Improving the Data Cache Performance of Multiprocessor Operating Systems[1]

Chun Xia and **Josep Torrellas**

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801
xia,torrella@csrd.uiuc.edu

Abstract

Bus-based shared-memory multiprocessors with coherent caches have recently become very popular. To achieve high performance, these systems rely on increasingly sophisticated cache hierarchies. However, while these machines often run loads with substantial operating system activity, performance measurements have consistently indicated that the operating system uses the data cache hierarchy poorly. In this paper, we address the issue of how to eliminate most of the data cache misses in a multiprocessor operating system while still using off-the-shelf processors.

We use a performance monitor to examine traces of a 4-processor machine running four system-intensive loads under UNIX. Based on our observations, we propose hardware and software support that targets block operations, coherence activity, and cache conflicts. For block operations, simple cache bypassing or prefetching schemes are undesirable. Instead, it is best to use a DMA-like scheme that pipelines the data transfer in the bus without involving the processor. Coherence misses are handled with data privatization and relocation, and the use of updates for a small core of shared variables. Finally, the remaining miss hot spots are handled with data prefetching. Overall, our simulations show that all these optimizations combined eliminate or hide 75% of the operating system data misses in 32-Kbyte primary caches. Furthermore, they speed up the operating system by 19%.

1 Introduction

Bus-based shared-memory multiprocessors with coherent caches have become very popular in recent years. These systems are equipped with fast processors, which stress the memory subsystem heavily. To achieve high performance, therefore, these machines are designed with sophisticated cache hierarchies that intercept processor requests effectively.

It is common for these machines to run multiprogrammed loads. These loads often include compilations, parallel or sequential compute-intensive applications, databases, or I/O and system call intensive applications. Often, these loads involve considerable use of the operating system. Given the importance of good cache performance in these machines, it is therefore necessary to ensure that the operating system uses the cache hierarchy efficiently.

Unfortunately, work by many researchers has consistently indicated that the operating system does not use the cache hierarchy very efficiently. While this is true for both instruction and data caches, in this paper we focus on the data caches exclusively. For example, Agarwal *et al* [1] pointed out the many cache misses caused by the operating system. Similarly, Maynard *et al* [15] found out that kernel data miss rates are usually higher than application data miss rates. Ousterhout [17] and Anderson *et al* [2] emphasized the many costly activities performed by the operating system like block copying. Chen and Bershad [9] reported that memory system accesses by the operating system have a high cost and that block operations are particularly costly. For a bus-based multiprocessor running system-intensive loads, Torrellas *et al* [19] reported that the operating system is responsible for a large fraction of the data cache misses. In addition, they showed that the dominant sources of data misses in the operating system are coherence activity and block operations. Similarly, Chapin *et al* [8] have recently reported similar findings for a NUMA multiprocessor running UNIX.

While all this past work has successfully characterized the problem, very little work has been done toward eliminating it [8, 19]. In this paper, we focus exclusively on eliminating most of the data misses in a multiprocessor operating system. Any changes that we propose, however, should be compatible with the use of off-the-shelf processors.

We use a performance monitor to examine traces of a 4-processor shared-memory machine running compilations, mixes of parallel and sequential applications, and system call intensive loads under UNIX. Based on our observations, we propose hardware and software support to eliminate misses and evaluate it via detailed simulations. The support targets block operations, coherence activity, and cache conflicts. For block operations, simple cache bypassing or prefetching schemes are undesirable. Instead, it is best to use DMA-like schemes that pipeline the data transfer in the bus without sending the data up to the processor. For coherence misses, we suggest data privatization and relocation, and the use of updates for a small core of shared variables. Finally, for the remaining miss hot spots, we suggest data prefetching. Overall, our simulations show that all these optimizations combined eliminate or hide 75% of the operating system data misses in 32-Kbyte primary caches. Furthermore, they speed up the operating system by 19%.

The rest of this paper is organized in five sections: Section 2 presents the experimental setup; Section 3 classifies the sources of data references and misses in the operating system; finally, in Sections 4-6, we propose and evaluate support to eliminate the main sources of data misses.

2 Experimental Setup

This section examines the hardware and software systems used to gather our data, the workloads selected, and the architecture simulated.

2.1 Hardware System

This work is based on address traces gathered from a 4-processor bus-based Alliant FX/8 multiprocessor. We use

[1] This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436 and RIA MIP 93-08098; ARPA Contract No. DABT63-95-C-0097; and NASA Contract No. NAG-1-613.

a hardware performance monitor [3] that gathers uninterrupted traces of application and operating system references in real time without introducing significant perturbation. 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 16-Kbyte first level instruction caches. Each probe has a trace buffer that stores over one million references. For each reference, the information stored includes 32 bits for the address accessed, 20 bits for a time-stamp, a read/write bit, and other miscellaneous bits.

The trace buffers typically fill in a few hundred milliseconds. When one of the four buffers nears filling, it sends a non-maskable interrupt to all processors. Upon receiving the interrupt, the processors trap into an exception handler and halt in less than ten machine instructions. Then, a workstation connected to the performance monitor dumps the buffers to disk. Alternatively, the data may be processed while being read from the buffer and then discarded. Once the buffers have been emptied, processors are restarted via another hardware interrupt. With this approach, we can trace an unbounded continuous stretch of the workload. More details are presented in [11].

2.2 Software Setup

The multiprocessor operating system used in our experiments is a slightly modified version of Alliant's Concentrix 3.0. Concentrix is multithreaded, symmetric, and is based on Unix BSD 4.2. All processors share all operating system data structures.

To perform a detailed performance analysis, we need to collect all data and instruction references issued by the processors. However, the performance monitor cannot capture instruction accesses that hit in the first level cache. To get around this problem, we instrument the operating system and application codes by adding machine instructions that cause data reads from specified addresses. The performance monitor can capture these data reads and then interpret their addresses according to an agreed-upon protocol.

To distinguish these *escape accesses* from real accesses, we do as follows. One first type of escape access is used to instrument every single basic block in the operating system. These escapes read odd addresses in the operating system code segment. We can easily distinguish these escapes from real instruction reads because the latter are aligned at even address boundaries. Note that these escapes are very easy to identify because the virtual and physical addresses of operating system code are the same.

The second type of escapes are used to instrument every single basic block in the application code. In the application program that we want to trace, we declare an array whose purpose is for generating escape references by accessing its elements. Accesses to these locations will cause physical addresses to be stored in the trace buffer of the performance monitor. Unfortunately, to make sense of the trace, we need the corresponding virtual address. Consequently, we need to inform the trace buffer of the virtual to physical mapping of the page that contains the array. This is done by having the operating system inform the buffer (again via escape references in the code segment) when a page fault occurs. These escapes will encode what virtual-to-physical page mapping has occurred.

Using this methodology, we insert one escape load at the beginning of each basic block. As a result, we can reconstruct all the virtual addresses of the instructions [11]. However, this instrumentation increases the size of the code by 30.1% on average. For this reason, we carefully compared the statistics gathered with and without this instrumentation for possible skew. We find that the incurred perturbation does not significantly affect the metrics that we measure [11]. In particular, there is no increase in page faulting activity or any significant change of the relative frequency in which operating system routines are executed.

With this instrumentation, we know what basic block was being executed at any point in the trace. Therefore, we can easily map the entries in the trace that are caused by real data accesses to the actual instruction in the assembly code that performed the access. From here, we can determine, in the large majority of cases, the data structure that was being accessed.

Finally, we feed the address traces to a cycle-by-cycle simulator of different memory systems of bus-based multiprocessors. Since we have the complete instruction and data reference trace of the workloads, we can accurately simulate the timing of the workloads in the new memory systems. We do, however, identify the synchronization events in the trace and make sure that their mutual exclusion functionality is maintained in the simulations. We simulate both instruction and data accesses. With the simulations, we determine the impact of the hardware and software support discussed in later sections.

2.3 Workloads

The choice of workloads is a major issue in a study of this type because of its impact on the results. We chose four system-intensive workloads that involve a variety of system activity. We ran each of them for about 15 seconds of real time.

TRFD_4 is a mix of 4 runs of a hand-parallelized version [4] of the *TRFD* Perfect Club code [6]. Each program runs with 4 processes, for a total of 16 processes. The code is composed of matrix multiplies and data exchanges. It is highly parallel yet synchronization intensive. It consists of about 450 lines of Fortran code. The most important operating system activities in the workload are page fault handling, process scheduling, cross-processor interrupts, processor synchronization and other multiprocessor-management functions.

TRFD+Make includes one copy of the hand-parallelized version of *TRFD* and 4 runs of the second phase of the C compiler, which generates assembly code given the preprocessed C code. The compiler phase traced has about 15,000 lines of C source code. Each compilation runs on a directory of 22 C files. The file size is about 60 lines on average. This workload has a mix of parallel and serial applications that forces frequent changes of regime in the machine and cross processor interrupts. There is also substantial paging.

ARC2D+Fsck is a mix of 4 copies of *ARC2D* and one copy of *Fsck*. *ARC2D* is another hand-parallelized Perfect Club code [12]. It is a 2-D fluid dynamics code consisting of sparse linear systems. Each copy runs with 4 processes. It has about 4,000 lines of Fortran code and 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 has about 4,500 lines of C code. It contains a wider variety of 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 in the background at a time. This workload executes a variety of system calls that involve scheduler activity, virtual memory management, process creation and termination, and I/O- and network-related activity.

Table 1: Characteristics of the workloads studied.

Characteristic	Workload			
	TRFD_4	*TRFD +Make*	*ARC2D +Fsck*	*Shell*
User Time (%)	49.9	38.2	42.7	23.8
Idle Time (%)	8.0	8.2	11.5	29.2
OS Time (%)	42.1	53.6	45.8	47.0
Stall Time Due to OS D-Accesses (% of Total Time)	14.0	14.9	11.3	13.3
D-Miss Rate in Primary Cache (%)	3.5	4.7	3.8	3.2
OS D-Reads / Total D-Reads (%)	40.4	53.6	44.5	61.3
OS D-Misses / Total D-Misses (%)	53.4	69.1	66.0	65.9

2.4 Simulated Machine

The baseline architecture that we simulate has 4 200-MHz processors. Each processor has a 16-Kbyte primary instruction cache, a 32-Kbyte primary data cache, and a 256-Kbyte unified lockup-free [14] secondary cache, all direct-mapped. The primary data cache is write through and, like the instruction cache, has 16-byte lines. The secondary cache is write back and has 32-byte lines. There is a 4-deep word-wide write buffer between primary and secondary caches and an 8-deep 32-byte wide write buffer between the secondary cache and bus. Reads bypass writes. We use the Illinois cache coherence protocol under release consistency. The bus is 8-bytes wide, cycles at 40 MHz, and has split transactions. Each secondary cache line transfer uses the bus for 20 processor cycles. Without resource contention, it takes 1, 12, and 51 cycles for a processor to read a word from the primary cache, secondary cache, and memory respectively. All contention is simulated, including cache port and bus access. This architecture, which we call *Base*, will be modified later in the paper.

3 Data References and Misses in the Operating System

Table 1 shows some important characteristics of the workloads studied. The top three rows of the table show the decomposition of the execution time into user, idle, and operating system time. From the table, we see that these workloads often run the operating system (42-54% of the time). The operating system time can be divided into cycles spent executing instructions and cycles spent stalling on memory accesses, either in the instruction or data memory hierarchies. The next row of the table shows that as much as 11-15% of the *total* time is wasted on processor stalls due to operating system accesses to the data memory hierarchy. This includes stalls due to read misses and write buffer overflows. The goal of this paper is to reduce this stall time.

The next three rows of the table examine the behavior of the operating system data references and misses. In this paper, miss rates and misses refer to reads only, not writes. The total miss rate in the 32-Kbyte primary data caches of the simulated machine is 3.2-4.7%. The operating system is responsible for a large fraction of the cache activity. Indeed, as shown in the last two rows, 40-61% of the read references to the data caches, and 53-69% of the data cache misses are caused by the operating system. These applications are, therefore, system intensive.

To understand the reasons for the data misses in the operating system, we classify the misses into three groups: misses during block operations, misses due to coherence activity, and other misses, mostly due to cache conflicts. The contribution of each group is shown in Table 2. From the table, we see that roughly 40% of the misses occur during block operations. The remaining misses are split roughly 1 to 3 between coherence misses and other, mostly conflict, misses. The exception is *Shell* which, because it is a sequential workload, has few coherence misses (6.2%). Overall, the distribution of the misses into the different categories is somewhat similar to the distribution measured in IRIX by Torrellas *et al* [19] and Chapin *et al* [8]. In their papers, these researchers pointed out the impact of block operations and coherence activity. In the rest of the paper, we focus on removing the misses in each these three categories in turn.

Table 2: Breakdown of operating system data misses. Only read misses are measured.

Source of OS Data Misses	Workload			
	TRFD_4	*TRFD +Make*	*ARC2D +Fsck*	*Shell*
Block Op. (%)	43.7	43.9	44.0	27.6
Coherence (%)	14.8	11.3	12.9	6.2
Other (%)	41.5	44.8	43.1	66.2

4 Handling Block Operations

Accesses to the source block in block operations cause the read misses shown in Table 2. These misses stall the processor. In addition to this overhead, however, block operations cause three other overheads: stall due to write buffer overflow while writing the destination block, stall due to future misses on data that is displaced from the cache by the source or destination block during the block operation and, finally, instruction execution.

Figure 1 shows the relative weight of these four overheads, which are named *Read Stall*, *Write Stall*, *Displ. Stall*, and *Instr. Exec.* respectively. The figure shows that, for the architecture studied, each of the *Read Stall*, *Write Stall*, and *Instr. Exec.* overheads accounts for about 30% of the overhead of block operations. The remaining 10% of the overhead is due to *Displ. Stall*. These results are fairly consistent across workloads. In the following, we first analyze these overheads in detail and then, based on the analysis, we evaluate different optimizations.

4.1 Analysis of the Overheads

We examine, in turn, the misses while reading the source block, the write buffer overflow while writing the destination block, and the misses on data displaced by the block operation.

4.1.1 Misses While Reading the Source Block

Cache misses while reading the source block induce a relatively large overhead in block operations (Figure 1). This is because a large fraction of the memory lines of the source block are not in the primary cache when the block operation

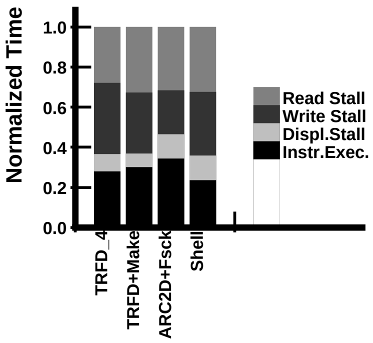

Figure 1: Impact of the different components that contribute to the overhead of block operations.

starts. Indeed, as the first row of Table 3 shows, only 41-71% of the lines of the source block are.

The processor stall time caused by these misses can be reduced by prefetching the source block before the actual data use. Software-based prefetching is in fact quite easy to support and is already present in commodity microprocessors like Alpha [18]. We can use software pipelining to hide more latency and loop unrolling to reduce instruction overhead [16]. Efficiently coding the prolog and epilog stages for these techniques is not very important: blocks are usually large. Indeed, rows 4-6 of Table 3 show that, on average, 55% the blocks have the size of a page (4 Kbytes), while 9% are between 4 and 1 Kbytes, and 36% are less than 1 Kbyte. We also note that the instruction overhead of prefetching is very small. After loop unrolling, we measured that prefetch instructions account for slightly over 5% of the instructions in block operations. Their effect is negligible.

An alternative scheme to reduce the read stall time is to perform the block operations in a DMA-like fashion without involving the processor. With this scheme, block operations will be fast because the data does not have to travel up the memory hierarchy all the way to the processor. Ideally, each word operation will be pipelined and take a few bus cycles. Of course, caches must be kept coherent as the operation progresses. This scheme also has the advantage of requiring very few instructions to perform the operation.

4.1.2 Write Buffer Overflow While Writing the Destination Block

As shown in Figure 1, write buffer overflow causes significant stall time. The large majority of this stall is caused by the write buffer between the cache and the bus. This buffer receives the writes that need to access the bus. These are writes that were directed to words not present in the caches of the writing processor or present in state *Shared*. In the latter case, all that is required is to place an invalidation signal on the bus.

Unfortunately, writes to the destination block usually involve bus accesses. Indeed, Row 2 of Table 3 shows that, right before the block operation, the secondary cache of the writing processor contains, on average, only 21% of the lines of the destination block in state dirty or exclusive. Furthermore, few bus accesses involve simply an invalidation signal: as shown in Row 3 of Table 3, less than 1% of the lines of the destination block are in the secondary cache in state *Shared*.

Obvious techniques to reduce this stall include deeper write buffers and higher bus and memory bandwidth. Alternatively, a scheme that performs the block operations in a DMA-like manner without involving the processor does not suffer from this problem.

4.1.3 Misses on Data Displaced by the Block Operation

Future misses on data displaced from the cache during block operations we call block displacement misses. We have measured the block displacement misses and grouped them into two categories: those that occur while a block operation is in progress and the rest. The former we call *inside* block displacement misses and, as shown in Row 7 of Table 3, account for 1-7% of the total operating system and application data misses. The latter we call *outside* misses and, as shown in Row 8, account for 9-16% of the misses. Overall, adding up the two rows, block displacement misses account, on average, for 16% of all the operating system and application data misses.

Clearly, all block displacement misses will disappear if we do not let the source and destination blocks into the cache during the block operation. Such strategy we call cache bypassing. With cache bypassing, however, first-time reuses of the block data will now cause cache misses. To determine these new misses we simulate cache bypassing in block operations. Of course, if the source or destination word is already in the cache, we access the cache. The new misses that now appear we call *reuses*. As before, we separate the reuses into those that occur while a block operation is in progress and the rest. The former we call *inside* reuses and, as shown in Row 9 of Table 3, would surprisingly number as much as up to 43% of the original data misses. The latter we call *outside* reuses and, as shown in Row 10, would number 1-3% of the original data misses. Overall, adding up the two rows, reuses would account for 29% of the original data misses on average.

These figures give two insights. First, since the number of reuses is higher than the number of displacement misses, if block operations bypass the caches there will be a net increase in the number of misses. Second, this effect is the result of the *inside* component of reuses and displacement misses. Indeed, there are fewer *outside* reuses than *outside* displacement misses while *inside* reuses far outnumber *inside* displacement misses. The latter effect largely results from consecutive block operations sharing the same block. In particular, the destination block of a first block operation is often the source block of a second block operation. This occurs, for example, when a process forks a second process which, in turn, forks a third one: each fork involves a block copy, and the destination block of the first copy is the source block of the second copy. Overall, this second observation suggests that the highest performance will be achieved if block operations bypass the caches (and, consequently, we benefit from the *outside* component of misses and reuses) and we somehow eliminate the impact of the *inside* reuse misses. The latter can be accomplished by performing prefetching during the block operation, thereby eliminating the *inside* reuse misses.

4.2 Evaluation of Optimizations

Based on this discussion, we now evaluate four different supports for block operations. Not all the schemes can use off-the-shelf processors.

- *Blk_Pref*: The data in the source block is software-prefetched into the first and second level caches with prefetch instructions. We perform software pipelining and loop unrolling.
- *Blk_Bypass*: Loads and stores in block operations bypass the two caches. However, to exploit spatial locality, the block data is transferred in chunks equal to the size of a cache line. We add two registers as wide as a

Table 3: Characteristics of the block operations. In the table, *Src* and *Dst* stand for source and destination blocks respectively. Unless otherwise noted, the data refers to the 32-Kbyte primary data cache.

Characteristic	Workload			
	TRFD_4	*TRFD +Make*	*ARC2D +Fsck*	*Shell*
Src lines already cached (%)	62.9	71.1	61.4	41.0
Dst lines already in secondary cache and *Dirty* or *Excl.* (%)	19.6	20.4	40.6	2.6
Dst lines already in secondary cache and *Shared* (%)	0.5	0.6	1.0	0.1
Blocks of size = 4 Kbytes (%)	91.5	70.3	30.8	29.1
Blocks of size < 4 Kbytes and >= 1 Kbyte (%)	1.9	5.2	24.4	3.6
Blocks of size < 1 Kbyte (%)	6.6	24.5	44.8	67.3
Inside displacement misses / total data misses (%)	6.8	5.5	4.1	1.3
Outside displacement misses / total data misses (%)	12.3	9.3	15.8	10.1
Inside reuses / total data misses (%)	42.7	24.3	39.2	1.4
Outside reuses / total data misses (%)	0.8	3.0	1.5	1.4

first level cache line in parallel with the first level cache. One of them holds the line of the source block currently operated upon, while the other holds the corresponding line of the destination block. Similarly, there are two registers as wide as a second level cache line in parallel with the second level cache for the same purpose. Of course, if the source or the destination line is in the caches of the originating processor, a cache access is performed. Loads are blocking.

- *Blk_ByPref*: Combination of *Blk_Pref* and *Blk_Bypass*. The source data is prefetched into a buffer that can hold 8 first level cache lines. This is because one single register for the source data like in *Blk_Bypass* would quickly overflow. The processor can access the prefetch buffer as fast as the primary cache. Writes to the destination block are cached. This is done to simplify the write buffer implementation.
- *Blk_Dma*: A smart cache controller module for the second level cache performs block operations in a DMA-like fashion while holding the bus for the duration of the operation. In the meantime, the originator processor is stalled. Caches are bypassed. The second level cache of the originator processor is read or updated only if it contains the source or destination data respectively. If it is updated, the update propagates to the first level cache. Similarly, using the bus snooping mechanism, the caches of the other processors may be read and updated. These events may slow down the bus transfer. To get started, the operation takes 19 cycles plus any time lost to contention to get the bus. Then, in the best case, the hardware transfers 8 bytes in the bus from source to destination memory every 2 bus cycles. This system can use off-the-shelf processors.

The impact of these optimizations on the number of misses is shown in Figure 2. For each workload, the figure shows the number of operating system read misses in the primary data cache for the *Base*, *Blk_Pref*, *Blk_Bypass*, *Blk_ByPref*, and *Blk_Dma* systems. The bars are normalized to the *Base* system and show the contribution of the misses in block operations. Focusing first on *Blk_Pref*, we can see that prefetching eliminates most of the misses in block operations. The few remaining misses in block operations are due to prefetches not being issued early enough. The misses in the *Other* category do not change. Overall, this simple optimization reduces the misses by 33% on average.

As expected, *Blk_Bypass* generally increases the number of misses. This is the result of a small decrease in the *Other* misses and, except in *Shell*, a sharp increase in block misses. This behavior can be predicted from the data in the last four rows of Table 3. Indeed, the *inside* reuses outnumber the *inside* block displacement misses and, therefore, cache bypassing will increase the misses in block operations. The *outside* reuses, however, are fewer than the *outside* block displacement misses and, therefore, cache bypassing will decrease the misses in the *Other* category. This scheme is obviously undesirable.

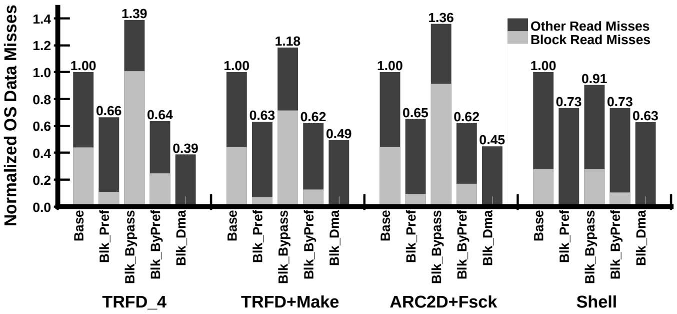

Figure 2: Normalized number of operating system read misses in the 32-Kbyte primary data caches under different support for block operations.

Blk_ByPref keeps the *Other* category of misses as low as in *Blk_Bypass*. This is because the two schemes differ only in the handling of block operations. *Blk_ByPref*, however, sharply decreases the number of misses in block operations. The only remaining ones are those for which the prefetches are not issued early enough. The number of such cases is higher than in *Blk_Pref* because there are more misses to prefetch. Overall, this optimization reduces the misses by 35% on average.

Finally, *Blk_Dma* eliminates all the block misses. This is because caches are bypassed. Furthermore, it keeps the *Other* category of misses as low as in *Blk_ByPref* because block operations do not displace cached data. Overall, with *Blk_Dma*, only 49% of the operating system data misses remain.

The impact of these optimizations on the execution time of the operating system in the workloads is shown in Figure 3. The figure does not include the user execution time because the latter does not change significantly with the optimizations proposed. While the figure shows several systems for each workload, we are currently interested only in the leftmost five bars for each workload. All bars are normalized to the *Base* system. Each bar is broken down into stall due to data read misses not overlapped (*D Read Miss*) or partially overlapped (*Pref*) by prefetches, write buffer overflow (*D Write*), instruction misses (*I Miss*), and instruction execution (*Exec*). Note that the bars include the impact of both primary and secondary cache misses.

Focusing first on *Base*, we see that the *D Write* plus the *D Read Miss* categories account for only 23-34% of the execution time. The remaining time is consumed by overheads targeted only indirectly by the optimizations presented in this paper: instruction execution and instruction misses. For

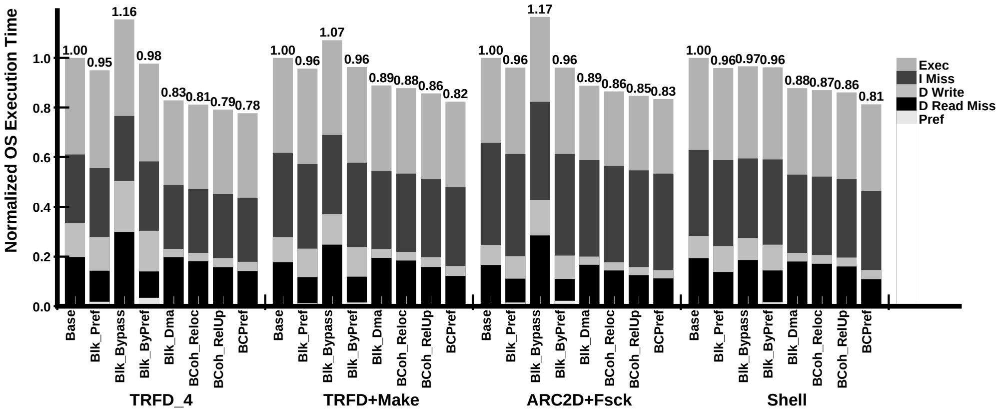

Figure 3: Normalized execution time of the operating system under different levels of support.

Table 4: Characteristics of copies of blocks smaller than a page. These blocks we call small blocks.

Metric	Workload			
	TRFD_4	*TRFD +Make*	*ARC2D +Fsck*	*Shell*
Small Block Copies / Block Copies (%)	11.0	40.7	76.1	83.5
Read-Only Small Block Copies / Small Block Copies (%)	14.0	43.9	25.0	8.7
Misses Eliminated by Deferred Copy / Total Data Misses (%)	0.1	0.4	0.3	0.1

this reason, the speedups that our optimizations can achieve are relatively modest.

From the figure, we see that *Blk_Pref* and *Blk_ByPref* reduce the *D Read Miss* time. This is because some of the read misses are eliminated or hidden. However, since some misses cannot be completely hidden, some *Pref* time appears. Similarly, the *D Write* time increases due to higher bus contention. Indeed, the workloads run faster while generating practically the same number of bus transactions than in the *Base* system. Therefore, there is more bus contention. Overall, the gains are small: *Blk_Pref* runs 4-5% faster, while *Blk_ByPref* runs 2-4% faster.

Blk_Bypass runs slower than *Base* in most cases. This is due to two reasons. First, the higher number of misses present results in a higher *D Read Miss* overhead. Second, cache bypassing causes more writes to be deposited in the write buffers thereby increasing the *D Write* overhead. Cache bypassing in its simple form, therefore, is not a good scheme.

Finally, *Blk_Dma* achieves execution time reductions of 11-17% relative to *Base*. In the figure, it looks as if the *D Read Miss* overhead had increased relative to *Blk_ByPref*. This is an artifact of the way we compute this overhead. Indeed, while the DMA-like block operation is in progress, the processor is stalled. In our accounting, we assign all this stall time to *D Read Miss*, even though some of the stall is likely to be caused by writes and, therefore, should be included in *D Write*. Overall, what is important is that, in *Blk_Dma*, the combination *D Read Miss* plus *D Write* often decreases. Even when it remains constant, the lower number of instructions executed results in a small reduction in the *Exec* and *I Miss* categories. Overall, the data suggests that the impact of *Blk_Dma* is large enough to justify adding the necessary features to support it in bus-based machines.

4.2.1 An Alternative Scheme for Block Copying: Deferred Copying

For block copying, another way to eliminate the costs of the block operation is not to perform the copy until right before the destination or source blocks are written to. Up until that point, all references to the destination block are somehow remapped to access the source block. If the blocks are never written, the copying never takes place. This scheme we call deferred copy.

This scheme is used by the operating system for page-sized blocks. It is known as *copy-on-write*. Copy-on-write is implemented by marking the pages that contain the source block as read-only in the TLB; when a write is attempted, an exception occurs and the copy is performed. Unfortunately, this scheme cannot be easily applied to blocks smaller than a page. However, Cheriton *et al* proposed a deferred copy scheme for blocks of various granularities in the VMP machine [10]. The VMP machine has special cache management mechanisms that support deferred copy. The authors, however, did not evaluate the gains of this mechanism.

To evaluate this mechanism, we first identify all operations that copy blocks whose size is smaller than a page. As shown in the first row of Table 4, these block operations account for 11-83% of all block copies. These copy operations can be divided into two groups: those where the source and destination blocks are never written in our traces after the block operation, and the rest. The former we call *Read-Only* block copies. In read-only block copies, the copy will not take place. The second row of Table 4 shows that read-only block copies account for 9-44% of the block copies considered. Finally, we simulate deferred copying for all blocks smaller than a page and find that only 0.1-0.4% of all the misses in the primary data cache are eliminated in this way (Row 3 of Table 4). Such a low impact in our workloads suggests not to support this scheme.

5 Handling Coherence Misses

The coherence misses in Table 2 are mainly due to barrier synchronization, infrequently-communicated variables, frequently-shared variables, and locks. The weight of each category is shown in Table 5.

Barrier synchronization accounts for 35-46% of the coherence misses in all workloads except *Shell*. The reason for the relatively high barrier activity is that, in the operating

Table 5: Breakdown of coherence misses in the operating system.

Source of Misses	Workload			
	TRFD_4	*TRFD +Make*	*ARC2D +Fsck*	*Shell*
Barriers (%)	45.6	35.0	41.2	4.8
Infreq. Com. (%)	22.1	19.9	22.5	25.5
Freq. Shared (%)	12.6	10.1	14.3	24.7
Locks (%)	7.9	13.5	1.9	19.0
Other (%)	11.8	21.5	20.1	26.0

system considered, parallel programs are all gang-scheduled. With gang scheduling, every time the scheduler chooses to run a parallel program, all processors need to synchronize in a barrier. Since the jobs in *Shell* are serial, barrier synchronization misses are less common.

Infrequently-communicated variables cause 20-25% of the coherence misses in the operating system. Often, these variables are counters that a processor increments when an event happens. These events happen frequently and are recorded by different processors. However, while these counters are updated frequently by different processors, they are used infrequently, usually at regular intervals when certain system-wide operations are performed. For example, the *vmmeter* data structure maintains the *v_intr* counter to count the number of cross-processor interrupts. This counter is incremented by a processor every time the processor receives a cross-processor interrupt. This event occurs frequently and, therefore, the variable is frequently updated by different processors. However, the counter is used only when the pager is invoked, which is relatively infrequently.

This behavior is common because of the way shared-memory multiprocessor operating systems have evolved from uniprocessor operating systems. Given a counter in a uniprocessor operating system, it is quite natural for the system designer to change the declaration of the counter to shared. However, this simple approach causes a large amount of useless misses.

The third category, namely frequently-shared variables, account for 10-25% of the coherence misses. Examples of such variables are pointers to processes in the system resource table. These pointers point to the processes that use a given resource. When the resource is preempted from that process, the pointer needs to be updated.

The next source of misses is locks. They cause 2-19% of the coherence misses. Important kernel locks include those associated with accounting, physical memory allocation, job scheduling, and the high resolution timer. This agrees with [19] and [8]. Finally, 12-26% of the coherence misses are due to other effects, including false sharing.

Note that, unlike in [19], we do not find many misses due to process migration. This is because Concentrix, unlike IRIX, does not allow processes to migrate. However, we now capture misses on synchronization variables, while the performance monitor in [19] was unable to. Overall, based on our observations, we propose three optimizations to reduce the number of coherence misses: data privatization, data relocation, and selective update. In the following, we discuss them in detail.

5.1 Data Privatization and Relocation

A natural approach to reduce the misses in infrequently-communicated variables is to privatize them. For example, given the *vmmeter.v_intr* counter that counts the number of cross-processor interrupts, we split it into as many private sub-counters as processors in the machine. To avoid false sharing, we place each private sub-counter in a different cache line. Then, we change the code of the pager so that, instead of reading one counter, it reads all the private sub-counters and adds them all up.

We also perform some data relocation. For example, variables that are clearly accessed in sequence are placed in the same cache line. As a result, an access to one of them fetches them all. Similarly, we identify the most obvious cases of false sharing and relocate the variables responsible for it to different cache lines. Obviously, the operating system designer needs advanced monitoring tools to detect these effects.

The impact of data privatization and relocation is shown in Figure 4. The figure is organized like Figure 2 and shows the number of operating system read misses in the primary data caches for several systems: *Base*, *Blk_Dma*, *BCoh_Reloc*, and *BCoh_RelUp*. *BCoh_Reloc* is *Blk_Dma* plus data privatization and relocation. *BCoh_RelUp* will be considered next. In the figure, the misses are divided into coherence misses and the rest. The figure shows that data relocation and privatization reduces the number of misses by 10% on average. Both coherence and *Other* misses decrease. *Other* misses decrease because spatial locality is enhanced.

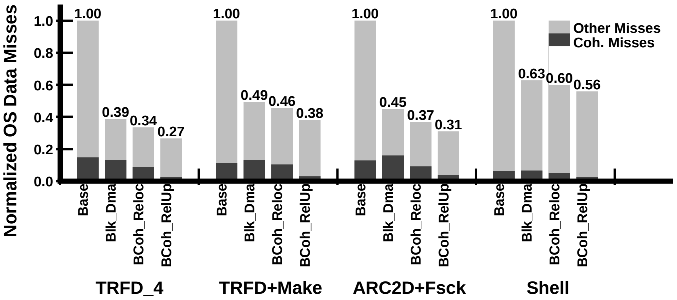

Figure 4: Normalized number of operating system read misses in the 32-Kbyte primary data caches under different optimizations.

Finally, Figure 3 shows the impact of data privatization and relocation on the execution time. From the figure, we see that the difference between the *Blk_Dma* and *BCoh_Reloc* bars is only about 2%. The impact of this optimization, therefore, is very small. However, it has no hardware cost.

5.2 Selective Update

To further eliminate coherence misses, we consider the use of an update protocol. Applying an update protocol to all operating system variables, however, would surely create too much traffic. This is because the dominant sharing patterns in the operating system are not producer-consumer. Indeed, roughly speaking, shared-memory multiprocessor operating systems are generated by replicating the operations of a uniprocessor operating system in several threads and then adding synchronization to ensure data consistency.

However, if we apply the update protocol to certain key variables only, we may eliminate many misses with little traffic cost. In our experiments, we use the Firefly [5] update protocol on three sets of variables. The first set is barriers, which amount to 48 bytes. The sharing behavior of barriers clearly favors an update over an invalidate protocol. The second set of variables is the 10 most active locks. It was indicated in [19] that most operating system locks tend to be acquired several times in a row by the same processor. Clearly, this is not the optimal pattern for update protocols. However, we are willing to tolerate some useless traffic if misses are eliminated. Each synchronization variable is placed in a different cache line.

Finally, the third set of variables includes some of the most frequently-shared ones that exhibit, at least in part, a producer-consumer behavior. They amount to a total of 176 bytes. This set includes *freelist.size*, which keeps the number of free pages or *cpievents*, an array that contains information on the processor causing a cross-processor interrupt. Of course, some of these variables may cause some useless updates.

Overall, these three sets of variables use 384 bytes. Since they are all statically allocated, they can all be allocated in one page. This is the approach that we simulate. Alternatively, if the machine is such that synchronization variables must be allocated from a special memory, these variables can be allocated in two pages, one for synchronization variables and one for regular variables. These pages are then filled up with seldom used variables. We then simulate the update protocol for these pages and the invalidate protocol for the rest. Note that this optimization can be supported with off-the-shelf processors if operating system variables are TLB-mapped. 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. Obviously, the operating system designer needs to have sophisticated tools to identify the right variables to use updates on.

The miss reduction resulting from applying the update protocol to these 384 bytes and the invalidation protocol to the rest is shown in Figure 4. Bars *BCoh_RelUp* correspond to the system in *BCoh_Reloc* plus this optimization. From the figure, we see that this optimization eliminates most of the coherence misses, reducing the total number of operating system misses in the primary cache by 15% on average. This is a significant reduction. Furthermore, it can be shown that this miss reduction is obtained with only a 3-6% increase in the amount of bus traffic over the invalidation protocol. Overall, choosing only a small subset of variables to use updates on seems to be a good choice: the resulting number of operating system data misses is only 1-3% higher than in a pure update protocol, while it saves 31-52% of the update traffic.

The impact of this optimization on the execution time is shown in the *BCoh_RelUp* bars of Figure 3. Comparing the *BCoh_RelUp* bars to the *BCoh_Reloc* bars, we see that the former have less *D Read Miss* time. As a result, the operating system execution time decreases by an average of 2% relative to *BCoh_Reloc*. Again, while this is a small gain, it is achieved with no extra hardware support.

6 Handling the Remaining Misses

The operating system still suffers some, mostly conflict, misses. To eliminate them, we try two techniques. First, we attempt to detect obvious cache conflicts between data structures. If detected, one of the data structures can be relocated and the conflict misses will disappear. Second, we perform data prefetching to hide the latency of the remaining misses.

To identify obvious cache conflicts, we use trace simulations to determine the pair of data structures involved in each conflict miss. In the analysis of the simulation results, we do not consider dynamically-allocated operating system data structures. This is done to ensure that results are repeatable across operating system reboots. Overall, the result of this expensive simulation is that, for our operating system, no two data structures suffer obvious conflicts with each other. Instead, a given data structure suffers conflicts with several data structures. These conflicts we call *random* conflicts. Therefore, no relocation is performed.

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 most misses, we determine the source code statements that cause the misses. These statements constitute miss hot spots. Miss hot spots tend to be the same for different workloads.

For this experiment, we have selected the 12 most active miss hot spots. They account for 29, 44, 22, and 51% of the remaining operating system data misses in the primary caches in *TRFD_4*, *TRFD+Make*, *ARC2D+Fsck*, and *Shell* respectively. These hot spots are 5 loops and 7 sequences. A sequence [20] is an ordered set of basic blocks executed with high frequency and in the same order. These miss hot spots do the following:

- Four loops loop over the array of page table entries, performing initialization or copies of several entries. One loop traverses a linked list of pages to find a free page.
- The 7 sequences include those that resume a process, perform timer functions for system accounting, execute the trap system call, perform context switching, and schedule a process. Typical prefetchable data structures are the table of system call functions or the timer data structure.

Note that many miss hot spots are not loops. This is a result of the relatively low frequency of loops in the operating system [20]. We expect that other UNIX systems will have quite similar miss hot spots.

Once the miss hot spots have been identified, we manually insert the prefetches. We use a prefetch instruction like the one used by *Blk_Pref* in Section 4.2 and supported, for example, by Alpha [18]. For the loops, we perform loop unrolling and software pipelining [16]. For the sequences, we move the prefetches as early as possible in the sequence. Often, however, the unavailability of the operands to compute the address to prefetch limits how far back the prefetches can be pushed. Another problem is that, sometimes, even if the prefetch is moved to the first few instructions of the sequence, not all the latency of the prefetch can be hidden. In this case, the prefetch should be moved to the one or more callers of the sequence. In our simple algorithm, however, we do not do this. In fact, we insert only a few prefetches in the whole kernel. The prefetches increase the dynamic instruction count of the miss hot spots by only 3.2%.

The result of applying prefetching is shown in the *BCPref* bars of Figure 5 (for *Block+Coherence+Prefetching*). The figure shows the number of operating system read misses in the primary data caches for *Base*, *Blk_Dma*, *BCoh_RelUp* and *BCPref*. The *BCPref* bars correspond to a system like *BCoh_RelUp* plus the prefetch optimization described here. In the figure, misses are divided into those in the miss hot spots and the rest.

The figure shows that *BCPref* hides practically all hot spot misses. The *Other* misses are not affected. On average, *BCPref* hides 32% of the misses in *BCoh_RelUp*. Overall, few misses now remain after the block, coherence, and prefetching optimizations: only 21-28% of the original ones.

Ideally, *BCPref* should not increase bus traffic because our hand-inserted prefetches rarely prefetch useless data. To confirm this, we compared the bus traffic in *BCPref* and *BCoh_RelUp*. We find that the amount of traffic in both systems differs by less than 1%. Traffic increase, therefore, is not an issue.

To see how this miss reduction translates into speedups,

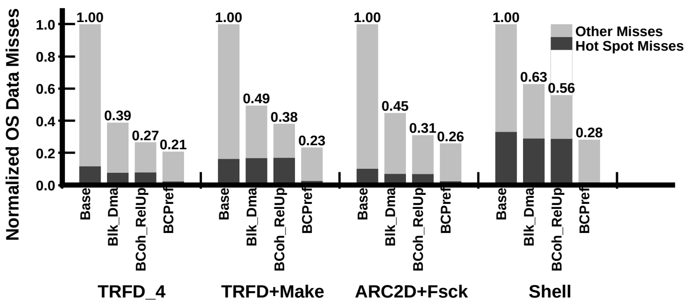

Figure 5: Normalized number of operating system read misses in the 32-Kbyte primary data caches under different optimizations.

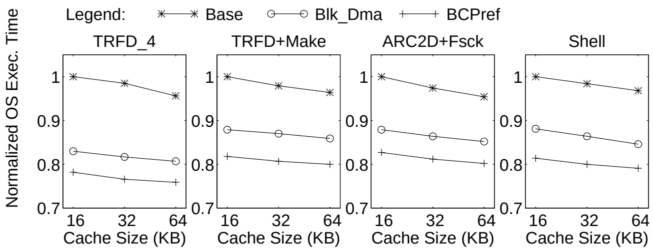

Figure 6: Normalized operating system execution time of the workloads for different primary data cache sizes. The line size of the cache is kept at 16 bytes. The secondary cache has 32-byte lines and a size of 256 Kbytes.

we examine the last bar of each workload in Figure 3. The *BCPref* bars show that the addition of the prefetches causes a reduction in the *D Read Miss* time. Furthermore, the overhead of the prefetch instructions is negligible. This can be seen by comparing the *Exec* time for *BCoh_RelUp* and *BCPref*. Finally, prefetches do not increase the bus contention. This can be deduced from the constant contribution of *I Miss* in *BCoh_RelUp* and *BCPref*. Overall, this optimization further speeds up the operating system by 4% on average.

After all the optimizations that we have proposed, the operating system runs on average 19% faster. Note that this has been accomplished without requiring any modification to existing commodity microprocessors. Moreover, the user execution time is practically unaffected by the proposed optimizations. This is because the prefetches discussed in Section 4 and 6 are almost always useful. For this reason, we have not shown charts with the user execution time.

7 Discussion

The set of optimizations that we propose, namely *BCPref*, can be supported by existing commodity processors. Unfortunately, the software involved requires sophisticated performance monitoring tools and is hard to automate. For example, the insertion of the prefetches and selection of the coherence protocol are based on a trace analysis that currently requires manual involvement. However, we feel that, given the importance of the operating system, such costs are acceptable.

Further optimizations are possible. However, the number of operating system misses remaining in the primary data caches is so small that optimizations are likely to have a low impact. Possible optimizations that can be attempted are page placement schemes that reduce conflicts in the secondary cache [7, 13], and the insertion of more prefetches. The former optimization has the shortcoming that the data placement is done at a page grain size, which is not optimal for the many small data structures in the kernel; the latter optimization has the problem that the pointer-intensive nature of the operating system makes it hard to add useful prefetches.

We have measured the effect of our optimizations on several different cache configurations. The results show that the optimizations work well for all of them. For example, Figure 6 shows the impact of the optimizations for different primary data cache sizes. The sizes of the caches range from 16 to 64 Kbytes, while the line size is fixed at 16 bytes. From the figure, we can see that *Blk_Dma* always outperforms *Base*, while *BCPref* always outperforms *Blk_Dma*.

Similarly, Figure 7 shows the impact of the optimizations for different line sizes in the primary data cache. The sizes of the caches lines range from 16 to 64 bytes, while the cache size is fixed at 32 Kbytes. From the figure, we can see that, again, *Blk_Dma* always outperforms *Base*, while *BCPref* always outperforms *Blk_Dma*. We do not show the user time in Figures 6 or 7 because it is practically unaffected by our optimizations.

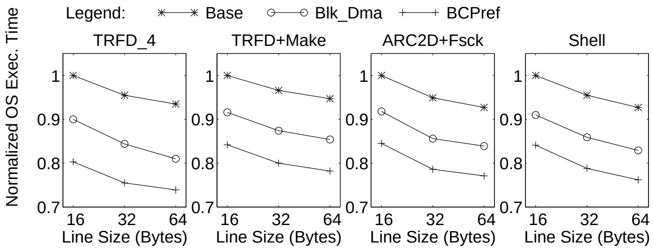

Figure 7: Normalized operating system execution time of the workloads for different line sizes in the primary data cache. The size of the cache is kept at 32 Kbytes. The secondary cache has 64-byte lines and a size of 256 Kbytes.

8 Summary

Previous work by several researchers has consistently indicated that the operating system does not use the data cache hierarchy efficiently. In this paper, we have focused on how to eliminate or hide most of the data misses in a multiprocessor operating system. To do so, we examined traces of a 4-processor bus-based shared-memory machine running system-intensive workloads under UNIX. Based on our observations, we proposed and evaluated via detailed simulations hardware- or software-based improvements that can be supported by off-the-shelf processors.

Our optimizations target misses induced by block operations, coherence activity and cache conflicts. For block operations, simple cache bypassing or prefetching schemes are undesirable. Instead, we propose a DMA-like scheme that pipelines the data transfer in the bus without involving the processor. For coherence misses, we suggest data privatization and relocation, and using updates for a small core of shared variables. Finally, for the remaining miss hot spots, we suggest data prefetching. These hot spots include a few loops and several frequently-executed sequences of basic blocks. On average, our optimizations eliminate or hide 75% of the operating system data misses in 32-Kbyte primary caches and speed up the operating system by 19%. Finally, we feel that significant further miss reductions will be difficult to achieve with optimizations supported by off-the-shelf processors.

Acknowledgments

We thank Liuxi Yang, Russ Daigle, Tom Murphy and Perry Emrath for their help with the hardware and operating system. We also thank the referees and the graduate students in the I-ACOMA research group for their feedback. Josep Torrellas is supported in part by an NSF Young Investigator Award.

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

[4] J. B. Andrews. Parallelization of TRFD. Internal Document, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, November 1991.

[5] J. Archibald and J. L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[6] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[7] B. N. Bershad, Dennis Lee, Theodore H. Romer, and J. B. Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, October 1994.

[8] J. Chapin, S. A. Herrod, M. Rosenblum, and A. Gupta. Memory System Performance of UNIX on CC-NUMA Multiprocessors. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 1–13, May 1995.

[9] J. B. Chen and B. N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 120–133, December 1993.

[10] 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.

[11] R. Daigle. A Minimally Perturbed Hybrid Hardware/Software Address Tracing Scheme for Multiprogramming, OS and Multiprocessor Workloads. Master Degree Thesis, Computer Science Department, University of Illinois, October 1994.

[12] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. The Cedar Fortran Project. Technical Report 1262, Center for Supercomputing Research and Development, October 1992.

[13] R. Kessler and M. Hill. Page Placement Algorithms for Large Real-Indexed Caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[14] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, 1981.

[15] A. Maynard, C. Donnelly, and B. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, October 1994.

[16] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

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

[18] R. L. Sites. Alpha AXP Architecture. *Digital Technical Journal*, 4(4), 1992.

[19] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174, October 1992.

[20] J. Torrellas, C. Xia, and R. Daigle. Optimizing Instruction Cache Performance for Operating System Intensive Workloads. In *IEEE Trans. on Computers*, To appear 1995. A shorter version appeared in Proceedings of the 1st International Symposium on High-Performance Computer Architecture, pages 360-369, January 1995.