Optimizing Instruction Cache Performance for Operating System Intensive Workloads[1]

Josep Torrellas, **Chun Xia**, and **Russell Daigle**[2]

Center for Supercomputing Research and Development
and Computer Science Department
University of Illinois at Urbana-Champaign, IL 61801

Abstract

High instruction cache hit rates are key to high performance. One known technique to improve the hit rate of caches is to use an optimizing compiler to minimize cache interference via an improved layout of the code. This technique, however, has been applied to application code only, even though there is evidence that the operating system often uses the cache heavily and with less uniform patterns than applications. Therefore, it is unknown how well existing optimizations perform for systems code and whether better optimizations can be found. We address this problem in this paper.

This paper characterizes in detail the locality patterns of the operating system code and shows that there is substantial locality. Unfortunately, caches are not able to extract much of it: rarely-executed special-case code disrupts spatial locality, loops with few iterations that call routines make loop locality hard to exploit, and plenty of loop-less code hampers temporal locality. As a result, interference within popular execution paths dominates instruction cache misses. Based on our observations, we propose an algorithm to expose these localities and reduce interference. For a range of cache sizes, associativities, lines sizes, and other organizations we show that we reduce total instruction miss rates by 31-86% (up to 2.9 absolute points). Using a simple model this corresponds to execution time reductions in the order of 10-25%. In addition, our optimized operating system combines well with optimized or unoptimized applications.

1 Introduction

High-performing instruction memory hierarchies are fundamental for the memory system to keep up with fast CPUs. To intercept instruction fetches with low latency, many machines rely on small on-chip instruction caches. Given that a miss may cause many tens of idle cycles for the CPU, these small caches must be able to capture the working set of the programs. Fortunately, it has been shown that high instruction reference locality helps caches intercept most of the references. Many common loads like transaction processing, compilations, or large multiprogramming mixes, however, often involve heavy use of the operating system. Given that the operating system code has a complex functionality, a large size, and interrupt-driven transfers of control among its procedures, caches may be less effective in intercepting the accesses in these workloads.

Indeed, there is some evidence that backs this claim. Clark [10] reported a lower performance of the VAX-11/780 cache when operating system activity was taken into account. Similarly, Agarwal *et al* [1] pointed out the many cache misses caused by the operating system. Torrellas *et al* [17] reported that the operating system code causes a large fraction of the cache misses and, in addition, suffers considerable self-interference in the cache. They also show that these self-interference misses are concentrated in relatively narrow ranges of addresses. Similarly, Chen and Bershad [7] report that systems code has lower locality than application code. They also point out the self-interference

[1]A short version of this paper was presented at First International Symposium on High-Performance Computer Architecture. This work was supported in part by the 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 grant 1-1-28028 from the Univ. of Illinois Research Board.

[2]Currently with Tandem Computers Inc., Cupertino, California.

in the cache. Other researchers like Ousterhout [16] and Anderson *et al* [3] also indicate the different nature of the operating system activity. Finally, Nagle *et al* [15] point out that instruction cache performance is becoming increasingly important in new-generation operating systems. Clearly, given the practical importance of achieving high hit rates in small instruction caches, the caching behavior of systems code needs to be understood better and improved.

Improving the performance of caches has been addressed by many researchers. It has been shown that it is feasible to reduce the misses in applications via improved code layout in the cache [11, 13, 14]. The technique is based on repositioning or replicating code, usually to reduce cache conflicts. McFarling's technique [13] uses a profile of the conditional, loop, and routine structure of the program. With this information, he places the basic blocks so that callers of routines, loops, and conditionals do not interfere with the callee routines or their descendants. Hwu and Chang's technique [6, 11] is based on identifying groups of basic blocks within a routine that tend to execute in sequence. These basic blocks are then placed in contiguous cache locations. Furthermore, routines are placed such that frequent callee routines follow immediately after their callers. The resulting spatial locality and low interference saves many misses. They also investigate function inlining [8] but find it largely ineffective because code expansion increases cache conflicts. Finally, Mendlson *et al* [14] perform code replication based on static information to eliminate conflicts. In most cases, the results are good. However, given the complexity of the operating system, it is not known whether similar methods can be successfully applied to it.

In this paper, we characterize and optimize the instruction cache performance of system intensive workloads. The workloads run on a 4-CPU Alliant FX/8 multiprocessor under a commercial multiprocessor Unix. We focus on two issues. Firstly, with the help of a hardware performance monitor, we characterize in detail the locality patterns of the operating system. We show that there is substantial locality to be exposed. However, rarely-executed special-case code disrupts spatial locality, loops with few iterations that call routines make loop locality hard to exploit, and plenty of loop-less code hampers temporal locality. As a result, interference within popular execution paths dominates instruction cache misses. Secondly, we design and evaluate new code placement algorithms tailored to expose the locality in systems code. As a result, many conflict misses disappear. For the range of cache organizations studied, we reduce total instruction miss rates by 31-86% (up to 2.9 absolute points). Furthermore, we compare our scheme to one of the best existing ones and consistently outperform it by a significant amount. The effectiveness of our technique is not affected by the degree of placement optimization for the application.

This paper is organized in four sections: Section 2 presents the experimental setup, Section 3 analyses the instruction miss and reference patterns of systems code, Section 4 presents our optimizing algorithm and, finally, Section 5 evaluates it.

2 Experimental Setup

This section examines the hardware and software systems used to gather our data and the workloads selected. While this work is done in the context of a parallel machine, our conclusions are also applicable to uniprocessors. We use a multiprocessor to capture a larger range of systems activity, including multiprocessor scheduling and cross-processor interrupt activity.

2.1 Hardware System

This work is performed on a 4-processor bus-based Alliant FX/8 multiprocessor. We use a hardware performance monitor that gathers uninterrupted reference traces of application and operating system in real time without introducing perturbation. The performance monitor [4] has one probe connected to each of the four processors. The probes collect all instruction and data references issued by the processors except those that hit in the per-processor 16-Kbyte first level instruction cache. 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 any of the four buffers nears filling, it sends a non-maskable interrupt to all processors. Upon receiving the interrupt, 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 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. Furthermore, this is done with negligible perturbation because the processors are stopped in hardware.

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 symmetric and is based on Unix BSD 4.2. All processors share all operating system data structures.

The performance monitor cannot capture instruction accesses that hit in the first level instruction cache. Therefore, since we want to collect all instruction accesses, the operating system and application codes must be instrumented. To do this with little perturbation, we execute single machine instructions that cause data reads to specified addresses. Recall that there is no first-level data cache. The performance monitor can capture the addresses read from and interpret them according to an agreed-upon protocol. This methodology was suggested in [17].

To distinguish these *escape accesses* from real accesses, we do as follows. One first type of escapes, used for operating system instrumentation, reads odd addresses in the operating system code segment. We can easily distinguish these escapes from instruction reads since the latter are aligned on even address boundaries. Furthermore, these escapes are easy to control and identify because virtual addresses for operating system code are equal to their physical addresses.

The second type of escapes are used for application tracing. In the application program that we want to trace, we declare an array whose purpose is for generating escape references by reading its elements. However, accesses to these locations will cause physical, not virtual addresses to be stored in the trace buffer of the performance monitor. Therefore, we need to inform the trace buffer of the virtual to physical page mapping. This is done by having the operating system inform the buffer (via escape accesses in the code segment) when a page fault occurs. These escapes will be encoded to tell the trace buffer what virtual-to-physical page mapping has occurred. Hence, when analyzing the address trace, we can reconstruct the virtual addresses of the array in the application.

In our setup, we first insert escape sequences at the entry and exit of each routine. With this information, we gather statistics such as the most frequently executed routines and the common paths through the operating system and application code. This minimal amount of instrumentation causes an inappreciable amount of perturbation. For some experiments, however, we need to instrument the beginning of every basic block with an escape reference. This information is needed to generate profiling information of basic block execution. This instrumentation is more intrusive to the behavior of the programs, since it increases the size of the code used by 30.1% on average[1]. For this reason, we carefully compared the statistics gathered with and without this instrumentation for possible skew. The statistics gathered from this kernel are close to those from the minimally instrumented kernel for the metrics that we are analyzing (for example, most frequently executed

[1]This relatively large increase is in part the result of instrumenting a non-RISC assembler code: the Alliant processors use Motorola 60820 assembler code.

routines). Hence, the incurred perturbation is not significant. In particular, there is no increase in page faulting activity.

Once the address traces are generated, we use three main tools to process them. The simplest one generates statistics such as the most frequently executed basic blocks, routine parent/child relationships or common execution paths through the operating system. It then generates a basic block flow graph with profile information. The next tool analyzes several of these basic block flow graphs, takes the average of them all and, using our algorithm, generates an optimized basic block layout for the operating system and application. The final tool is the cache simulator, with which we determine the effectiveness of the new basic block layout. We simulate several different cache organizations.

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 tried to choose four system intensive workloads that involve a variety of system activity.

TRFD_4 is a mix of 4 copies of a hand parallelized version of the *TRFD* Perfect Club code [5]. Each program is run with 4 processes. The code is predominately composed of matrix multiplies and data interchanges. It is highly parallel yet synchronization intensive. The most important operating system activity in this application is process scheduling, cross-processor interrupts, processor synchronization, and other multiprocessor-management functions. Each program consists of about 450 lines of Fortran code.

TRFD+Make is a mix of one copy of *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 4 compilations, each 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 force frequent changes of regime in the machine and cross processor interrupts. There is also substantial paging. The compiler phase traced has about 15,000 lines of C source code.

ARC2D+Fsck is a mix of 4 copies of *ARC2D* and one copy of *Fsck*. *ARC2D* is another hand-parallelized Perfect Club code. It is a 2-D fluid dynamics code consisting of sparse linear systems solvers. It runs with 4 processes, although it is not as highly parallel as *TRFD*. It has about 4,000 lines of Fortran code and causes operating system activity like that of *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 I/O code than *Make*. It has about 4,500 lines of C code.

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 into the background. This workload executes a variety of system calls that involve context switching, scheduler activity, virtual memory management, process creation/termination, I/O and network-related activity. While we have not been able to run any database workload, *Shell* has some similarity with database loads in that both loads have heavy system call activity.

Each workload runs for a total of about one minute of real time. For most of the experiments, we take the average of the four processors in the machine.

3 Analysis of the Instruction Miss and Reference Patterns

To gain insight into the cache performance of operating system intensive workloads we now examine the miss and reference patterns of systems code. We focus on locality issues. We will use this insight later to design the code placement algorithms.

3.1 Miss Patterns

Misses on systems code are clustered in relatively narrow address ranges. As an example, Figure 1-(a) shows the instruction misses in the operating system for one of the 16 Kbyte direct-mapped instruction caches of the Alliant FX/8 multiprocessor running *TRFD+Make*. In the figure, the misses are shown as a function of the virtual address of the code that suffers the miss.

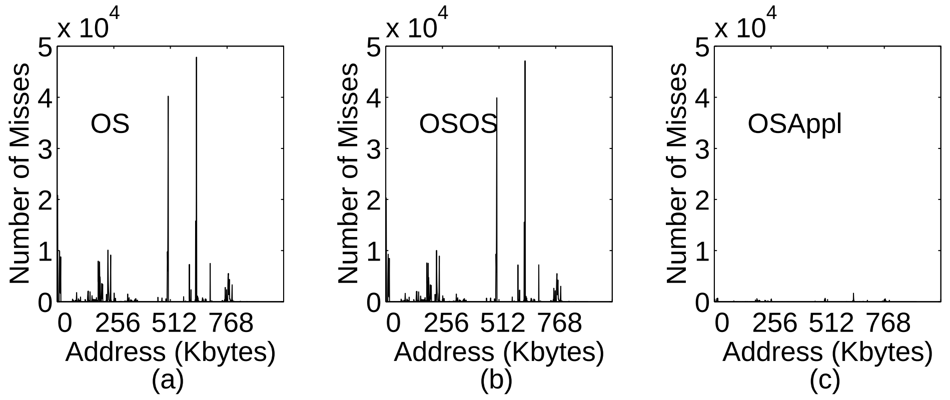


Figure 1: Number of misses on operating system code in one of the Alliant FX/8's 16 Kbyte caches as a function of the virtual address of the code. The data corresponds to *TRFD+Make*. Charts (a), (b) and (c) show the total misses, the component caused by self-interference, and the component caused by interference with the application respectively. Misses caused by first-time references are negligible. The data in the charts corresponds to only one processor in the machine. Each data point corresponds to the misses on a 1-Kbyte address range.

These instruction misses are caused by either first-time references, self-interference, or interference with the application. Since the misses resulting from first-time references are negligible, the misses in Figure 1-(a) are divided into self-interference misses (Chart (b)) and misses resulting from interference with the application (Chart (c)). These charts, in agreement with those for the other workloads, show that for medium to small direct-mapped caches running the operating system intensive workloads that we studied, operating system misses are dominated by self-interference. Indeed, self-interference misses account for over 90% of the operating system misses in all the workloads studied. This is because the transitions between application and operating system account for little.

Many of these misses are caused by repeated conflicts between two frequently-called routines or between caller and callee routines within the same popular execution path. This effect creates sharp peaks of misses. For example, the highest peak in Figure 1-(b) is caused by conflicts between the routines that handle the timer and those that perform multiplication and division. This peak contains 21.3%, 12.6%, 16.2%, and 3.5% of the total operating system misses in *TRFD_4*, *TRFD+Make*, *ARC2D+Fsck*, and *Shell* respectively. Similarly, the other high peak in Figure 1-(b) is caused by conflicts between the routines that perform user/system transitions and those that handle the beginning of system calls. This peak contains 14.3%, 8.6%, 8.7%, and 11.6% of the total operating system misses in *TRFD_4*, *TRFD+Make*, *ARC2D+Fsck*, and *Shell* respectively. As we see, these peaks dominate the misses in all the workloads studied. Unfortunately, however, conflicts vary from operating system recompilation to recompilation. Therefore, ad hoc optimizations are not appropriate. Instead, we need to design automatable techniques that produce an optimized instruction layout for systems code. To do so, we first need to examine the locality in the reference patterns of the operating system code.

3.2 Reference Patterns

Several important characteristics of the instruction reference patterns of the operating system are shown in Table 1. The data in the table corresponds to one processor in the machine. The first three rows of the table show the size of the operating system sections that the workloads execute. From the table, we see that the workloads execute 32,000-123,000 bytes of the code (first row in the table), which correspond to 3.4-13.1% of the code (second row in the table), or 3.6-13.4% of the basic blocks (third row of the table). Combining all workloads, only 18% of the operating system code is ever referenced and only 26% of the routines are ever invoked.

Table 1: Characteristics of the operating system instruction references. The data corresponds to one processor in the machine. *BB* stands for basic block and *Invoc* for invocations.

OS Code Characteristics	Workload			
	TRFD_4	*TRFD +Make*	*ARC2D +Fsck*	*Shell*
Size of Executed OS Code (Bytes)	31,866	122,710	76,228	92,908
Size of Executed OS Code (%)	3.4	13.1	8.1	9.9
Number of Executed OS BBs (%)	3.6	13.4	8.8	10.4
Interrupt Invoc. (% of Total Invoc.)	76.0%	65.7%	73.8%	29.7%
Page Fault Invoc. (% of Total Invoc.)	23.0%	21.3%	21.9%	12.0%
SysCall Invoc. (% of Total Invoc.)	0.0%	11.2%	2.4%	54.7%
Other Invoc. (% of Total Invoc.)	1.0%	1.8%	1.9%	3.6%

This data illustrates the first important characteristic of the operating system, namely that large sections of its code are seldom accessed. The reason is that the operations performed by the operating system include many special cases that happen very infrequently but that need to be handled for correctness. The significance of this observation is that most of the code has little impact on cache performance. Therefore, we have more freedom to perform optimizations for the frequently-executed routines.

The large concentration of the operating system references in certain routines can be graphically seen in Figure 2. For each workload, the figure plots the number of references to operating system instructions as a function of the virtual address of the referenced instruction. The data in the figure corresponds to only one processor in the machine. From the figure, we can see the uneven distribution of the references. Clearly, the size of the code accessed in each of the workloads is a small fraction of the total kernel size.

A second characteristic of the operating system references is that different workloads generally exercise the same popular routines. Indeed, as shown in Figure 2, the peaks are in similar positions in the different charts. This means that many popular routines are common to all these workloads. To gain insight into these workloads, we measured the reasons why the operating system was invoked. The invocations are classified into four main classes:

- Interrupts: invocations to service interrupts like cross-processor, clock, I/O, or multiprocessor synchronization interrupts.
- Page Faults: invocations to service page faults or TLB misses.
- System Calls: invocations to execute system calls.
- Other: other invocations.

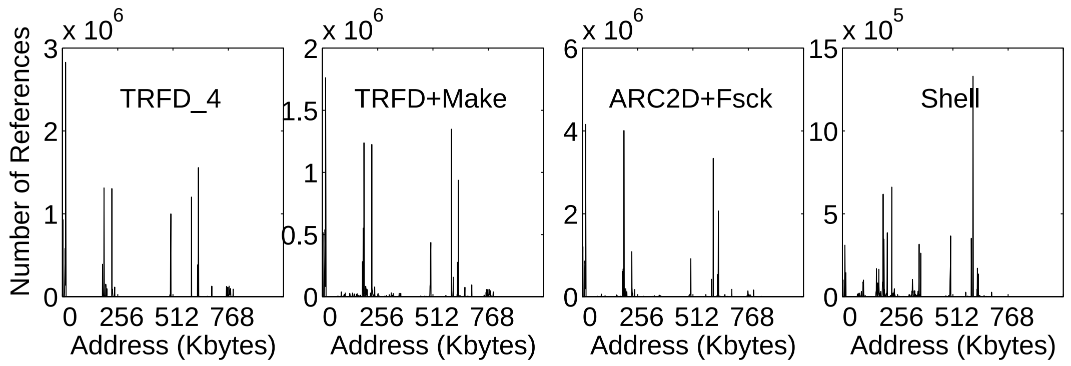


Figure 2: Number of references to operating system code as a function of the virtual address of the referenced instruction. The data corresponds to only one processor in the machine. Each data point corresponds to the references to an address range of 1 Kbyte.

The last three rows of Table 1 break down the operating system invocations in each workload into the four categories described. From the data, we can see that each workload invokes the operating system in different ways. For example, in *TRFD_4*, interrupt invocations dominate. This is the result of the cross-processor interrupts and synchronizations that take place when running parallel applications. In *Shell*, on the other hand, system call invocations dominate. This is because the workload invokes many operating system services. The other two workloads exhibit intermediate behavior. Overall, therefore, these workloads look different. However, as Figure 2 shows, these differences do not prevent them from sharing many of the popular sections of the operating system code.

To optimize the cache performance, we will perform transformations that expose the locality in the areas that are frequently accessed. For our purposes, we distinguish three types of locality, namely spatial, loop, and temporal locality. We consider each of them in turn.

3.2.1 Spatial Locality

Previous work had indicated that the operating system code had low spatial locality. This is a result of the many branches necessary to get around seldom-executed code. However, we do not care about the original spatial locality of the code. Instead, we are interested in how deterministic the sequences of executed basic blocks are, independently of how far apart in the code these basic blocks are. This is because we can always place these basic bocks close by in memory and expose spatial locality.

An examination of operating system address traces shows that they often contain regular repeatable sequences of instructions. A given set of such instructions, which we call a *sequence*, may span tens of routines, and it is not part of any obvious loop. Sequences are the result of complex operating system functions entailing series of fairly deterministic operations with little loop activity. Examples of such functions are handling a page fault, processing an interrupt, or the first stages of servicing a system call. We note that a large fraction of the references and misses in the operating system occur in a set of clearly defined and frequently executed sequences. Furthermore, many such misses are the result of interference within the same sequence. For instance, the two high peaks in Figure 1-(b) are caused by such interference.

Figure 3 gives initial evidence of the determinicity with which the operating system code is executed. The figure shows the number of outgoing arcs that have a given probability of being used given that the basic block that they leave is executed. To generate the figure, we use the address traces of all the workloads and examine the arcs that connect the basic blocks in the operating system code. These arcs include conditional and unconditional branches, basic block fall-throughs, and procedure calls and returns. From the figure, we see that most arcs either have a very high or a very low probability of being used after the basic block is executed. Indeed, 73.6% of the arcs have a probability larger or equal to 0.99. Similarly, 6.9% of the arcs have a probability smaller or equal to 0.01. Overall, therefore, transitions between basic blocks are fairly deterministic. This implies that the sequences of executed basic blocks are deterministic.

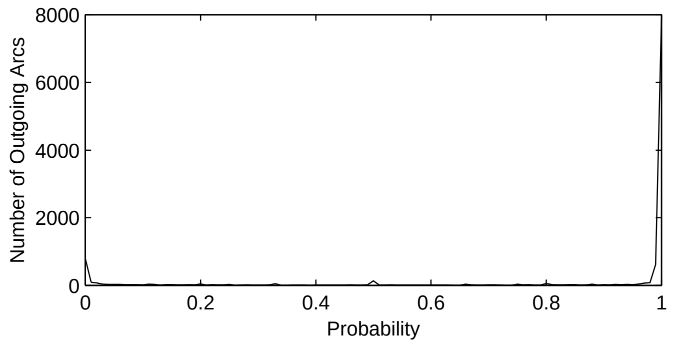


Figure 3: Distribution of the probability that an outgoing arc is executed given that the basic block that it leaves is executed.

To expose spatial locality we begin by identifying *seeds*, or starting basic blocks of sets of sequences. These seeds are readily observable in our operating system and we believe that they are easily identifiable in other operating systems as well. They are the starting points of common operating system functions. Generally, the code associated with them is written in assembly. Examples of seeds are the starting point of interrupt handling, page fault handling, system call servicing, or context switching. Then, starting from the seeds, we use a greedy algorithm to build the 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 callee, following the control transfer out of the basic block with the highest probability of being used. We stop when all the successor basic blocks have already been visited, or they have an execution count smaller than a given fraction of the total number of basic block invocations (*ExecThresh*), or all the outgoing arcs have less than a certain probability of being used (*BranchThresh*). In that case, we start again from the seed looking for the next acceptable basic block. Note that we often end up placing some of the basic blocks of a callee routine surrounded by basic blocks of the caller. This is one of the main differences between an algorithm proposed by Chang and Hwu [6] and ours. Once we have created the sequences out of the four seeds, we catenate them and place them in the cache contiguously. With this placement, we expose much spatial locality and, consequently, reduce self-interference misses.

We will describe some details of the algorithm in Section 4. In this section, however, we show that the basic blocks that our algorithm would select for the most important sequences exhibit very predictable control transfer patterns. This proves that there is significant spatial locality to expose. Furthermore, these basic blocks account for a large fraction of the cache misses. This implies that the rewards of exposing spatial locality will be high.

For this experiment, we consider the sequences that would fit without self-conflict in an 8 Kbyte cache and those that would fit in a 16 Kbyte cache (Table 2). The former, which we call *core* sequences and show in the leftmost part of the table, have a total size of about 7.8 Kbytes. The latter, which we call *regular* sequences and show in the rightmost part of the table, have a total size of about 14.5 Kbytes. Obviously, regular sequences are a superset of core sequences and are created with lower values of *ExecThresh* and *BranchThresh*. The core sequences contain a total of 471 basic blocks, spanning 61 routines; the regular sequences contain 832 basic blocks spanning 89 routines.

Table 2: Characteristics of the sequences of basic blocks. Recall that regular sequences are a superset of core sequences. BB stands for basic block.

Workload	Core Seq. (471 BBs spanning 61 routines)					Regular Seq. (832 BBs spanning 89 routines)				
	Predictability		Weight			Predictability		Weight		
	Probab. Go to Any BB in Seq.	Probab. Go to Next BB in Seq.	Static # BBs (% of Exec'd)	# Refs. (%)	# Misses (%)	Probab. Go to Any BB in Seq.	Probab. Go to Next BB in Seq.	Static # BBs (% of Exec'd)	# Refs. (%)	# Misses (%)
TRFD_4	0.99	0.77	27.7	67.1	74.7	0.98	0.79	38.3	73.8	88.5
TRFD+Make	0.97	0.72	7.4	51.4	63.1	0.96	0.79	13.1	56.7	75.5
ARC2D+Fsck	0.97	0.72	11.3	59.8	67.7	0.96	0.79	20.0	64.0	79.5
Shell	0.95	0.71	7.8	22.7	34.9	0.96	0.77	13.0	37.6	56.6

Table 2 shows that the basic blocks in these sequences exhibit very predictable control transfer patterns. Indeed, Column 2 shows that the execution of a basic block in a core sequence has probability 0.95-0.99 of being followed by the execution of another basic block in a core sequence. Furthermore, Column 3 shows that it has probability 0.71-0.77 of being followed by the execution of the next basic block in the same sequence. The corresponding numbers for regular sequences are 0.96-0.98 (Column 7) and 0.77-0.79 (Column 8). Given that the average size of these basic blocks is 21.3 bytes, a miss on a 64 byte cache line prefetches 2 other basic blocks that will often be executed immediately or soon after. Consequently, there is significant spatial locality to expose. In addition, these sequences cause most of the misses. Indeed, while core sequences account for a small fraction of the executed basic blocks (7-28% in Column 4), they involve a large fraction of the references (23-67% in Column 5) and cause most of the misses (35-75% in Column 6). Regular sequences show the same behavior: 13-38% of the basic blocks cause 38-74% of the references and as much as 57-88% of the misses. Consequently, exposing spatial locality can potentially remove many misses.

3.2.2 Loop Locality

The second type of locality that we examine is the locality provided by loops. To identify the loops, we use dataflow analysis [2]. For our analysis, we divide the loops into those that do not call procedures and those that do. We now consider each category in turn.

Loops Without Procedure Calls

Our measurements show that these loops do not dominate the execution time of the operating system. This can be seen in Table 3: Column 2 shows that the dynamic count of the instructions in these loops is as low as 29-39% of all instructions. The equivalent number for *TRFD_4* is 69% and for *ARC2D* is 96%. The table also shows that the static count of the instructions in these loops is as low as around 3% of the executed instructions (Column 3) or around 0.2% of all instructions (Column 4).

Furthermore, these loops tend to execute few iterations per invocation. This is illustrated in the leftmost chart of Figure 4, which shows a distribution of these loops according to how many iterations are executed per loop invocation. Of the total number of 156 different loops executed, 50% execute 6 or fewer iterations per invocation. Furthermore, about 75% execute 25 or fewer.

Table 3: Fraction of the operating system instructions that belong to loops without procedure calls.

Workload	Dyn. Loops/ Dynamic OS (%)	Static Loops/ Static Exec'd OS (%)	Static Loops/ Static OS (%)
TRFD_4	34.5	3.9	0.1
TRFD+Make	28.9	2.9	0.4
ARC2D+Fsck	31.5	2.7	0.2
Shell	39.4	3.0	0.3

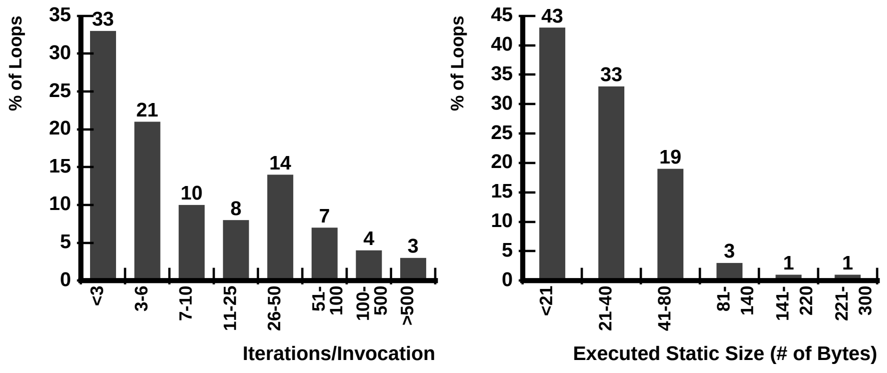


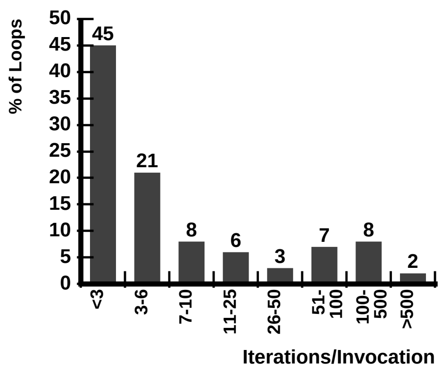
Figure 4: Behavior of the operating system loops that do not call procedures. The leftmost chart shows the distribution of the number of iterations per loop invocation, while the rightmost chart shows the distribution of the static size of the executed part of the loops.

Finally, the static size of the executed part of these loops is small. For example, as shown in the rightmost chart of Figure 4, the largest of these loops spans 300 bytes. Therefore, barring conflicts, caches should have no problem intercepting these loops.

Loops With Procedure Calls

These loops execute complex operations, often distributed among many tens of routines. For instance, one of these loops occurs when the memory allocated by a process has to be freed up after the process dies. The operating system has to loop over all the page tables freeing up the pages and page table entries. At the same time, lots of checks need to be made, like checking if the pages are shared and, if so, decrement counters instead of freeing memory.

Figure 5 characterizes the 71 such loops that we isolated. The leftmost chart shows the distribution of the number of iterations per invocation; the rightmost chart shows the static size of the executed part of the loops, including the size of the executed part of the routines they call and their descendants. As seen in the leftmost chart, these loops have few iterations per invocation, usually 10 or less. However, as shown in the rightmost chart, their size is huge. Their median size is 2 Kbytes, and a few of them have more than 16 Kbytes. Therefore, it is difficult for caches to cache these loops effectively.



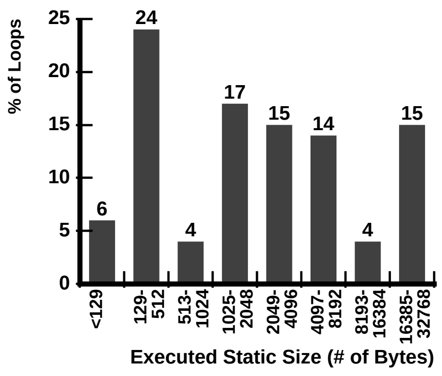


Figure 5: Behavior of the operating system loops that call procedures. The leftmost chart shows the distribution of the number of iterations per loop invocation. The rightmost chart shows the static size of the executed part of the loops including the static size of the executed part of the routines they call and their descendants.

Overall, we would like to layout each loop in the cache so that no two instructions in the loop or its callee procedures conflict with each other. In this case, misses will be limited to the first iteration of the loop. While such a layout is easy to devise for loops without procedure calls, it is hard for those with callees. Indeed, subroutines need to be placed carefully. Furthermore, subroutines are often called by more than one loop. These subroutines, therefore, would have to be laid out avoiding conflicts with more than one loop. We will consider these issues in Section 4.

3.2.3 Temporal Locality

The final type of locality that we consider is temporal locality. To gain insight into this locality, we counted the number of times that each routine is invoked dynamically and then ordered the routines from higher to lower value of this count. Figure 6 shows the normalized value of such count. The figure shows that there are a few routines that are invoked much more frequently than the rest. Indeed, while there are about 600 different routines executed, a few of them account for most of the invocations. One of the main characteristics of such routines is that they often have a very small size, specially if we consider only the section that is often executed. Examples of such routines are those that perform lock handling, timer management, state save and restore in context switches and exceptions, TLB entry invalidation, or block zeroing.

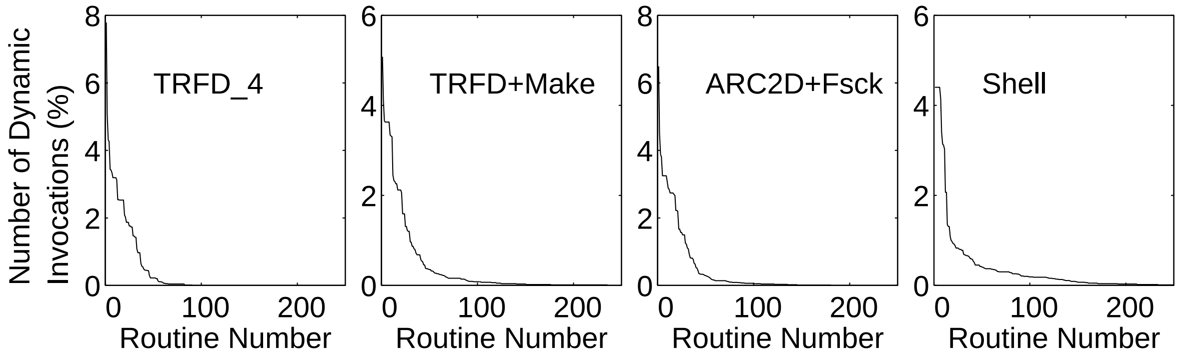


Figure 6: Number of times that each operating system routine is invoked dynamically. For a given chart, the routines are ordered from most to least frequently executed. The charts are then normalized to 100 invocations.

To determine the amount of temporal locality in the most popular routines we measure the number of operating system instruction words fetched between two consecutive calls to a given routine. The measurements are taken *within* an operating system invocation and the statistics are reset across invocations. We perform this measurement for the 10 most popular routines and then take the average. The result is presented in Figure 7 as a histogram. The figure shows that, when one of these routines is called, it has about 25% probability of being called again in less than 100 instruction words, and as much as about 70% probability of being called in less than 1,000 instruction words. As shown in the *Last Inv* column, it has about 9% probability of not being called again in this operating system invocation. The data corresponds to the average of the four workloads. This clearly shows that there is temporal locality to be exploited.

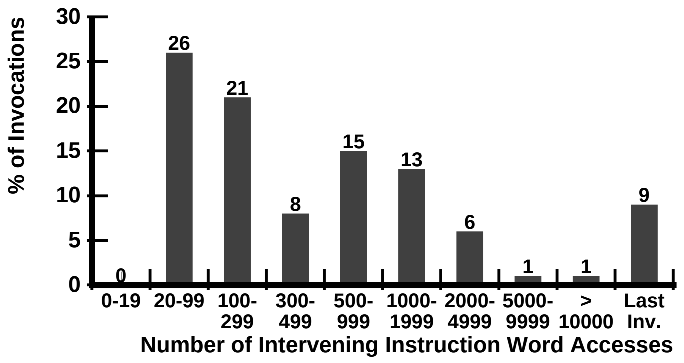


Figure 7: Understanding temporal locality. The chart shows the number of operating system instruction words referenced between two consecutive calls to the same routine in the same operating system invocation. The data corresponds to the 10 most frequently invoked routines and is the average of the four workloads.

Unfortunately, temporal locality may be hard to exploit. This is because, as we saw before, the operating system has relatively few loops. Indeed, between two consecutive invocations of one of these popular routines, the operating system may execute much code, instead of sitting in a tight loop. The result is that there is a high probability of displacing the popular routine from the cache before it is reused.

A more dramatic picture of the skew in instruction execution can be seen by measuring the number of times that each basic block is executed. Figure 8 shows a chart similar to Figure 6 for basic blocks instead of routines. The figure includes the executed basic blocks for all the workloads and orders them according to the number of times they are executed. To eliminate the distortion caused by loops, we assume that loops only perform one iteration per invocation. In the figure, the peak on the left side of the chart reaches 5%.

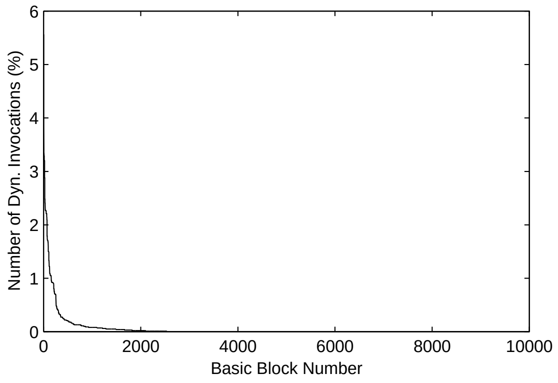


Figure 8: Number of times that each basic block in the operating system is invoked. The approximately 8,500 executed basic blocks in the operating system are ordered from most to least frequently executed and the chart is then normalized to 100 invocations. The data corresponds to the union of the four workloads. The peak on the left of the chart reaches 5%.

From the figure, we see that a few basic blocks account for the majority of the basic block invocations. Indeed, of the about 8,500 basic blocks executed, 22 are executed more than 3.0% of the total invocations, and 157 are invoked more than 1.0% of the total invocations. On the other hand, there are nearly 6,000 basic blocks that are executed less than 0.01% of the total invocations.

To exploit temporal locality, we must ensure that the most frequently executed basic blocks remain in the cache. This is done by assigning the most frequently executed basic blocks to a contiguous range of addresses (*SelfConfFree*). Then, we make sure that only rarely-executed code is placed in addresses that can conflict with the SelfConfFree area. Since there is plenty of rarely-executed code, this is feasible.

3.2.4 Summary

The analysis so far leads us to four conclusions. First, since only a moderate fraction of the operating system code is used frequently, generating an optimized placement will be simpler. Second, there is plenty of spatial locality in sequences. Third, important loops have few iterations and often call procedures, which makes it difficult to exploit loop locality. Finally, there is temporal locality in small routines called frequently. With this evidence, we now proceed to design an algorithm that generates an optimized code layout.

4 Instruction Placement Algorithm

Our algorithm uses the experimental evidence discussed above to expose the three types of localities more and, as a result, minimize operating system self-interference misses. In our algorithm, we represent the program as a directed flow graph $G = \{V, E\}$. Each node $v_i \in V$ denotes a basic block and each arc $e_j \in E$ denotes a transition between two basic blocks. Transitions are caused by conditional and unconditional branches, basic block fall-throughs, and procedure calls and returns. Each node and arc has a weight that is determined via profiling. The node and arc weights are the number of times that the basic block and the transition are executed respectively. All unexecuted basic blocks and transitions are pruned from the graph. With all this information, we will now lay out the code to expose the three types of locality. In the following, we consider each type of locality in turn.

4.1 Spatial Locality

We start by exposing spatial locality with the generation and placement of the sequences of basic blocks. For this algorithm, we use the *ExecThresh* and *BranchThresh* thresholds described in Section 3.2.1. The weight of a node divided by the sum of the weights of all nodes gives a ratio that we compare to *ExecThresh*; the weight of an outgoing edge divided by the weight of the node it leaves gives a ratio that we compare to *BranchThresh*. In our algorithm, we have a loop that repeatedly selects a pair of values for *ExecThresh* and *BranchThresh*, generates the resulting sequences, and places them in memory. In each iteration of this loop, we lower the values of *ExecThresh* and *BranchThresh*, therefore capturing more and more rarely-executed segments of code.

An example of how we apply the algorithm is shown in Figure 9. Chart *(a)* in the figure shows the basic block flow graph for a set of four operating system routines, namely *push_hrtime*, *read_hrc*, *check_curtimer*, and *update_hrtimer*. For simplicity, the figure does not show the return-from-subroutine arcs and all unexecuted basic blocks are removed. The basic blocks are numbered. In the example, we assume that basic block zero of *push_hrtime* is a seed and that the other three routines are not called from any other routine. The number associated with each node is its weight divided by the sum of the weights of all nodes; the number associated with each arc is its weight divided by the weight of the node the arc comes from.

Chart *(b)* shows the resulting basic block layout in memory after applying our algorithm in two passes: first with (*ExecThresh*, *BranchThresh*) = (0.01, 0.1) and then with (*ExecThresh*, *BranchThresh*) = (0, 0). In the (0.01, 0.1) pass, we first place nodes 0, 1, 4, and 8 of *push_hrtime*. Then, since node 8 calls routine *read_hrc*, we place nodes 0, 1, 2, and 3 of *read_hrc*. Then, we place nodes 9, 10, 11, 12 of *push_hrtime*, nodes 0, 1, 2, and 5 of *check_curtimer*, node 13 of *push_hrtime*, node 0 of *update_hrtimer*, and nodes 14, 15, 17, 18, and 19 of *push_hrtime*. Since node 19 does have any successor, we start again from the seed and find node 16 of *push_hrtime* as another placeable basic block. Finally, in the (0, 0) pass, we place nodes 5 and 7 from *push_hrtime*.

By iteratively selecting lower and lower values of the thresholds, we place the code in the cache in segments of decreasing frequency of execution. This minimizes the impact of self-interference because popular sequences will be placed close to other equally popular ones and therefore cannot conflict with them. In our algorithm, we select the sets of values for (*ExecThresh*, *BranchThresh*) so that the length of each of the most important sequences ranges from 1 to 4 Kbytes. While the exact length of each sequence is not that important, it is preferable to keep it as short as 1-4 Kbytes to reduce conflicts. For sequences with less popular basic blocks, we let each sequence grow longer.

The sets of values that we selected for (*ExecThresh*, *BranchThresh*) in our algorithm are shown in Table 4. The first set is chosen somewhat arbitrarily to be *ExecThresh* = 1.4% and *BranchThresh* = 40%. As shown in the table, these values give reasonably-sized sequences. Then, successive

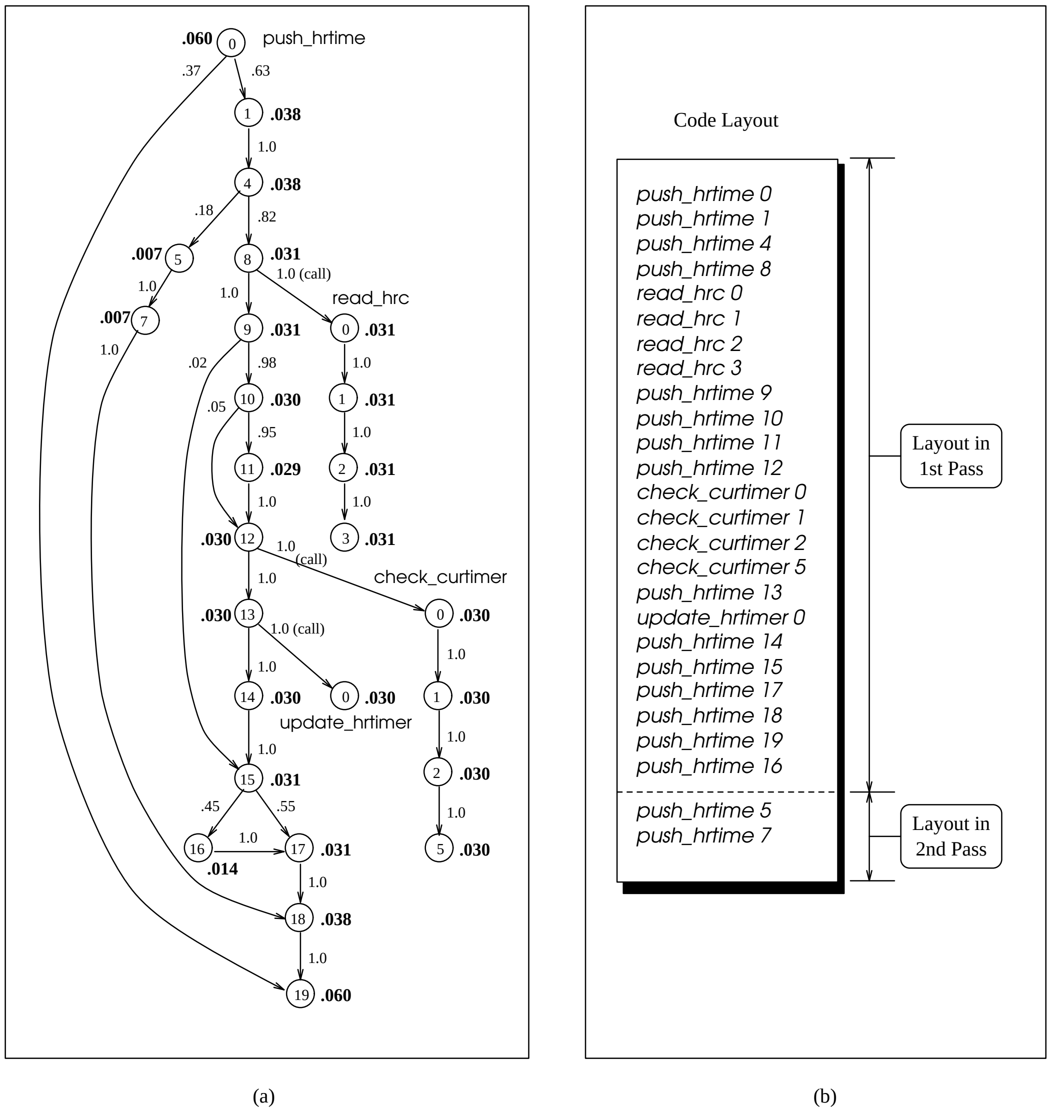


Figure 9: Placing sequences of basic blocks for a set of four routines. Chart *(a)* shows the basic block flow graph, while Chart *(b)* shows the resulting basic block layout in memory.

iterations use values for these two thresholds that usually decrease by one order of magnitude every time (Table 4). This is repeated until all operating system code is selected.

Table 4: *ExecThresh* and *BranchThresh* values used to generate the sequences. For each seed, we show, from top to bottom, the successive values of *ExecThresh* and *BranchThresh* used. For each pair of values, we show the length of the resulting sequence in number of basic blocks (BB) and bytes.

	Seeds			
	Interrupt	Page Fault	SysCall	Other
ExecThresh (%)	*BranchThresh* (%) # of BBs # of Bytes	*BranchThresh* (%) # of BBs # of Bytes	*BranchThresh* (%) # of BBs # of Bytes	*BranchThresh* (%) # of BBs # of Bytes
1.4	0.4 49 810	- - -	- - -	- - -
0.5	0.1 139 1946	0.4 28 576	- - -	- - -
0.1	0.01 79 1618	0.1 116 2146	0.4 70 1208	- - -
0.01	0.01 379 4784	0.01 235 5254	0.1 889 14274	0.4 282 4272
10^{-5}	0.001 391 6382	0.01 1222 26712	0.01 2457 52146	0.1 194 2934
0	0 166 1988	0 410 7474	0 1340 25154	0 86 1851

With this algorithm, we improve both the placement of the basic blocks within routines and the relative placement of the routines. These two aspects of placement optimizations are already addressed by Chang-Hwu's [11] algorithm. However, our algorithm further exposes spatial locality by generating sequences that cross routine boundaries. For example, a sequence may contain a few basic blocks of the caller routine, then the most important basic blocks of the callee routine, and then a few basic blocks more from the caller routine.

A possible alternative to our scheme could be function inlining. In function inlining, the whole callee routine is inserted between the caller's basic blocks, not just a few basic blocks of the callee. Function inlining, however, expands the active code size and may increase the chance of conflicts. Indeed, while Chen *et al.* [8] limited inlining to frequent routines only, their results revealed that inlining may not be a stable and effective scheme. For this reason, we do not consider inlining in our algorithm.

4.2 Temporal Locality

To exploit temporal locality, we start by dividing the memory into logical caches. Logical caches are memory regions of size equal to the cache and which start at addresses that are multiples of the cache size (Figure 10). In the lowest *SelfConfFree* bytes of each logical cache we will not place sequences. Instead, in the SelfConfFree area of the first logical cache we place the most frequently-executed basic blocks of the operating system. These basic blocks are selected in order, pulled out of the sequences they belonged to, and placed in the SelfConfFree area in order until it fills. Then, in the SelfConfFree area of the other logical caches we place seldom-executed code. This way, the most popular *SelfConfFree* bytes in the operating system will seldom or never conflict with any other operating system instruction. The resulting layout is shown in Figure 10. The figure contains a loop area that will be described later.

In our optimization, we do not want to favor the basic blocks that belong to loops because loops will be optimized independently (Section 4.3). For this reason, when we compute the number of times that a basic block is executed, if the basic block belongs to a loop, we assume that the loop had only one iteration per invocation. Overall, after performing some experiments described in Section 5.3, we conclude that a good size for the SelfConfFree area in 4-32 Kbyte caches is 1 Kbyte. Filling 1 Kbyte of each logical cache with seldom-executed code is not difficult given that there is abundant seldom-executed code in the operating system.

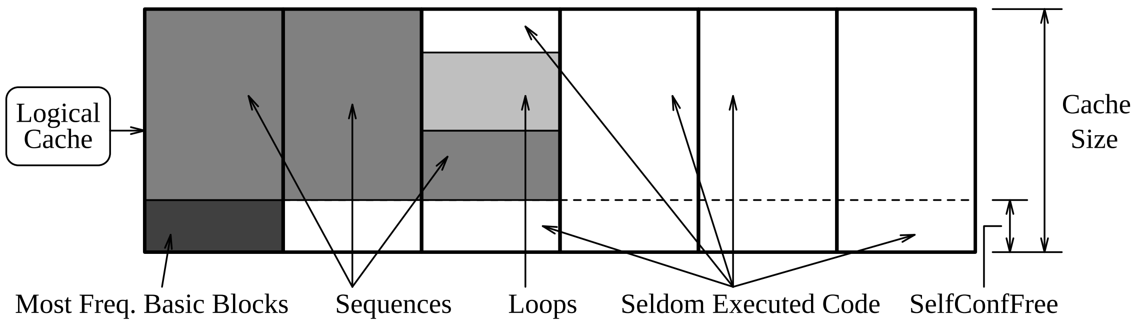


Figure 10: Optimized layout of the code in memory. Addresses increase from bottom to top within a cache-sized chunk and then from left to right.

4.3 Loop Locality

Finally, to exploit loop locality, we start by identifying all the loops in the code. We use dataflow analysis as discussed by Aho *et al* [2]. Then, we select the loops with at least a minimum number of iterations per invocation (currently set to 6). We pull the basic blocks of these loops out of the sequences and put them, in the same order, in a contiguous area at the end of the sequences. With this change, we compact the sequences, thereby potentially exploiting more spatial locality and reducing the chance of conflicts within long sequences. Since the basic blocks extracted belong to a loop, they are not likely to suffer many misses. Indeed, if the loop does not call any routine, the basic blocks will at most suffer misses in the first iteration of the loop. In our algorithm, however, we perform this optimization on both loops that call routines and loops that do not call routines. The resulting layout is shown in Figure 10.

To perform the basic block motion required to expose the three localities, we have to add extra branches, and therefore the code increases in size. However, since we also remove some branches, the increase in dynamic size is, on average, as low as 2.0%.

4.4 Advanced Optimizations

To optimize the loops that call routines, we have attempted a more advanced optimization. Unfortunately, while we describe the optimization here, it usually causes slowdowns and, therefore, we did not include it in our algorithm. The basic idea of the optimization is to make sure that the basic blocks of the loop and the basic blocks of the routines the loop calls and their descendants are placed so that they do not conflict with each other. If this is possible, then any misses will be confined to the first iteration of the loop. Obviously, if the loop executes many iterations, the misses in the first iteration will not matter.

To perform this optimization, we start by identifying loops that have at least a minimum number of iterations per invocation (currently set to 6) and call routines. Once a loop is identified, we determine all the routines it calls and their descendants. Then, we assign each loop to one logical cache. The logical caches chosen are those past the memory area assigned to sequences and loops in Figure 10. For each logical cache, the loop is placed at address SelfConfFree bytes past the beginning of the logical cache. This is done to avoid conflicting with the most frequently-executed basic blocks in the first logical cache. The layout of the loop is then followed by the basic blocks of the routines the loop calls and their callees. This way, the loop and its callees do not interfere. Obviously, all the code placed now has to be pulled out of the sequences, thereby compacting the sequences further.

Unfortunately, two loops may call the same routine. To identify this case and to keep the number of routines under consideration small, the algorithm is a bit more complex. The algorithm uses a data structure called the conflict matrix. The conflict matrix lists in its X-axis the loops with callees and its Y-axis the routines called by at least one of these loops. This matrix simply identifies which loops call which routines. The routines are ordered by their invocation frequency, and those with less than a certain threshold invocation frequency are removed from the matrix. Currently, the threshold is set such that only 50 routines are kept in the matrix.

After the matrix is generated, we place each caller loop in a different logical cache. Then, we successively pick up the next most popular routine in the matrix and place it in the correct logical cache as close as possible to the caller loop. If a given routine is called by two loops, we select an area of the two corresponding logical caches that has not been used by any routine placed so far. Then, in one of the logical caches, we place the routine, while in the other logical cache we place unrelated rarely-executed code. This way, the callee routine does not conflict with any of the two caller loops or the routines they invoke. To illustrate the algorithm, Figure 11 shows an example memory layout with memory organized as in Figure 10. In the figure, loop *L1* calls routines *r1*, *r2*, and *r3*; loop *L2* calls *r2* and *r4*; and loop *L3* calls *r5*. Note that, to prevent *r2* from conflicting with *L2* or *r4*, we have to leave a gap of size equal to *r2*'s size between *L2* and *r4*.

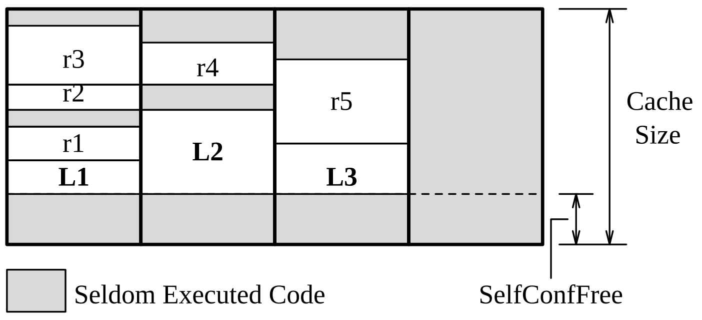


Figure 11: Optimizing the placement of loops that call routines.

While the algorithm described may seem intuitive, it only achieves a small reduction in the number of conflict misses and it does so at the expense of failing to expose much spatial locality. Indeed, the main purpose of the optimization is to reduce the number of conflicts between a loop and its callee routines. The impact of this optimization, however, is small because these loops have, for the most part, a low number of iterations per invocation. Unfortunately, this optimization also disrupts the exploitation of spatial locality. This is because the callee routines are pulled out of the sequences where they helped expose spatial locality. Furthermore, they are moved into an area where they are interspersed with rarely-executed code and therefore can expose less spatial locality. Overall, therefore, we do not implement this optimization in our algorithm. We present an evaluation of this optimization in Section 5.5.

5 Evaluation

After describing our algorithm in the previous section, we now examine its performance impact in a variety of situations. We examine different levels of optimization: *Base* refers to the original unoptimized layout; *C-H* refers to the layout generated by Chang-Hwu's [11] algorithm; *OptS* refers to our layout with SelfConfFree area, sequences, and no loop optimization; *OptL* is *OptS* plus the simple loop optimization described in Section 4.3; finally, *OptA* is *OptS* plus the layout of the application optimized with sequences and the simple loop optimization described in Section 4.3. For the applications, we do not set up any SelfConfFree area because the behavior can vary widely among applications. Furthermore, we use the `main` function as the seed to generate sequences, and place the sequences in the cache starting from the side opposite to that used for the operating system. In all experiments, the layouts are created after taking the average of the profiles of all the workloads.

For one of our workloads, *Shell*, the application references are not available. However, the accuracy is not affected significantly because the number of application references issued by this shell script with *who*, *finger*, and similar commands is tiny. In the following, we first examine the effect of the level of optimization; then we consider the effect of the cache size, SelfConfFree area size, cache line size, and associativity; finally we look at more advanced optimizations.

5.1 Effect of the Level of Optimization

The effect of the different level of layout optimization on the number of misses is shown in Figure 12. For reference purposes, the leftmost chart shows the breakdown of the normalized number of references into operating system and application references. The rightmost chart shows the number of misses in an 8 Kbyte direct-mapped cache with 32 byte lines for different layouts. The bars are grouped in workloads. For each workload, we plot the misses for *Base*, *C-H*, *OptS*, *OptL*, and *OptA*, all normalized to *Base*. Each bar is divided, from bottom to top, into operating system misses caused by self-interference and by interference with the application, and application misses caused by interference with the operating system and by self-interference. Cold misses are negligible.

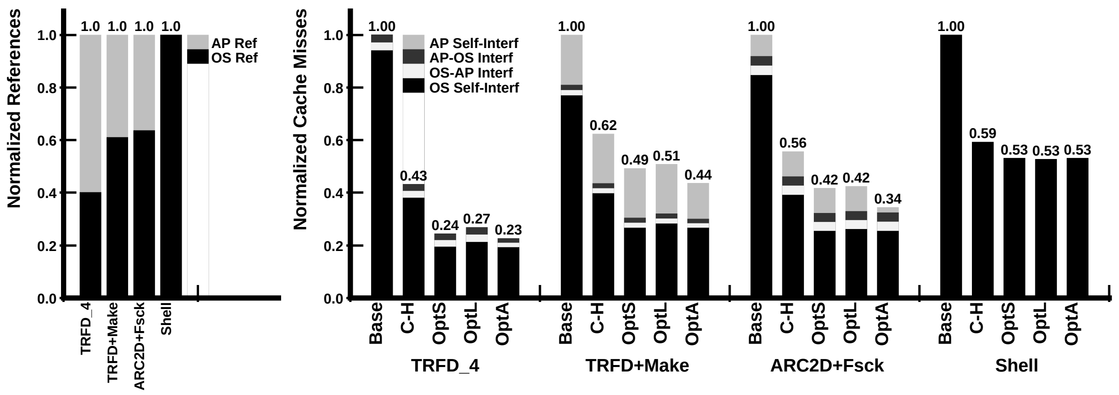


Figure 12: Normalized number of references (leftmost chart) and misses (rightmost chart) for different levels of layout optimization in an 8 Kbyte direct-mapped cache with 32 byte cache lines.

Examining the references, we see that for three workloads the operating system accounts for 40-60% of the references; for *Shell*, all the references shown belong to the operating system. As intended, these are indeed system intensive loads. Looking now at the *Base* miss bars for all the workloads, we see that most of the misses are caused by the operating system. This is partly the result of choosing well-behaved applications like loop-intensive scientific programs: both *TRFD* and *ARC2D* have a tiny miss rate that waters down the effect of *Make* or *Fsck*. We also note that the large majority of misses are the result of self-interference as opposed to cross-interference. This is because we examine a small cache: the misses in the transition between operating system and application are outnumbered by the misses induced by working set changes while one of the two runs.

Moving on to the *C-H* layout, we see that *C-H* is very successful in reducing self-interference misses in the operating system. The resulting total misses are now only 43-62% of those in *Base*. Chang and Hwu's algorithm is, therefore, effective even for the operating system. Moreover, the improvements are performed without harming the application significantly. Our *OptS* algorithm, however, does even better. The number of misses is now only 24-53% of those in *Base*. On average, this layout reduces the misses in the *C-H* layout by 25%. The reduction, as expected, occurs in the self-interference misses. Moreover, the application does not suffer any impact.

Looking at the next layout, *OptL*, we see that it performs slightly worse than *OptS* except in *Shell* where it performs slightly better. Recall that, to generate *OptL* from *OptS*, we pull the basic blocks that belong to loops out of the sequences and place them together in a contiguous area in memory. With this we hoped to squeeze the sequences into a smaller number of cache blocks and therefore enhance spatial locality and decrease the chance of cache interference within a sequence. Unfortunately, loops now cause conflicts in the cache with the sequences they were pulled out of. The combination of the good and bad effects leaves the number of misses almost unchanged or sometimes higher. We see, therefore, that effectively placing loops in the cache is a challenging task. Overall, an important part of the problem is that loops have few iterations and, therefore, loop locality is not a major effect.

Finally, focusing on the last layout, namely *OptA*, we see that optimizing the application layout achieves a further reduction of misses. For the workloads with application references, misses decrease 4-19% over *OptS*. Most of the reduction occurs in application self-interference misses, although in TRFD_4 the operating system - application cross interference decreases slightly. We have compared our algorithm to *C-H* for the applications. We find that our algorithm reduces more application misses than *C-H* in all three applications (on average over 15% more). The reason is that we preserve basic block sequences spanning several routines. In fact, for applications like *Make* or *Fsck*, the miss reductions are of the same order of magnitude as those achieved for the operating system. For the scientific applications, however, the reductions are smaller and similar to *C-H*. This is because sequences are less important and, instead, tight loops dominate. In general, however, we feel that more experimentation with applications is necessary to get a deeper insight into the issues. One important point that is clear, though, is that our optimized operating system can coexist well with an optimized application: optimizing the layout for one does not harm the other and vice-versa.

To help understand all these results, Figure 13 analyzes the operating system references and misses recorded. The leftmost chart breaks down the references depending on what type of basic block they access. We have four types of basic blocks: *MainSeq* if they belong to sequences with up to an *ExecThresh* of 0.01%; *SelfConfFree* if they belong to the SelfConfFree area; *Loops* if they belong to loops; and *OtherSeq* if they belong to the remaining sequences. Since a given basic block changes type across layouts, we chose the type that the basic block had in *OptL*. The rightmost chart decomposes the operating system misses into the same categories. As before, the bars are grouped in workloads and refer to *Base*, *C-H*, *OptS* and *OptL* (*OptA* is not interesting).

Examining the references, we see that for three workloads the MainSeq and SelfConfFree basic

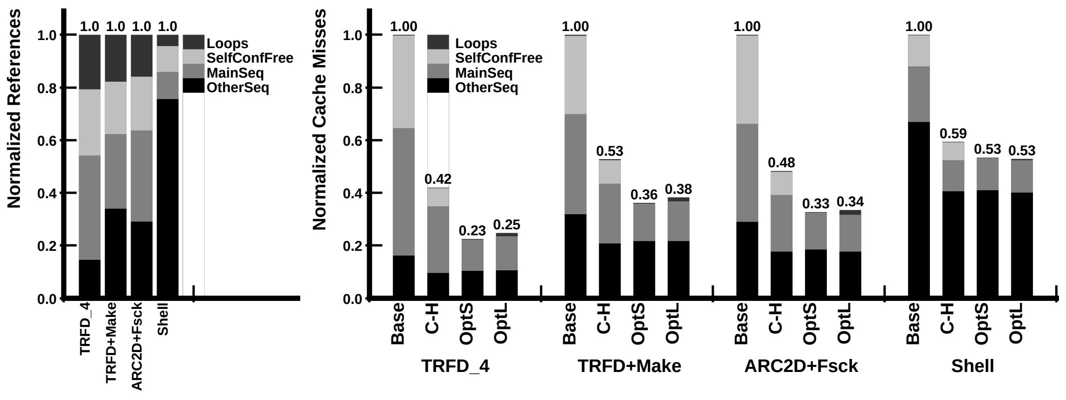


Figure 13: Classification of the operating system references (leftmost chart) and misses (rightmost chart) for different levels of layout optimization in an 8 Kbyte direct-mapped cache with 32 byte cache lines.

blocks account for 50-65% of the references. The exception is *Shell* where these basic blocks do not dominate the references. Instead, various system calls spread over OtherSeq dominate the references. We also note that, as expected, loops with 6 or more iterations are referenced little: they account for at most a bit over 20% of references. The picture, however, is even more skewed if we look at the misses. Focusing on *Base*, we see that loops cause practically no misses. Moreover, MainSeq and SelfConfFree increase their share to 67-83% (33% for *Shell*). It makes sense, therefore, for layout optimizations to focus on these basic blocks and not on loop basic blocks. Looking at the next two bars, we see that *C-H* and, specially, *OptS* do well on MainSeq and SelfConfFree basic blocks. We see that, in nearly all cases, *OptS* reduces the misses on MainSeq basic blocks beyond the number in *C-H* and eliminates the misses on SelfConfFree basic blocks. This justifies our spatial and temporal locality optimizations respectively. Both *C-H* and *OptS* also reduce some of the OtherSeq misses. Finally, looking at the *OptL* bar, we see that, in most cases, the misses in MainSeq and Loops increase over their value in *OptS*. This shows that *OptL* performs worse because there is interference between the loops and the sequences the loops were pulled out of.

Finally, before finishing this section, we show that all these optimizations indeed eliminate the most prominent conflicts in the operating system code. This can be seen in Figure 14, which shows the distribution of the operating system misses as a function of the address of the instruction that suffered the miss. The figure shows the sum of the misses of all workloads for 8-Kbyte direct-mapped caches with 32-byte cache lines. From left to right, the figure shows the misses under *Base*, *C-H*, and *OptS*. While a given routine is placed in different addresses under different layouts, for clarity purposes, in the *C-H* and *OptS* charts, the routines are plotted in the same sequence as they were in *Base*.

The figure shows that the *C-H* optimization reduces the sizes of the miss peaks in the *Base* layout. Furthermore, the *OptS* optimization further decreases the sizes of the peaks. The result is a distribution with only small peaks.

5.2 Effect of the Cache Size

Overall, the previous charts do not give us the complete picture of the impact of these optimizations. For this, we have to examine the miss rates and speedups. Figure 15-(a) shows the total miss rates for 4, 8, 16, and 32 Kbyte caches with 32 byte lines. The figure shows that the miss rate of *Base* is 0.87-6.75%. It also shows that *C-H* reduces the miss rate by, on average, 39-60%. This is a

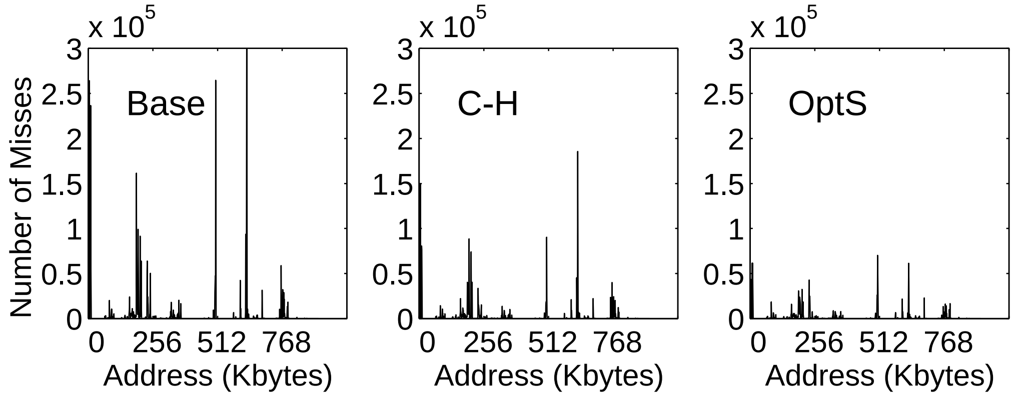


Figure 14: Distribution of the operating system misses as a function of the address of the instruction that suffered the miss. The data corresponds to the sum of the misses of all workloads for 8-Kbyte direct-mapped caches with 32-byte cache lines. From left to right, the charts correspond to *Base*, *C-H*, and *OptS*.

significant reduction. However, *OptS* further reduces the miss rate for 4-16 Kbyte caches. Indeed, it reduces the miss rate in *C-H* by an average of 19-38%. Overall, the optimized miss rate is now 0.12-3.87%. The relative gain of *OptS* over *C-H* tends to increase with larger caches until 16 Kbytes. For a cache as large as 32 Kbytes, however, the miss rate of both *OptS* and *C-H* is approximately the same. This is because the cache captures most of the operating system working set already.

To get a very rough idea of how these miss rate reductions might translate into execution speed increases, we consider a machine where references take 1 cycle, miss penalties are 10, 30, or 50 cycles, respectively, data references are 30% the number of instruction references, the data miss rate is 5%, and we neglect any slowdown due to I/O activity. A 50-cycle instruction miss penalty is comparable to that of a 2-cluster DASH [12], since in DASH the kernel resides in one cluster only. The resulting speed increases of *OptS* over the *Base* layout for the previous cache organizations are shown in Figure 15-(b). From the data, we see that, with a 30-cycle miss penalty, our scheme can produce gains in the order of 10-25%. Overall, as the miss penalty increases, the most effective cache size is the 8-Kbyte one.

5.3 Effect of the Size of the SelfConfFree Area

While the optimized layout evaluated in previous sections contained a 1-Kbyte SelfConfFree area, we did not justify this choice. In this section, we consider the impact of the size of the SelfConfFree area. We examine a layout without SelfConfFree area and three others with a SelfConfFree area that contains basic blocks that account, individually, for at least 3.0%, 2.0%, and 1.0% of the total number of executed basic blocks respectively. Obviously, the larger the execution frequency cut-off, the fewer the number of basic blocks that qualify, and the smaller the SelfConfFree area is. The sizes of the SelfConfFree areas in bytes for the different layouts are 0, 376, 1286, and 2514 respectively. In the previous sections, we had used the 2.0% cut-off, which corresponds to 1286 bytes or around 1 Kbyte.

The number of misses in the different layouts is shown in Figure 16. The figure shows the misses for three different cache sizes, namely 4-Kbytes (top row), 8-Kbytes (middle row), and 16-Kbytes (bottom row). All caches are direct-mapped and have 32-byte lines. For each workload, we show several bars, which correspond to the *Base* layout, a layout without SelfConfFree area (*None*), and layouts with different frequency cut-offs for the basic blocks in the SelfConfFree area: *3.0%*, *2.0%*,

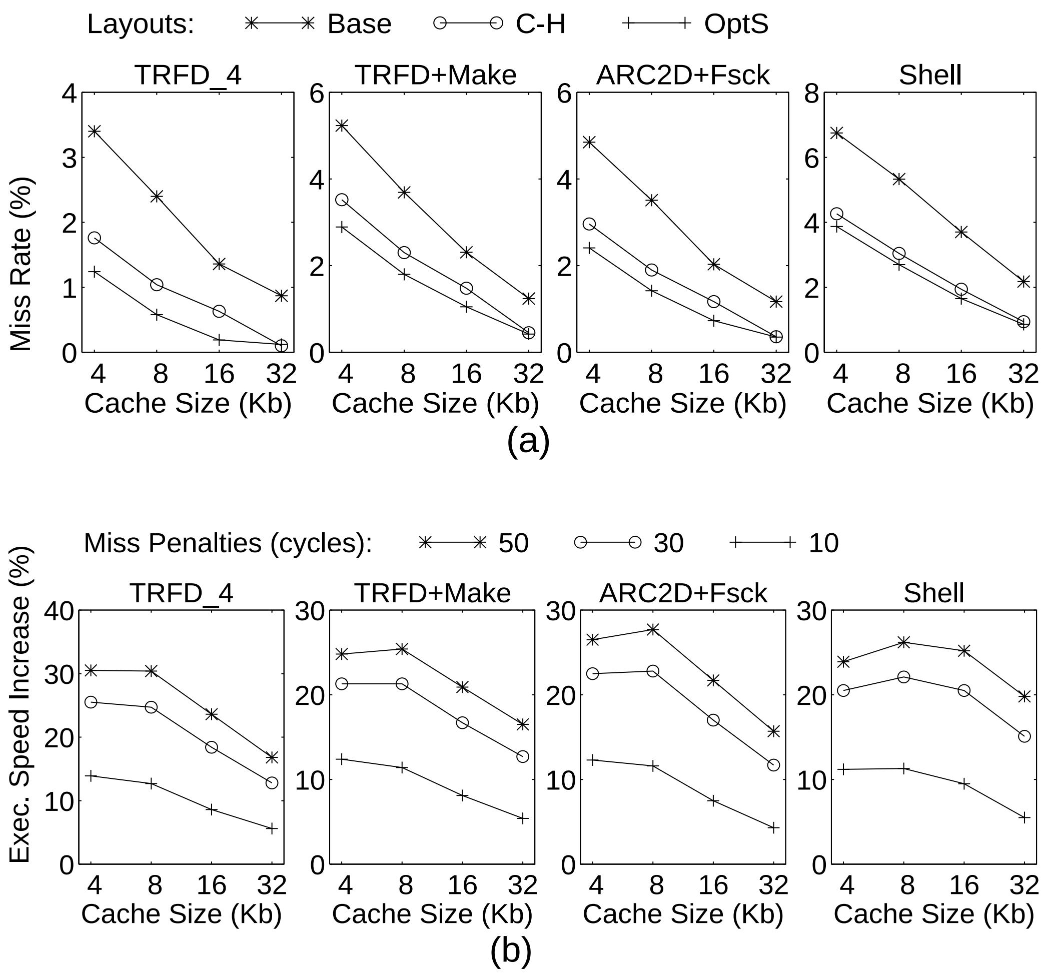


Figure 15: Total instruction miss rates (Chart (a)) and estimated execution speed increase of *OptS* over *Base* using a very simple model (Chart (b)) for different cache sizes. The line size is fixed at 32 bytes.

and *1.0%*. In each workload, the number of misses for a layout is normalized to the number of misses for the *Base* layout.

Intuitively, an increase in the size of the SelfConfFree area causes positive and negative effects. On the one hand, a larger SelfConfFree area shields more routines from operating system self-interference and, as a result, eliminates misses in these routines. On the other hand, however, less area is left for the rest of the routines, which will, therefore, suffer more conflict misses. Clearly, once the SelfConfFree area is larger than a certain value, the second effect will dominate. Unfortunately, different workloads prefer different SelfConfFree area sizes. Obviously, the SelfConfFree area will be harmful to a workload if the area contains code that is never exercised by that workload. Consequently, we would like the SelfConfFree area to contain code that is frequently accessed by all workloads. We can maximize the chance of this happening by keeping the SelfConfFree area relatively small.

From the figure, we see that the 2.0% cut-off layout outperforms or performs as well as the other layouts in over half of the experiments. This layout has about 1 Kbyte of SelfConfFree area. We also note from the figure that, for the 4-Kbyte cache, the largest SelfConfFree area (1.0%) tends to perform best, while for the 16-Kbyte cache the smallest SelfConfFree area (3.0%) tends to perform best. The reason is that, as caches get larger, there is less need for a SelfConfFree area because fewer conflicts exist. Finally, we also note that, for a given cache size, the optimal layout varies with the workload. This is a result of the different code executed by different workloads. Overall, we recommend using a 1-Kbyte SelfConfFree area for 4-16 Kbyte caches.

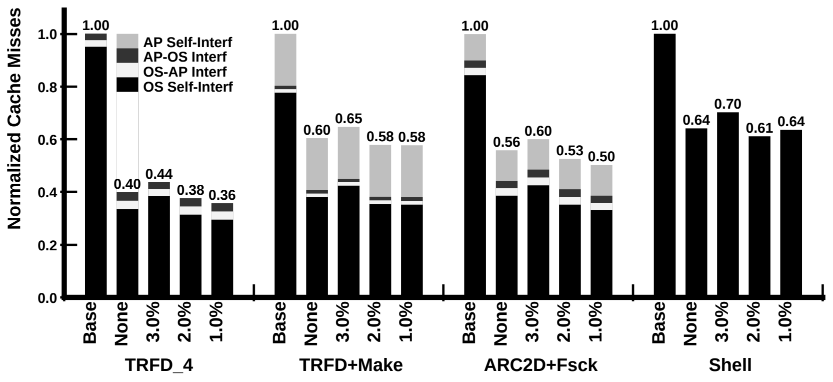
5.4 Effect of the Cache Line Size and Associativity

To show that the proposed algorithm outperforms *Base* and *C-H* in a variety of situations, we now consider variations in the line size and associativity of the caches. Figure 17 shows the miss rates for *Base*, *C-H*, and *OptS* while varying the line size of an 8 Kbyte cache from 16 to 128 bytes (Figure 17-(a)) and varying its associativity from direct-mapped to 8-way (Figure 17-(b)). Focusing on line size changes we see that, in all cases, *C-H* outperforms *Base* and *OptS* outperforms *C-H*. Furthermore, the relative gains of the optimized layouts increase as the line gets longer. For example, *OptS* reduces the miss rate by 59% on average for 16-byte lines and 70% on average for 128-byte lines. This occurs because the optimizations expose spatial locality that longer lines can exploit. In addition, the optimizations also remove part of the increasing interference caused by the availability of fewer cache lines.

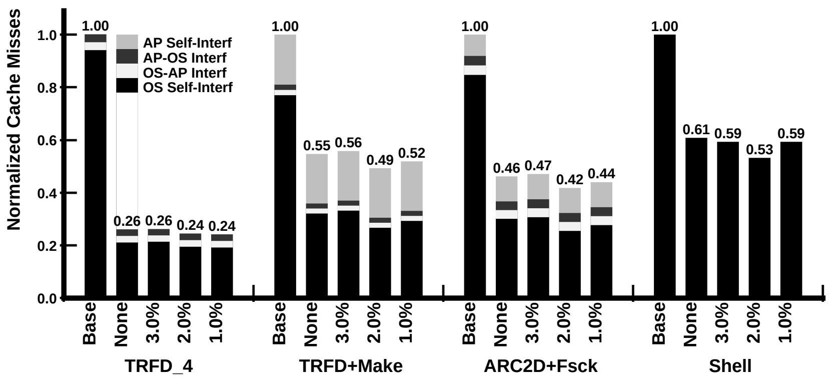
Regarding the changes in associativity, we again see that, in all cases, *C-H* outperforms *Base* and *OptS* outperforms *C-H*. As associativity increases, the relative gains of the optimized layouts are smaller. For example, *OptS* reduces the miss rate by 55% on average for a direct-mapped cache and by 41% on average for an 8-way set associative cache. This effect occurs because increased associativity removes in hardware some of the misses that our techniques eliminate in software. Note, however, that as the figure shows, the software approach is much better, even disregarding any cache speed considerations. Indeed, the miss rate for direct-mapped *OptS* is lower than for 8-way set-associative *Base*. Furthermore, there is no need to use high associativity for *OptS*: *OptS* already yields most of its benefits with direct-mapped caches.

5.5 Other Optimizations

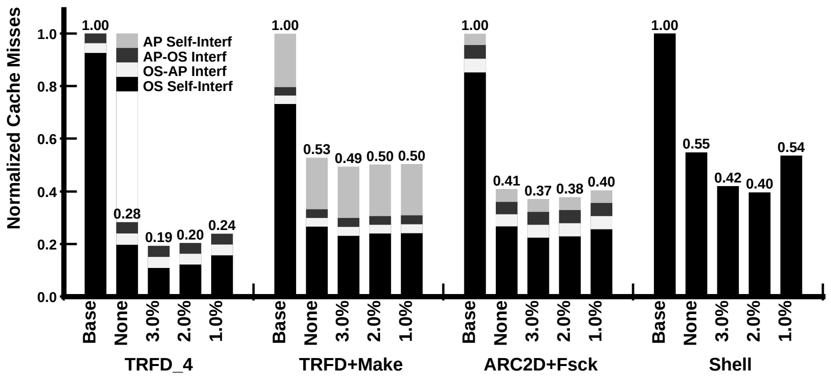
To finish the evaluation, we consider three architectural or algorithmic changes. In all experiments, we keep the total cache size equal to 8 Kbytes and the line size equal to 32 bytes. First, we examine partitioning the on-chip cache into two halves: one for the operating system and the other for the application. We apply our algorithm to both caches. Clearly, this setup is not too attractive: while it will eliminate any cross interference, it will cause more self-interference. The resulting number



4-Kbyte caches.

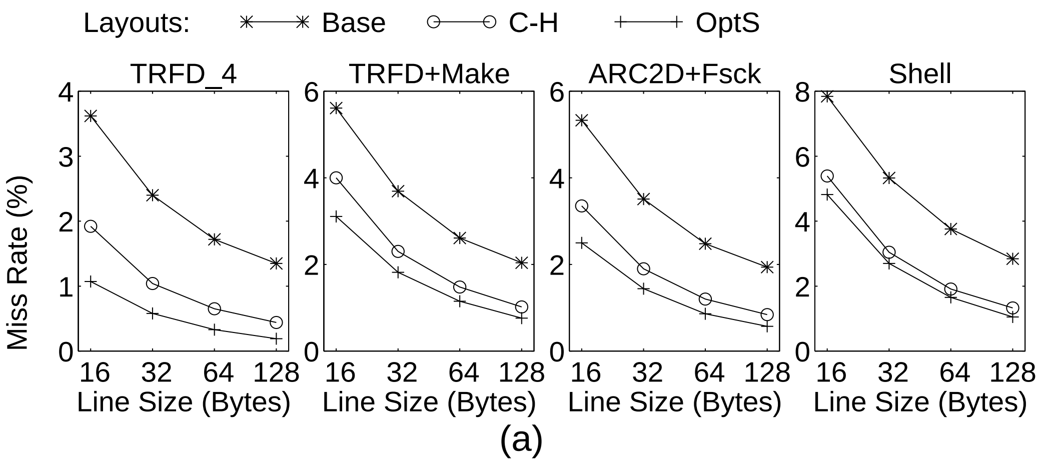


8-Kbyte caches.



16-Kbyte caches.

Figure 16: Effect of the size of the SelfConfFree area on the total number of misses. The bars correspond to the *Base* layout, a layout without SelfConfFree area (*None*), and layouts with different frequency cut-offs for the basic blocks in the SelfConfFree area: 3.0%, 2.0%, and 1.0%. All caches are direct-mapped and have 32-byte blocks.



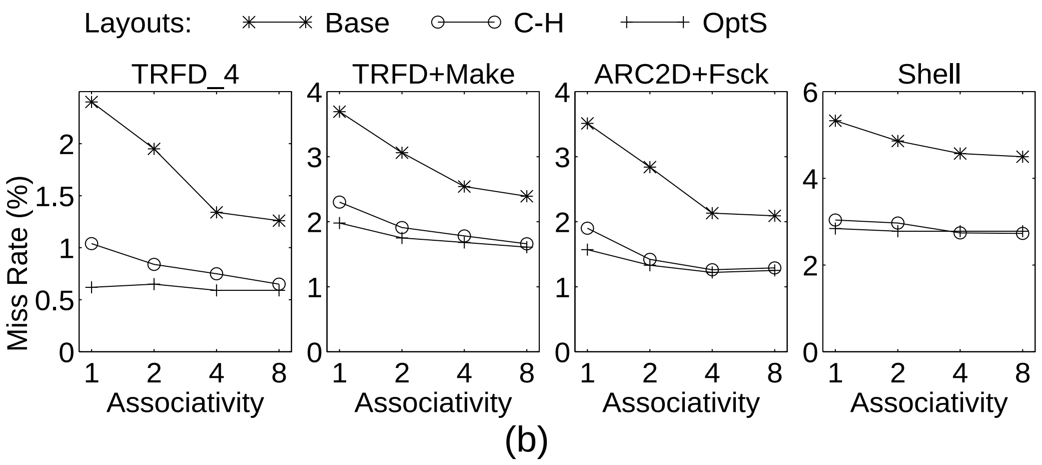


Figure 17: Cache miss rates for different line sizes (Chart (a)) and associativities (Chart (b)). The cache size is fixed at 8 Kbytes.

of misses is shown in the bars labeled *Sep* (for separate) in Figure 18. For reference purposes, the figure also shows the number of misses under *Base* (first bar in each workload) and *OptA* (second bar in each workload). The figure shows that, as expected, the *Sep* setup is not desirable. The total number of misses increases relative to *OptA* in all workloads.

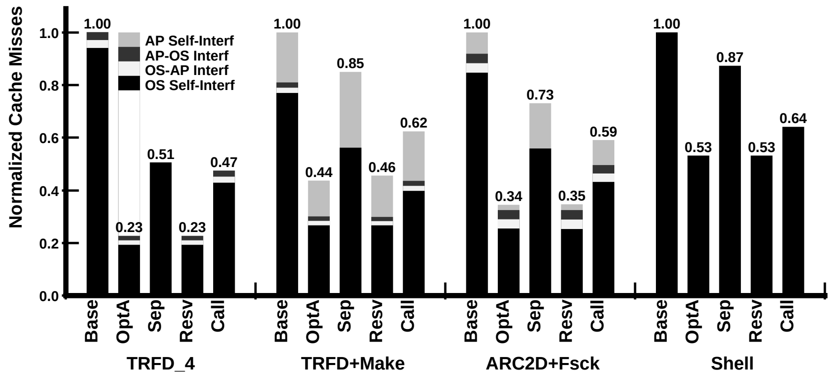


Figure 18: Normalized number of misses for different setups. In all cases, the total cache size is 8 Kbytes and the line size 32 bytes.

An alternative to the previous scheme is to provide a very small cache dedicated to the important sections of the operating system only. A similar idea has been suggested in the literature [9]. We have set up a 1 Kbyte such cache (about the size of SelfConfFree) where the most important parts of the sequences are saved. An additional 7 Kbyte cache has been made available to the application and rest of the operating system. The operating system is now laid out without SelfConfFree area. The number of misses in the new setup is shown in the *Resv* (for reserved) bars in Figure 18. The figure shows that the number of misses increases slightly over *OptA*. The operating system - application interference does not decrease much, while the application self interference increases slightly. Therefore, for these workloads, setting up a small reserved cache is not as good as cleverly laying out a SelfConfFree area in software: the performance is approximately the same and the cost is much higher.

The previous two changes target the few cross interference misses. However, we have seen that most of the remaining misses are caused by self-interference and belong to the OtherSeq category. To remove some of these misses, we have extended our algorithm to optimize the loops that call routines as described in Section 4.4.

The misses after extending our algorithm in this way are shown in the *Call* (for callees) bars of Figure 18. Focusing on the operating system misses, we see that they increase by 20-100% over *OptA*. The increase results from the loss of spatial locality caused by pulling the callee routines out of the sequences. The removal of the interference between loops and the routines they call probably saves some misses. However, the overall effect is an increase in misses. Overall, optimizing loops with the procedures they call is hard because of the many parameters involved and the small iteration counts of the loops involved. Consequently, we acknowledge the prevailing importance of exposing spatial locality and recommend not to use this optimization.

6 Conclusions

This paper addresses the problem of how to use the compiler to optimize the performance of on-chip instruction caches under operating system intensive loads. In the past, there has been evidence that the operating system often used the cache heavily and with less uniform patterns than applications.

However, it was unknown how well existing optimizations performed for the operating system and whether better optimizations could be found. Given that high instruction cache hit rates are key to high performance, this problem is worth addressing.

This paper makes two contributions. Firstly, it characterizes the locality patterns of the operating system. It is shown that there is substantial spatial locality to be exposed, a complex and hard to exploit loop locality, and much temporal locality to be exposed. To expose spatial locality and reduce conflicts within the same popular execution path, we propose the concept of sequences and show how to build them. We believe that all these conclusions can be applied to many operating systems to a large extent.

Secondly, a new code placement algorithm specifically tailored to systems code is evaluated. For a range of cache sizes, associativities, lines sizes, and other organizations we show that our scheme, *OptS*, reduces total instruction miss rates by 31-86% (up to 2.9 absolute points). Using a simple model, this corresponds to execution time reductions in the order of 10-25%. Furthermore, our scheme consistently outperforms one of the best existing algorithms by a significant amount. Finally, the effectiveness of our algorithm is not affected by the degree of application layout optimization.

7 Acknowledgments

We thank Pen-Chung Yew, Wen-Mei Hwu, the graduate students in our group and the referees for their feedback. We also thank Tom Murphy and Perry Emrath for their help with the hardware and operating system.

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] A. Aho, R. Sethi, and J. Ullman.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley, Reading, MA, 1986.

[3] 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.

[4] 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.

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

[6] P. P. Chang and W. W. Hwu.
Trace Selection for Compiling Large C Application Programs to Microcode.
In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitectures*, pages 21–29, November 1988.

[7] 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.

[8] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu.
The Effect of Code Expanding Optimizations on Instruction Cache Design.
IEEE Transactions on Computers, 42(9):1045–1057, September 1993.

[9] 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.

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

[11] W. W. Hwu and P. P. Chang.
Achieving High Instruction Cache Performance with an Optimizing Compiler.
In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 242–251, June 1989.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam.
The Stanford Dash Multiprocessor.
IEEE Computer, pages 63–79, March 1992.

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

[14] A. Mendlson, S. Pinter, and R. Shtokhamer.
Compile Time Intruction Cache Optimizations.
In *Computer Architecture News*, pages 44–51, March 1994.

[15] D. Nagle, R. Uhlig, T. Mudge, and S. Sechrest.
Optimal Allocation of On-chip Memory for Multiple-API Operating Systems.
In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 358–369, April 1994.

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

[17] 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.