loops and seven instruction sequences. They account for 25-50 percent of the remaining operating system data misses in the primary cache. While inserting the prefetches requires intensive manual work, the importance of the operating system may make the effort worthwhile. We call the scheme *Dpref*.

Inserting prefetch statements early enough in the irregular operating system code is hard. As a result, we should expect to hide on-chip cache misses only. For this reason, Table 2 shows that *Dpref* targets on-chip misses only. To see the estimated impact on the number of misses, we examine Fig. 1. The columns labeled *Dpref* correspond to the *Cohr* environment plus data prefetching. From the figure, we see that the scheme hides 4-26 percent of the remaining misses. However, nearly all of these misses are *M1* misses. As a result, the estimated impact on execution time is small, as shown in the *Dpref* bars of Fig. 2. The bars show that data prefetching reduces the execution time of the operating system by an average of only 3.1 percent. Column *Dpref* in Table 3 shows that few misses now remain.

5 Trade-Off Analysis

After evaluating several hardware and software supports for the operating system, we now examine their relative cost-effectiveness. In our analysis, we start by examining the interference among the schemes and then assess the overall cost-effectiveness of the schemes.

5.1 Interference Among the Schemes

While instructions and data share secondary and tertiary caches, the optimizations that we perform on the instruction stream do not have much effect on the data accesses. This

TABLE 4
Cost-Effectiveness Comparison of the Different Hardware and Software Supports Proposed

Optimization	H/W Support?		S/W Support?		Performance Increase (%)	Design Effort	System Cost
	On-Chip	Off-Chip	Compiler	OS			
Lopt			X		9.6	Medium	≈ 0
Gpref	X		X	X	7.8	Low	≈ 0
Strm		X			6.2	Low	Medium
Blk		X			11.3	Medium	Medium
Cohr	X	X	X	X	10.8	High	Low
Dpref	X		X		3.1	High	≈ 0

can be seen from Table 3. Indeed, when the *Lopt*, *Gpref*, and *Strm* instruction optimizations are performed, the data misses in the caches change little: As shown in the table, *M1.D* + *M2.D* + *M3.D* change from 35.2 in *Base* to 35.0 in *Strm*.

Similarly, the optimizations on the data stream do not have much effect on the instruction accesses either. The only exception is the block operation optimization which, as shown in Table 3, decreases number of the *M2.I* instruction misses a bit. The reduction occurs because, by not caching the block of data, we displace fewer instructions. However, the coherence and data prefetching optimizations affect only very slightly the instruction misses. Overall, therefore, we conclude that the instruction and data optimizations do not interfere much in the caches.

Some optimizations need to be applied in groups to be more effective or cheaper. For the instruction optimizations, *Gpref* and *Strm* need to use a layout optimized by *Lopt*. Otherwise, fewer misses will be in sequence and, therefore, the prefetcher and the stream buffers will be less effective. Similarly, *Blk* should be applied before the other data optimizations. This is because, since *Blk* eliminates many misses, the analysis for *Cohr* and *Dpref* needs to consider fewer variables and, therefore, can be simpler.

Finally, since the machine is shared by applications and the operating system, we want to know the effect of each optimization on the user time. Our simulation results show that the user time only increases by about 2 percent on average after all the operating system optimizations presented are applied. Consequently, the user code is largely unaffected by these operating system supports.

5.2 Overall Cost-Effectiveness of the Optimizations

While all the optimizations analyzed speed up the operating system execution, they are not all equally cost-effective. To approximately assess the relative cost-effectiveness of these schemes, we compare them under three points of view, namely performance, design effort, and system cost. In this section, we keep the same order of optimizations as before and we measure the estimated performance impact of an optimization relative to a system with all the previous optimizations turned on. We do this because, as indicated in Section 5.1, the instruction optimizations are largely independent of the data optimizations and, in addition, we have ordered the optimizations in the order that maximizes the effectiveness and the simplicity of them. Design effort is estimated by the complexity of the system and the extent of manual involvement. System cost consists of two aspects. For on-chip modifications, the system cost is determined by the increase in die size. For off-chip modifications, the system cost is determined by the amount and complexity of the board logic required. Overall, the total cost is a combination of design effort and system cost.

Table 4 compares the estimated performance, design effort, and system cost of the schemes analyzed. In addition, for each scheme, the table indicates whether or not it requires on-chip or off-chip hardware support, compiler support, or operating system support. Such supports have been discussed in Section 4. In the table, the estimated performance is measured by how much the scheme reduces the execution time of the operating system in Fig. 2. Recall that we have shown that the instruction and the data optimizations are largely independent, that *Lopt* should be performed before the other instruction optimizations, and that *Blk* should be performed before the other data optimizations. As a result, we feel that there is no need to compute, for each optimization, the estimated change in performance relative to *Base*. Instead, we can use the data in Fig. 2. Overall, from the table, we see that the highest performing schemes are *Blk*, *Cohr*, and *Lopt*, while the least performing one is *Dpref*.

The design effort required in the different schemes varies a lot. The schemes with the highest design effort are *Dpref* and *Cohr*. Both schemes are likely to require extensive manual involvement to profile and modify the code. In *Dpref*, the work involves finding miss hot spots and then inserting prefetches, while, in *Cohr*, the work involves finding update patterns, false sharing, privatizable variables, and other sharing patterns and then relocating variables. *Lopt* and *Blk* require a medium design effort. In *Lopt*, the profiling and basic block motion is time consuming and may run into unexpected problems. However, the process can be automated to a large extent. *Blk* involves redesigning the off-chip memory hierarchy to add a DMA-like block transfer engine. The design is not too hard because a lot of the hardware and interface logic required is similar to a current DMA. However, we also need to make minor modifications to the operating system to invoke the new module in block operations. Finally, *Strm* and *Gpref* require a lower design effort. *Strm* requires modifying the off-chip memory hierarchy to include stream buffers. For *Gpref*, we assume that the processor already supports sequential prefetching. *Gpref* requires small changes in the control logic of the on-chip sequential prefetcher and TLB,

in the encoding of the branch instructions, and in localized points of the operating system.

The system cost of the different schemes is also quite different. *Blk* and *Strm* have a medium system cost because they need board logic to perform block operations or instruction streaming. *Cohr* has a lower system cost, although it involves supporting two cache coherence protocols. Finally, the rest of the schemes have even lower system costs. Indeed, *Lopt* is a software-only scheme. *Dpref* and *Gpref* require simple on-chip modifications in processors that already support prefetching. They do not increase the die size much.

Overall, we feel that all the schemes except *Dpref* are probably cost-effective. The most cost-effective ones are possibly *Lopt* and *Blk*.

6 Conclusions

High-performance multiprocessor workstations are becoming increasingly popular. Since many of the workloads that these machines will run are operating system intensive, there is a lot of interest in the type of support for the operating system that the memory hierarchy of these machines should provide. This paper has addressed this question.

We have evaluated a comprehensive set of hardware and software supports for the operating system to use the memory hierarchy more effectively: code layout optimization, guarded sequential instruction prefetching, instruction stream buffers, support for block operations, support for coherence activity, and software data prefetching. We evaluated the impact of these six schemes under a uniform environment. These schemes have a largely complementary impact and, when combined, speed up the operating system by about 40 percent on average. Finally, we have compared the cost-effectiveness of these schemes. The most cost-effective schemes are code layout optimization and block operation support, while the least cost-effective one is software data prefetching.

Acknowledgments

We thank the referees and the graduate students in the I-ACOMA group for their feedback. We also thank Tom Murphy, Perry Emrath, and Liuxi Yang for their help with the hardware and operating system, and Intel and IBM for their generous support. This work was supported in part by the U.S. National Science Foundation under grants NSF Young Investigator Award MIP 94-57436, RIA MIP 93-08098, MIP 93-07910, and MIP 89-20891, NASA Contract No. NAG-1-613, and gifts from Intel and IBM.

References

[1] A. Agarwal, J. Hennessy, and M. Horwitz, "Cache Performance of Operating System and Multiprogramming Workloads," *ACM Trans. Computer Systems,* vol. 6, no. 4, pp. 393-431, Nov. 1988.

[2] T. Anderson, H. Levy, B. Bershad, and E. Lazowska, "The Interaction of Architecture and Operating System Design," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 108-120, Apr. 1991.

[3] J.B. Andrews, "A Hardware Tracing Facility for a Multiprocessing Supercomputer," Technical Report 1009, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, May 1990.

[4] J. Archibald and J.L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Systems,* vol. 4, no. 4, pp. 273-298, Nov. 1986.

[5] M. Berry et al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *Int'l J. Supercomputer Applications,* vol. 3, no. 3, pp. 5-40, Fall 1989.

[6] J. Chapin, S.A. Herrod, M. Rosenblum, and A. Gupta, "Memory System Performance of UNIX on CC-NUMA Multiprocessors," *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems,* pp. 1-13, May 1995.

[7] J.B. Chen and B.N. Bershad, "The Impact of Operating System Structure on Memory System Performance," *Proc. 14th ACM Symp. Operating System Principles,* pp. 120-133, Dec. 1993.

[8] R. Daigle, C. Xia, and J. Torrellas, "Low Perturbation Address Trace Collection for Operating System, Multiprogrammed, and Parallel Workloads in Multiprocessors," technical report, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, Mar. 1996.

[9] W.W. Hwu and P.P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proc. 16th Ann. Int'l Symp. Computer Architecture,* pp. 242-251, June 1989.

[10] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* pp. 364-373, May 1990.

[11] A. Maynard, C. Donnelly, and B. Olszewski, "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 145-156, Oct. 1994.

[12] J. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware," *Proc. Summer 1990 USENIX Conf.,* pp. 247-256, June 1990.

[13] V. Pai, P. Ranganathan, and S. Adve, "The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology," *Proc. Third Int'l Symp. High-Performance Computer Architecture,* pp. 72-83, Feb. 1997.

[14] K. Pettis and R.C. Hansen, "Profile Guided Code Positioning," *Proc. SIGPLAN 1990 Conf. Programming Language Design and Implementation,* pp. 16-27, June 1990.

[15] P. Ranganathan, V. Pai, H. Abdel-Shafi, and S. Adve, "The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 144-156, June 1997.

[16] M. Rosenblum, E. Bugnion, S.A. Herrod, E. Witchel, and A. Gupta, "The Impact of Architectural Trends on Operating System Performance," *Proc. 15th ACM Symp. Operating System Principles,* Dec. 1995.

[17] J. Torrellas, A. Gupta, and J. Hennessy, "Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 162-174, Oct. 1992.

[18] J. Torrellas, C. Xia, and R. Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workloads," *Proc. First Int'l Symp. High-Performance Computer Architecture,* pp. 360-369, Jan. 1995.

[19] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer, "Instruction Fetching: Coping with Code Bloat," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 345-356, June 1995.

[20] C. Xia and J. Torrellas, "Improving the Data Cache Performance of Multiprocessor Operating Systems," *Proc. Second Int'l Symp. High-Performance Computer Architecture,* pp. 85-94, Feb. 1996.

[21] C. Xia and J. Torrellas, "Instruction Prefetching of Systems Codes with Layout Optimized for Reduced Cache Misses," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* pp. 271-282, May 1996.

Chun Xia received the BEng degree in electrical engineering and the MEng degree in computer science from Tsinghua University, Beijing, China, in 1985 and 1989, respectively. He received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1996. From 1985-1990, he was a reseracher at the Institute of Microelectronics of Tsinghua University, Beijing, China. From 1996-1998, he was with Sun Microsystems as a senior software engineer, working on a high-scalable and available Internet server cluster project and the “Fullmoon” Solaris clustering project. In 1998, he founded BrightInfo, an Internet commerce software company and has been its CTO since then. His research interests include highly scalable and available distributed systems on the Internet, Internet content communication infrastructure, and multiprocessor systems architecture and operating systems.

Josep Torrellas received a PhD in electrical engineering from Stanford University in 1992. He is an associate professor in the Computer Science Department at the University of Illinois at Urbana-Champaign, with a joint appointment with the Electrical and Computer Engineering Department. He is also vice-chairman of the IEEE Technical Committee on Computer Architecture (TCCA). In 1998, he was on sabbatical at the IBM T.J. Watson Research Center researching the next generation processors and scalable computer architectures. He received the U.S. National Science Foundation (NSF) Research Initiation Award in 1883, the NSF Young Investigator Award in 1994, and, from the University of Illinois, the C.W. Gear Junior Faculty Award and Xerox Award for Faculty Research in 1997. Dr. Torrellas' primary research interests are new processor, memory, and software technologies and organizations to build uni and multiprocessor computer architectures. He is the author of more than 60 refereed papers in major journals and conference proceedings. He has been on the organizing committees of many international conferences and workshops. Recently, he co-organized the First and Second Workshops on Computer Architecture Evaluation Using Commercial Workloads and the Seventh and Eighth Workshops on Scalable Shared Memory Multiprocessors. He is a member of the IEEE.