Optimizing the Instruction Cache Performance of the Operating System[1]

Josep Torrellas, **Chun Xia**[2], and **Russell Daigle**[3]

Computer Science Department
University of Illinois at Urbana-Champaign, IL 61801
torrella,xia,daigle@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

Abstract

High instruction cache hit rates are key to high performance. One known technique to improve the hit rate of caches is to minimize cache interference by improving the layout of the basic blocks of the code. However, the performance impact of this technique has been reported for application code only, even though there is evidence that the operating system often uses the cache heavily and with less uniform patterns than applications. 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. Based on our observations, we propose an algorithm to expose these localities and reduce interference in the cache. For a range of cache sizes, associativities, lines sizes and organizations we show that we reduce total instruction miss rates by 31-86%, or up to 2.9 absolute points. Using a simple model, this corresponds to execution time reductions of the order of 10-25%. In addition, our optimized operating system combines well with optimized and unoptimized applications.

1 Introduction

High-performing instruction memory hierarchies are fundamental for the memory system to keep up with fast processors. To intercept instruction fetches with low latency, machines rely on on-chip instruction caches. Given that a miss may cause tens of idle cycles for the processor, these caches must be able to capture the working sets of the workloads. Fortunately, it has been shown that high instruction reference locality often helps caches intercept most of the references in scientific and engineering applications. However, many common workloads like transaction processing, compilations, or multiprogramming mixes 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 its accesses.

Indeed, there is some evidence that backs this claim. Clark [11] reported a lower performance of the VAX-11/780 cache when operating system activity was taken into account. Similarly, Agarwal *et al* [2] pointed out the many cache misses caused by the operating system. Torrellas *et al* [23] 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

[1]An earlier version of this paper was presented at the First International Symposium on High-Performance Computer Architecture in January of 1995. 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, grant 1-1-28028 from the University of Illinois Research Board and gifts from Intel and IBM.

[2]Currently with Sun Microsystems, Inc.

[3]Currently with Tandem Computers, Inc.

concentrated in relatively narrow ranges of addresses. Similarly, Chen and Bershad [8] reported that system code has lower locality than application code. They also pointed out the self-interference in the cache. Other researchers like Ousterhout [20] and Anderson *et al* [4] also indicated the different nature of the operating system activity. Finally, Nagle *et al* [19] pointed 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 instruction caches, the caching behavior of systems code needs to be understood better and improved.

Improving the performance of instruction caches has been addressed by many researchers. It has been shown that it is feasible to reduce the misses in applications by improving the layout of the code in memory [1, 12, 13, 15, 16, 17, 18, 21, 22, 24]. The techniques proposed are based on repositioning or replicating code at the procedure or basic block level, usually to reduce cache conflicts or utilize the cache lines better. In most cases, these techniques perform quite well, speeding up applications by 5-30% or even more. The schemes where the block of code that gets repositioned or replicated is the procedure [1, 24] tend to be less effective. This is because a procedure has parts that are invoked frequently and parts that are not. As a result, it is not the optimal unit to handle. Instead, the schemes that move or replicate basic blocks [12, 13, 15, 16, 18, 21, 22] are the most effective ones. Among these schemes, McFarling's technique [16] 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 [7, 15] 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 algorithms used by Pettis and Hansen [21] and Heisch [13] are similar to Hwu and Chang's. Overall, they are all focused on increasing the spatial locality of the code and, therefore, reduce cache interference. Gupta and Chi [12] focus instead on aligning loop bodies, conditional clauses or even procedures to cache line boundaries. Mendlson *et al* [18] perform code replication based on static information to eliminate conflicts. Chen *et al* [9] and McFarling [17] also examined the related issue of function inlining. Overall, function inlining seems to be largely ineffective in the presence of an optimized layout of basic blocks because the code expansion caused by inlining increases cache conflicts.

Our work differs from past work in at least two major ways. First, we report on the effectiveness of optimizing the layout of the operating system code. All the previous work has reported on the effectiveness of optimizing the layout of user applications only. Heisch [13] and Wu [24] claim that they have applied their algorithms to operating system code too. However, their papers do not provide any data on how effective their algorithms are for the operating system. The second difference of our work is that we use a detailed analysis of the reference and miss patterns of the operating system to design our basic block-based algorithms. The analysis allows us to address spatial, temporal, and loop locality.

For our experiments, we use a 4-CPU Alliant FX/8 multiprocessor running a commercial multiprocessor Unix. 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. In addition, low-iteration loops that call routines make loop locality hard to exploit. Finally, plenty of loop-less code hampers temporal locality. To expose these three localities, we design and evaluate a new code placement algorithm tailored to operating system code. For the range of cache organizations studied, we reduce total instruction miss rates by 31-86%, which corresponds to up to 2.9 absolute points. Furthermore, we compare our scheme to a previously proposed one and show that we consistently outperform it by a significant amount. The effectiveness of our algorithm is largely unaffected by the degree of placement optimization for the application.

This paper is organized in four sections: Section 2 presents the experimental setup, Section 3 analyzes 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, most of our conclusions should also be 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. The machine uses Alliant processors, which run Motorola 68020 assembly code with some extensions. We custom-designed a hardware performance monitor that gathers uninterrupted reference traces of application and operating system in real time without introducing much perturbation. The performance monitor [5] 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 on-chip 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 physical 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 trace buffers to disk. Alternatively, the data may be processed while being read from the buffers 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 relatively low 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 on-chip instruction cache. Therefore, since we want to collect all instruction accesses, the operating system and application codes must be instrumented. To do this with low perturbation, the code is augmented with machine instructions that cause data reads from specific addresses. Recall that the performance monitor can see all data accesses. The performance monitor, therefore, captures these addresses and interprets them according to an agreed-upon protocol. This methodology was suggested in [23].

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

The second type of escapes is used for application tracing. In the application program that we want to trace, we declare an array whose purpose is for generating escape references when we read its elements. However, since the performance monitor sees physical addresses only, we need to inform the trace buffer of the virtual to physical page mapping of the pages that contain the array. This is done by having the operating system inform the trace buffer (via escape accesses in the code segment) of 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 little 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. This relatively large increase is in part the result of instrumenting a non-RISC assembly code. To test the accuracy of the measurements, we carefully compared the statistics gathered with and without this instrumentation for possible skew. The statistics gathered from this kernel are close to those gathered from the minimally-instrumented kernel for the metrics that we are analyzing (for example, most frequently executed 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 from different runs, 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 workloads that involve a variety of system activity.

TRFD_4 is a mix of 4 copies of a parallelized version of the *TRFD* Perfect Club code [6]. Each program runs 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 a parallel *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 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 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-related 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 in the background. This workload executes a variety of system calls that involve context switching, scheduler activity, virtual memory management, process creation and termination and I/O- and network-related activity.

Make, *Fsck* and *Shell* are compiled with Alliant's C compiler. *TRFD* and *ARC2D* are parallelized with the Cedar Fortran compiler [14] and then by hand. Each workload is traced for about 1 minute of real time, although the process of tracing and simulating takes about one full day. For most of the experiments, we report measurements corresponding to 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 system code. We focus on locality issues. We will use this insight to design the code placement algorithms later.

3.1 Miss Patterns

Misses on system 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*. The cache line size is 32 bytes. In the figure, the misses are shown as a function of the virtual address of the code that suffers the miss. The code size of our operating system is about 940 Kbytes.

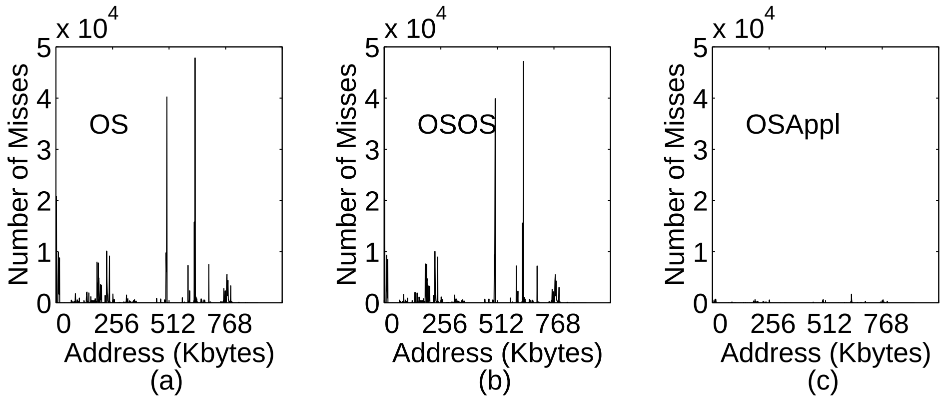


Figure 1: Number of misses on operating system code in one of the Alliant FX/8's 16-Kbyte caches as a function of the virtual address of the code. The data corresponds to *TRFD+Make*. The cache line size is 32 bytes. Charts (a), (b) and (c) show the total misses, the component caused by self-interference, 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. Each data point corresponds to the number of misses in 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 (Figure 1-(b)) and misses resulting from interference with the application (Figure 1-(c)). These charts are in agreement with those for the other workloads. They show that, for modest-sized 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.

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 only one processor. 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 only one processor. *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 an 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. From the figure, we can see the uneven distribution of the references and that large fractions of the kernel are practically not accessed.

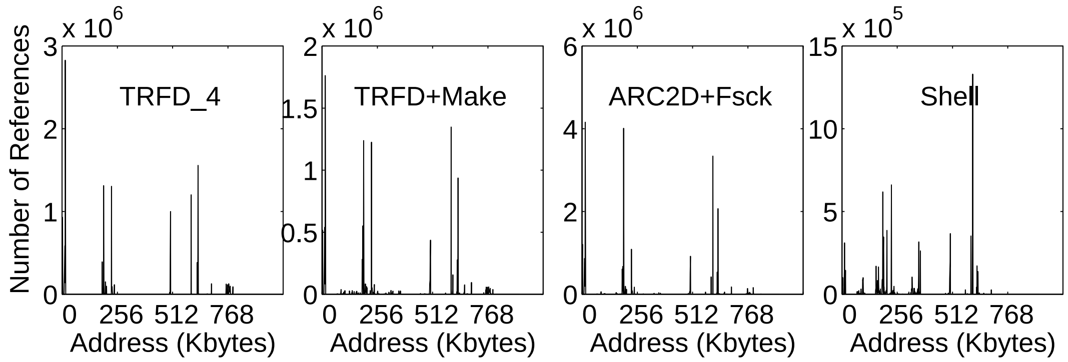


Figure 2: Number of references to operating system code as a function of the virtual address of the referenced instruction. The data corresponds to only one processor. Each data point corresponds to the references to an address range of 1 Kbyte.

We also see in Figure 2 that 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 measure the reasons why the operating system is 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.

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 a behavior similar to that of *TRFD_4*. Overall, while these four workloads are different from each other, they all tend to exercise the same 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 [8]. 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, irrespective of how far apart in the code these basic blocks are. This is because we can place these basic blocks in the same memory line 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 an *instruction sequence* or *sequence*, may span tens of routines, and 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, a significant fraction of such misses are the result of interference within the same sequence. For instance, the two highest peaks in Figure 1-(b) are caused by such an interference.

Figure 3 gives initial evidence of the determinism with which the operating system code is executed. The figure is generated out a flow graph of executed basic blocks, where each basic block is annotated with the number of times that it is executed. The arcs between basic blocks correspond to conditional and unconditional branches, basic block fall-throughs, and procedure calls and returns. Each arc between two basic blocks is annotated with the number of times that that path is taken. To generate the flow graph, we use the address traces of all the workloads. The figure shows the number of arcs leaving a basic block that have a given probability of being taken. From the figure, we see that most arcs either have a very high or a very low probability of being taken. 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 fairly deterministic too.

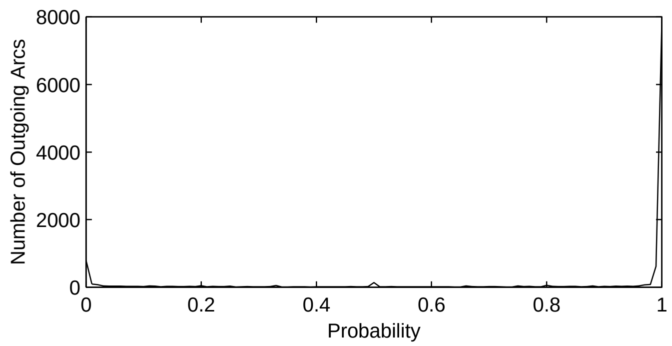


Figure 3: Distribution of the probability that an outgoing arc is executed given that the basic block that it leaves is executed.

To expose spatial locality we begin by identifying *seeds*, or basic blocks that start sets of sequences. These seeds are operating system entry points like the interrupt handler, the page fault handler and the system call handler. 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 routine, 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 execution count of the most popular basic block (*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 [7] and ours. Once we have created the sequences out of the 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 the algorithm in detail in Section 4. In this section, however, we show that the basic blocks that our algorithm selects 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 while 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 [3]. 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 only 29-39% of all operating system instructions. The equivalent number for application code is much higher. For example, it is 69% for *TRFD* and 96% for *ARC2D*. The table also shows that the static count of the instructions in these loops is around 3% of the executed instructions (Column 3) or around 0.2% of all instructions (Column 4).

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

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.

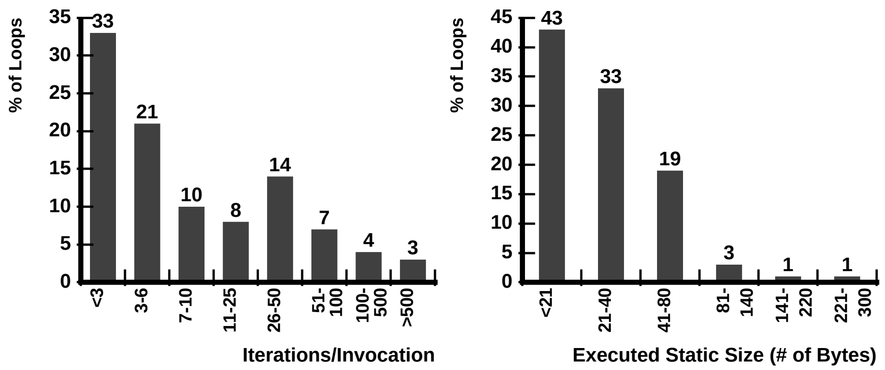


Figure 4: Behavior of the operating system loops that do not call procedures. The leftmost chart shows the distribution of the number of iterations per loop invocation, while the rightmost chart shows the distribution of the static size of the executed part of the loops.

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, it is relatively unlikely that there will be cache conflicts within individual 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 table entries 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 while 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 many of them are larger than 16 Kbytes. Therefore, it is likely that there will be cache conflicts within individual loops.

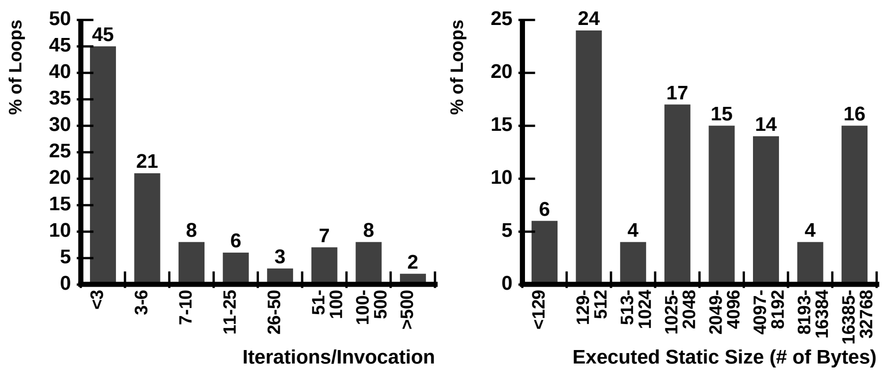


Figure 5: Behavior of the operating system loops that call procedures. The leftmost chart shows the distribution of the number of iterations per loop invocation. The rightmost chart shows the static size of the executed part of the loops including the static size of the executed part of the routines 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 may be hard for those with procedure calls. 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. The number of invocations of each (ordered) routine is shown in Figure 6. The figure is normalized so that the total number of invocations for all the routines is 100. 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, especially 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.

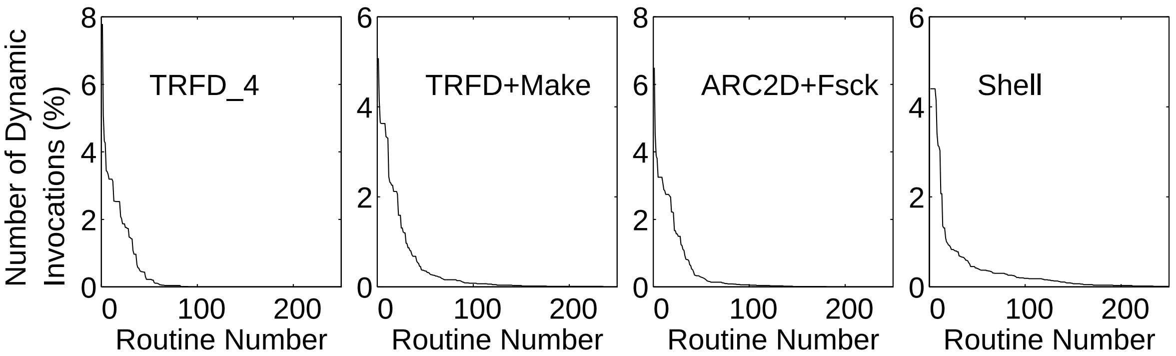


Figure 6: Number of times that each operating system routine is invoked dynamically. For a given chart, the routines are ordered from most to least frequently invoked (X axis). The charts are then normalized to a total of 100 invocations.

To determine the amount of temporal locality in the most popular routines we measure the number of 32-bit operating system instruction words fetched between two consecutive calls to a given routine. The number of instruction words is smaller than the number of instructions because some instructions are shorter than 32 bits. 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.

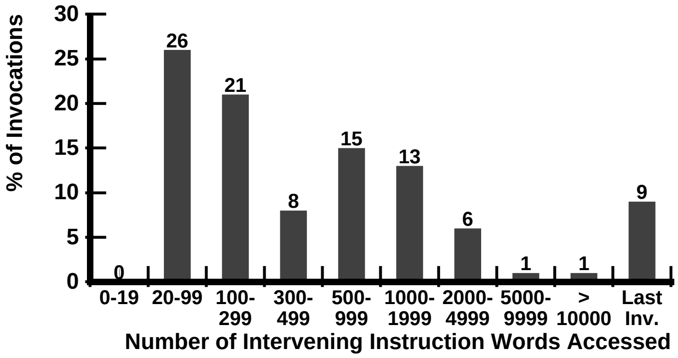


Figure 7: Understanding temporal locality. The chart shows the number of operating system instruction words referenced between two consecutive calls to the same routine in the same operating system invocation. The data corresponds to the 10 most frequently invoked routines and is the average of the four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

Unfortunately, temporal locality may be harder to exploit than these figures suggest. 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 access very different addresses instead of sitting in a tight loop. The result is that there is a good probability of displacing the popular routine from the cache before it is reused.

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 memory addresses (*SelfConfFree*). Then, we make sure that only rarely-executed code is placed in memory addresses that can conflict with the SelfConfFree area in the cache. 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, the generation of an optimized placement will be simpler. Second, there is plenty of spatial locality in sequences. Third, loops usually have few iterations and often call procedures, which makes it difficult to exploit loop locality. Finally, there is temporal locality in frequently-called small routines. With this evidence, we now proceed to design an algorithm that generates an optimized code layout.

4 Algorithm for Instruction Placement

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 execution count of a basic block divided by the execution count of the most popular basic block gives a ratio that we compare to *ExecThresh*. The weight of an outgoing edge divided by the weight of the node that 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. To compute the execution count of basic blocks that belong to loops, we assume that the loop has only one iteration per invocation. We do this because loops will be optimized independently (Section 4.3).

An example of how we apply the algorithm is shown in Figure 8. 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*. The basic blocks are numbered. For simplicity, the figure does not show the arcs for the return from subroutine and all unexecuted basic blocks are removed. In the example, we assume that basic block zero of *push_hrtime* is the only seed. Therefore, the other three routines are only called from *push_hrtime*. The number associated with each node is its weight divided by the weight of the most frequently-executed basic block. The number associated with each arc is its weight divided by the weight of the node that the arc comes from.

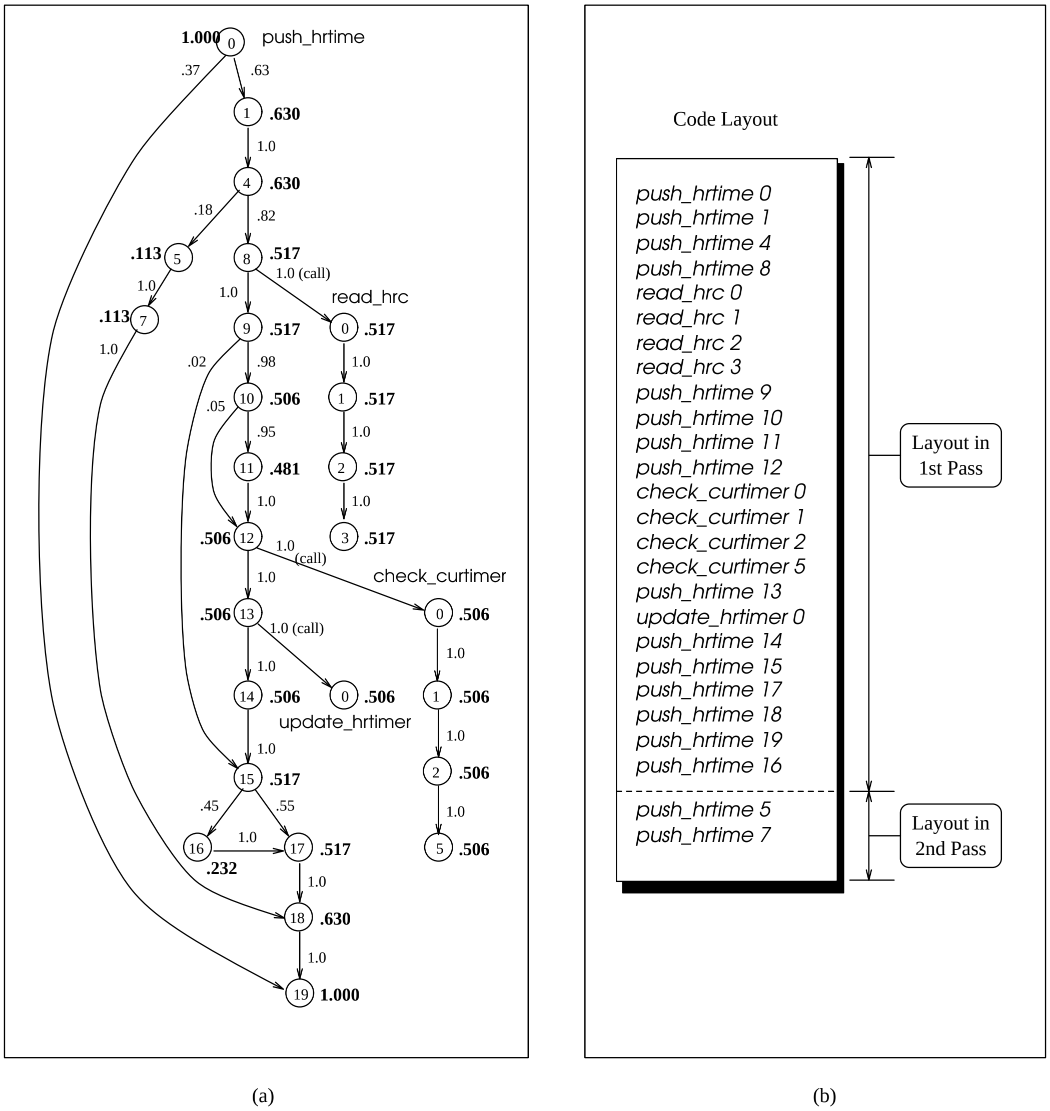


Figure 8: Placing sequences of basic blocks for a set of four routines. Chart *(a)* shows the basic block flow graph, while Chart *(b)* shows the resulting basic block layout in memory.

Chart *(b)* shows the resulting basic block layout in memory after applying our algorithm in two passes: first with (*ExecThresh*, *BranchThresh*) = (0.20, 0.1) and then with (*ExecThresh*, *BranchThresh*) = (0, 0). In the (0.20, 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 and 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 not 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 memory 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, will not 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* = 28.0% and *BranchThresh* = 40%. As shown in the table, these values give reasonably-sized sequences. Then, successive 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.

ExecThresh (%)	Seeds: Interrupt *BranchThresh* (%) # of BBs # of Bytes	Seeds: Page Fault *BranchThresh* (%) # of BBs # of Bytes	Seeds: SysCall *BranchThresh* (%) # of BBs # of Bytes	Seeds: Other *BranchThresh* (%) # of BBs # of Bytes
28.0	0.4 49 810	- - -	- - -	- - -
10.0	0.1 139 1946	0.4 28 576	- - -	- - -
2.0	0.01 79 1618	0.1 116 2146	0.4 70 1208	- - -
0.2	0.01 379 4784	0.01 235 5254	0.1 889 14274	0.4 282 4272
$2\text{x}10^{-4}$	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 also addressed by Chang-Hwu's [15] 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, then a few basic blocks more from the caller routine, and then the rest of the basic blocks of the callee routine.

A possible alternative to our scheme could be function inlining. In function inlining, the whole callee routine, not just the few frequently-executed basic blocks of it, is inserted between the caller's basic blocks. In addition, the basic blocks of the callee routine may be replicated. Function inlining, therefore, is less effective, may expand the active code size and may increase the chance of conflicts.

Indeed, while Chen *et al* [9] 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 size of the cache and which start at addresses that are multiples of the cache size (Figure 9). 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. In 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 9. The figure contains a loop area that will be described later.

As in Section 4.1, 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 again 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 the other logical caches with seldom-executed code is not difficult given that there is abundant seldom-executed code in the operating system.

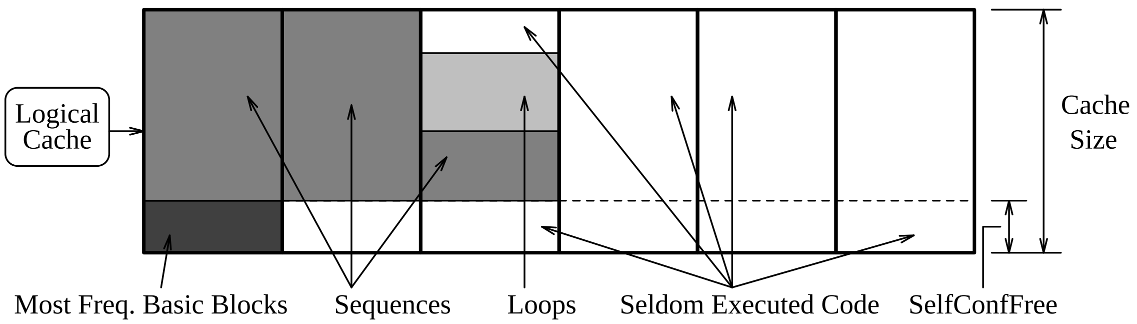


Figure 9: Optimized layout of the operating system code in memory. Addresses increase from bottom to top within a cache-sized chunk and then from left to right.

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* [3]. 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 9.

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 code size is only an average of 2.0%.

4.4 Advanced Optimizations

To optimize the loops that call routines, we have explored a more advanced optimization. Unfortunately, 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 that 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. 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 identify 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 in Figure 9. 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 layout of the basic blocks of the routines the loop calls and their callees. In this way, the loop and its callees do not interfere. Obviously, the code that we place now is pulled out of the sequences, thereby compacting the sequences further.

It is possible that two loops call the same routine. To handle this case, we need to complicate the algorithm a bit. The algorithm uses a data structure called the conflict grid. The conflict grid lists in its X-axis the loops and in its Y-axis the routines called by at least one of these loops. This grid 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 pruned from the grid. The threshold is set such that only 50 different routines are kept in the grid.

After the grid is generated, we place each loop on a different logical cache. Then, we successively pick up the next most popular routine in the grid and place it on 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. In this way, the routine does not conflict with any of the two caller loops or the other routines that they call. To illustrate the algorithm, Figure 10 shows an example memory layout with memory organized as in Figure 9. The figure corresponds to code where 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 left a gap of size equal to *r2*'s size between *L2* and *r4*.

While the algorithm described may seem intuitive, it only eliminates a few conflict misses and fails to expose much spatial locality. The main purpose of this algorithm is to reduce the number of conflicts between a loop and the routines that it calls. The impact of this optimization, however, is small because these loops tend to have few 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 little spatial locality. Overall, therefore, we do not implement this optimization in our algorithm. We evaluate it separately in Section 5.5.

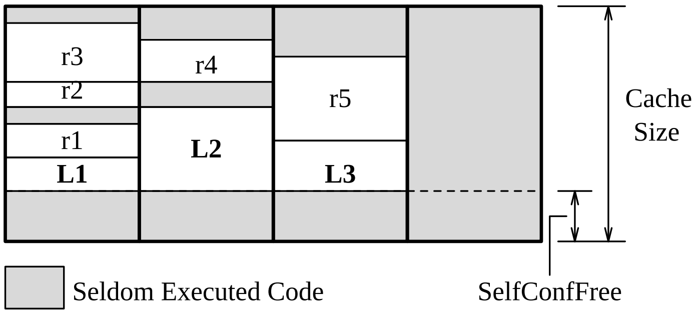


Figure 10: Optimizing the placement of loops that call routines.

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 Unix layout; *C-H* refers to the layout generated by Chang-Hwu's algorithm [15]; *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 optimizing the layout of the application with sequences and the simple loop optimization described in Section 4.3. For the operating system, we set the SelfConfFree area to 1 Kbyte (Section 5.3), while for the applications, we do not set up any SelfConfFree area because the behavior can vary widely across applications. For the applications, 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 layout 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 Layout Optimization

The effect of the different level of layout optimization on the number of misses is shown in Figure 11. 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, operating system misses caused by interference with the application, application misses caused by interference with the operating system and application misses caused by self-interference. Cold misses are negligible.

Examining the references, we see that, for three workloads, the operating system accounts for 40-60% of the instruction references. For *Shell*, all the references shown belong to the operating system. As intended, these are system-intensive workloads. 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 largely the result of using loop-intensive scientific applications: both *TRFD* and *ARC2D* have

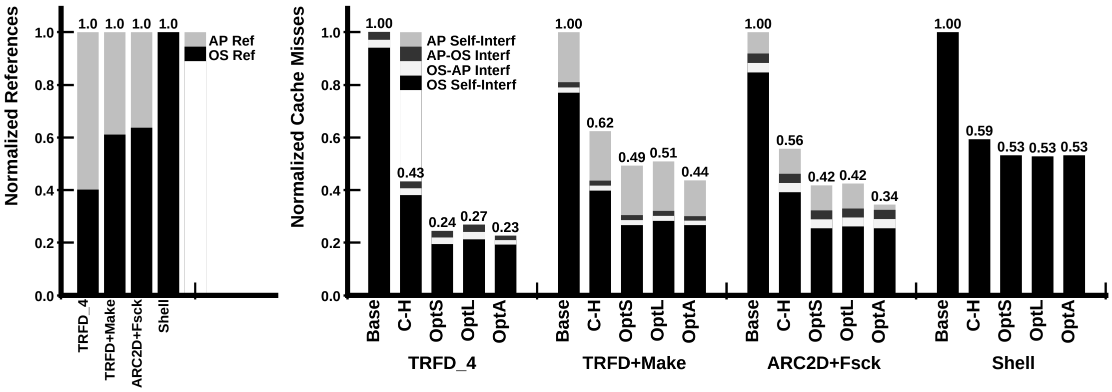


Figure 11: Normalized number of instruction references (leftmost chart) and misses (rightmost chart) for different levels of layout optimization in an 8-Kbyte direct-mapped cache with 32-byte cache lines.

a small instruction miss rate. We also note that the majority of misses are the result of self-interference as opposed to cross-interference. This is because, in the small caches that we examine, the misses in the transition between application and operating system are outnumbered by the misses induced by working set changes while the operating system or while the application run.

Moving on to the *C-H* layout, we see that *C-H* is successful in removing many 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 with *OptS* is 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 on average slightly worse than *OptS*. 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. Unfortunately, loops now conflict 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 placing loops in the cache effectively is a challenging task. 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 further reduces the number of misses. For the workloads with application references, misses decrease an average of 11% relative to *OptS*. Most of the reduction occurs in application self-interference misses. We have compared our algorithm to *C-H* for the applications. We find that our algorithm removes an average of 15% more application misses than *C-H*. The reason is that our algorithm generates basic block sequences spanning several routines. For applications like *Make* and *Fsck*, the miss reductions yielded by our algorithm are similar to those achieved for the operating system. For *TRFD* and *ARC2D*, however, the reductions are smaller and similar to those yielded by *C-H*. This is because sequences are less important and, instead, tight loops dominate. Overall, more experimentation with applications is necessary to gain a deeper insight. One important point that is clear, though, is that our optimized operating system can coexist well with unoptimized and optimized applications.

To help understand all these results, Figure 12 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: *SelfConfFree* if they belong to the SelfConfFree area; *Loops* if they belong to loops with 6 or more iterations; *MainSeq* if they belong to sequences with an *ExecThresh* no lower than 0.20%; 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).

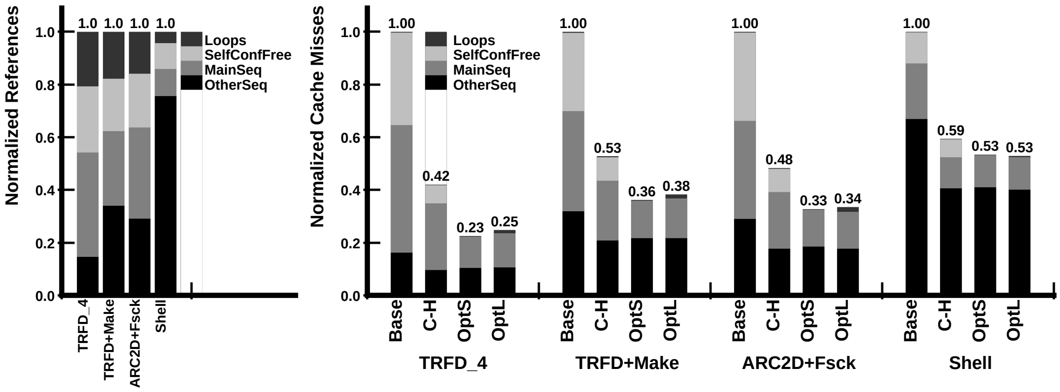


Figure 12: Classification of the operating system references (leftmost chart) and misses (rightmost chart) for different levels of layout optimization in an 8-Kbyte direct-mapped cache with 32-byte cache lines.

Examining the references, we see that, for three workloads, the MainSeq and SelfConfFree basic 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 from the *Loops* category that, as expected, loops with 6 or more iterations are referenced little. 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, especially, *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 13, 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 basic block is placed in different addresses under different layouts, for clarity purposes, in the *C-H* and *OptS* charts, the basic blocks are plotted in the same sequence as they were in *Base*. The figure shows that the peaks of misses decrease from *Base* to *C-H* to *OptS*. In *OptS*, the peaks are all very small.

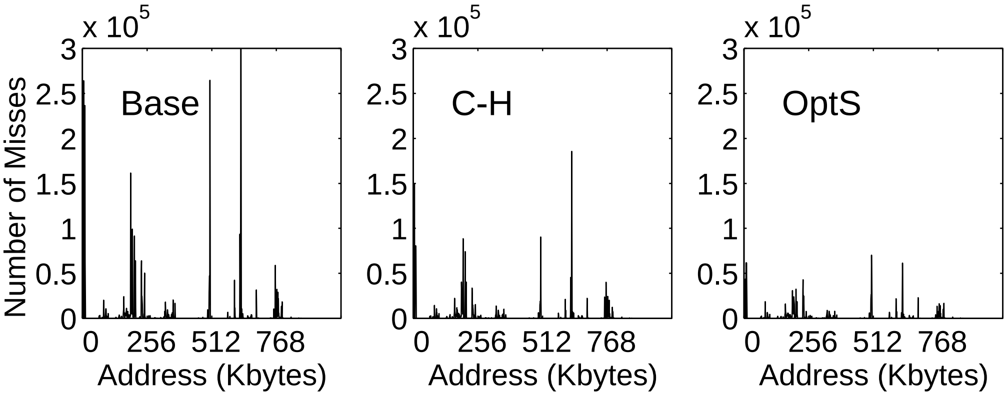


Figure 13: Distribution of the operating system misses as a function of the address of the instruction that suffered the miss. The data corresponds to the sum of the misses of all workloads for 8-Kbyte direct-mapped caches with 32-byte cache lines. From left to right, the charts correspond to *Base*, *C-H* and *OptS*.

5.2 Effect of the Cache Size

To gain a better insight into the impact of these optimizations, we examine the miss rates and speedups. Figure 14-(a) shows the total miss rates for a range of cache sizes for 32-byte cache 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 an average of 40-60%. This is a significant reduction. *OptS* further reduces the miss rate for caches of up to 16 Kbytes. It reduces the miss rate in *C-H* by an average of 20-40%. Overall, the optimized miss rate is now 0.12-3.87%. The relative gain of *OptS* over *C-H* is the largest for 16-Kbyte caches. For 32-Kbyte caches, the miss rates of both *OptS* and *C-H* are approximately the same. This is because the cache already captures most of the operating system working set. For workloads that additionally exercise other parts of the operating system or for newer, bloated operating system codes that have larger working sets, it is possible that *OptS* continues to outperform *C-H* for larger caches.

To get a very rough idea of how these miss rate reductions might translate into execution speed increases, we consider three single-issue machines where the miss penalties are 10, 30 and 50 cycles, respectively, data references are 30% of the number of instruction references, and the data miss rate is 5%. The resulting speed increases of *OptS* over the *Base* layout for the previous cache organizations are shown in Figure 14-(b). From the data, we see that, with a 30-cycle miss penalty, our scheme can produce speed increases of the order of 10-25%.

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. In addition, we examine three layouts where the SelfConfFree area contains basic blocks whose ExecThresh is at least 0.60, 0.40 and 0.20 respectively. Obviously, a high ExecThresh means that few basic blocks qualify and that the SelfConfFree area is small. 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 0.40 cut-off, which corresponds to the SelfConfFree area of about 1 Kbyte.

The number of misses in the different layouts is shown in Figure 15. The figure shows the misses

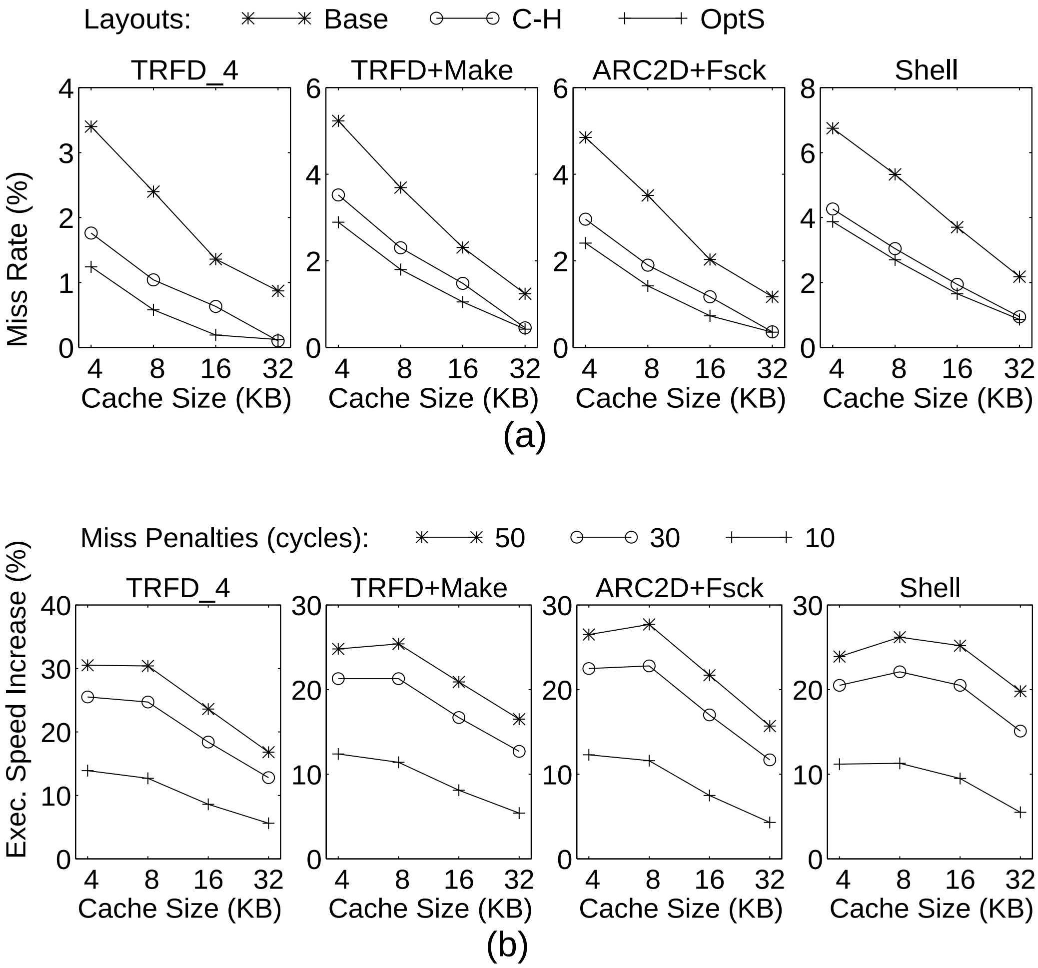
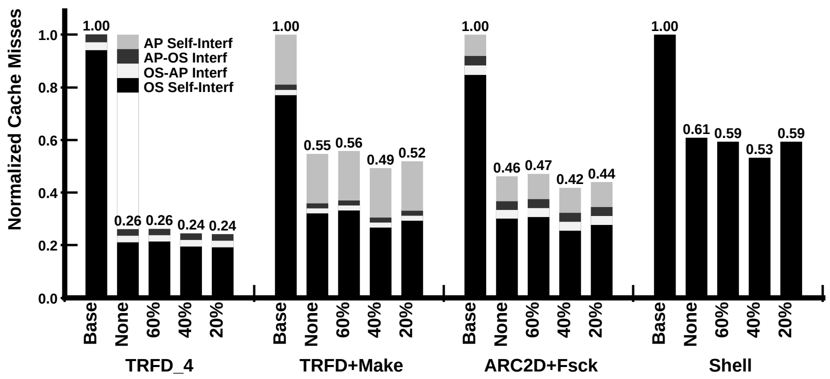
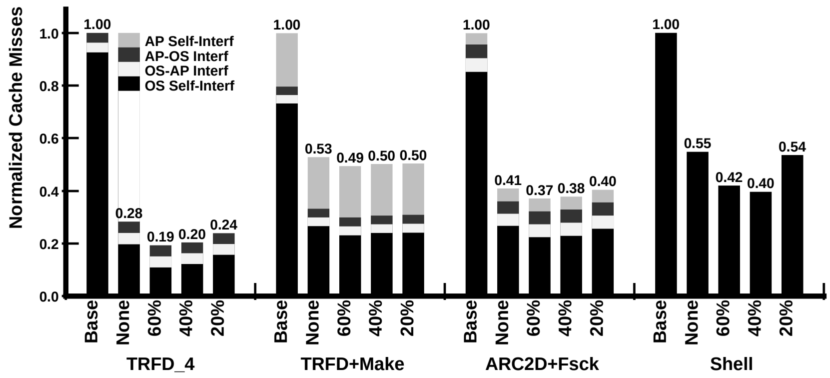


Figure 14: Total instruction miss rates (Chart (a)) and estimated execution speed increase of *OptS* over *Base* using a very simple model (Chart (b)) for different cache sizes. The line size is fixed at 32 bytes.

for two different cache sizes, namely 8-Kbytes (top 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, an optimized layout without SelfConfFree area (*None*) and optimized layouts with different ExecThresh cut-offs for the basic blocks in the SelfConfFree area: 60%, 40% and 20%. In each workload, the number of misses for a layout is normalized to the number of misses for *Base*.



8-Kbyte caches.



16-Kbyte caches.

Figure 15: 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 ExecThresh cut-offs for the basic blocks in the SelfConfFree area: 60%, 40% and 20%. All caches are direct-mapped and have 32-byte blocks.

Intuitively, an increase in the size of the SelfConfFree area induces 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. Unfortunately, different workloads prefer different sizes.

From the figure, we see that the 40% cut-off layout outperforms or performs as well as the other layouts in over half of the experiments. As indicated before, this layout has about 1 Kbyte of SelfConfFree area. We also note from the figure that, as the cache gets larger, the size of the best SelfConfFree area tends to decrease. The reason is that, as caches get larger, fewer conflicts exist. Overall, we recommend using a 1-Kbyte SelfConfFree area for these cache sizes.

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 16 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 16-(a)) and varying its associativity from direct-mapped to 8-way (Figure 16-(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.

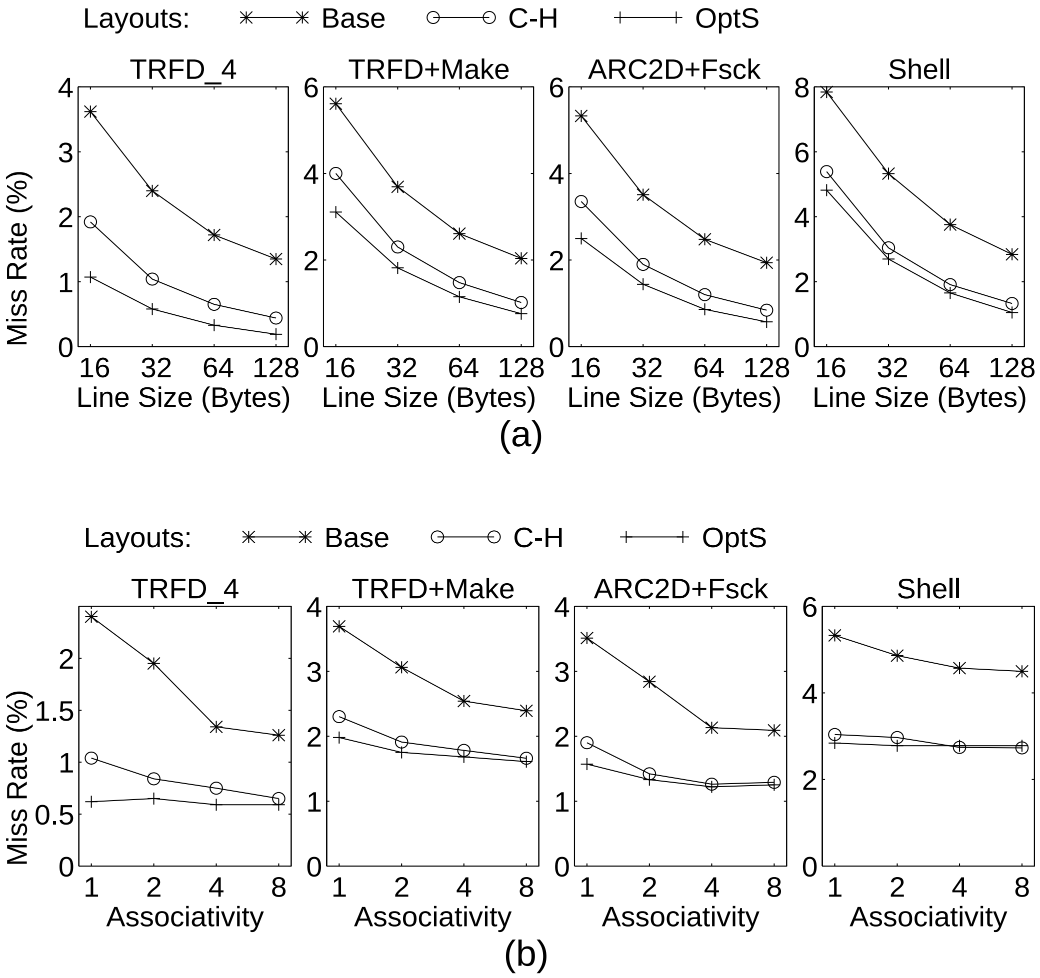


Figure 16: Cache miss rates for different line sizes (Chart (a)) and associativities (Chart (b)). The cache size is fixed at 8 Kbytes.

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 decrease. For example, *OptS* reduces the miss rate by an average of 55% for a direct-mapped cache and by an average of 41% for an 8-way set associative cache. This effect occurs because increased associativity eliminates, in hardware, some of the misses that our technique eliminates 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. This setup eliminates all cross interference between operating system and application at the expense of causing more self-interference in the operating system and in the application. However, this setup is undesirable because previous sections have shown that the majority of misses come from self-interference, not from cross interference between operating system and application. The resulting number of misses is shown in the bars labeled *Sep* (for separate) in Figure 17. 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 good. The total number of misses increases relative to *OptA* in all workloads.

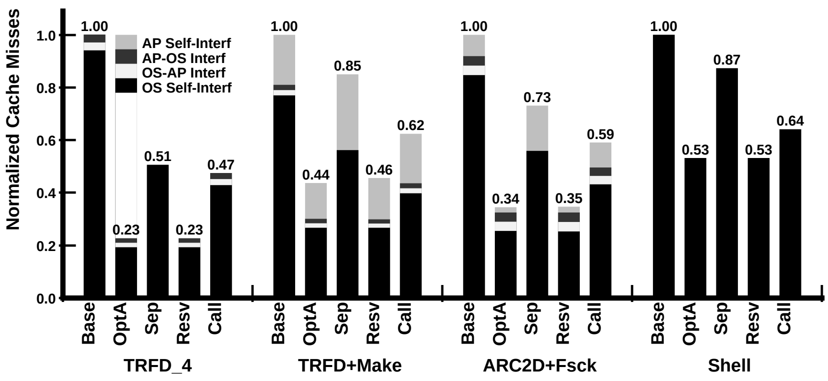


Figure 17: Normalized number of misses for different setups. In all cases, the total cache size is 8 Kbytes and the line size 32 bytes.

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 for the VMP multiprocessor [10]. We have set up a 1 Kbyte such cache (about the size of SelfConfFree) where the most important parts of the sequences are saved. A 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 total number of misses in the new setup is shown in the *Resv* (for reserved) bars in Figure 17. 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 that we have. However, we have seen that most of the remaining misses are caused by self-interference and belong to the OtherSeq category (Section 5.1). 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 17. Focusing on the operating system misses, we see that they increase by 20-100% over *OptA*. The increase results from the lower spatial locality and the higher interference caused by pulling the callee routines out of the sequences. Overall, optimizing loops with the procedures that they call is hard because of the many parameters involved and the small iteration count of the loops involved. Consequently, we acknowledge the prevailing importance of exposing spatial locality and recommend not to use this loop optimization.

6 Conclusions

This paper addressed 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 had been evidence that the operating system often used the cache heavily and with less uniform patterns than applications. However, it was little known 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 was worth addressing.

This paper made two contributions. Firstly, it characterized the locality patterns of the operating system. It was shown that there is substantial spatial locality to be exposed, a complex and hard to exploit loop locality, and temporal locality to be exposed. To expose spatial locality and reduce conflicts within the same popular execution path, we proposed the concept of sequences and showed how to build them. We believe that our conclusions are applicable to other operating systems and, possibly, to other types of systems codes.

Secondly, a new code placement algorithm specifically tailored to systems code was proposed and evaluated. For a range of cache sizes, associativities, lines sizes, and other organizations we showed that our scheme, *OptS*, reduced total instruction miss rates by 31-86%, which corresponds to up to 2.9 absolute points. Using a simple model, this corresponds to execution time reductions of the order of 10-25%. Furthermore, our scheme consistently outperformed an existing algorithm by a significant amount. Finally, the effectiveness of our algorithm was not affected by the degree of application layout optimization.

7 Acknowledgments

We thank the referees, Pen-Chung Yew, Wen-Mei Hwu and the graduate students in the I-ACOMA group for their feedback. We also thank Tom Murphy, Perry Emrath and Liuxi Yang for their help with the hardware and operating system. We also thank Intel and IBM for their generous support. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] A. Agarwal, P. Chow, M. Horowitz, J. Acken, A. Salz, and J. Hennessy.
On-Chip Caches for High-Performance Processors.
In *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference*, pages 1–24, March 1987.

[2] 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.

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

[4] 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.

[5] 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.

[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] 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.

[8] 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.

[9] 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.

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

[12] R. Gupta and C.-H. Chi.
Improving Instruction Cache Behavior by Reducing Cache Pollution.
In *Proceedings of Supercomputing 1990*, pages 82–91, November 1990.

[13] R. R. Heisch.
Trace-Directed Program Restructuring for AIX Executables.
In *IBM Jornal of Research and Development*, pages 595–603, September 1994.

[14] J. Hoeflinger.
Cedar Fortran Programmer's Handbook.
Technical Report 1157, Center for Supercomputing Research and Development, October 1991.

[15] 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.

[16] 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.

[17] S. McFarling.
Procedure Merging with Instruction Caches.
In *Proceedings of the SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 71–79, June 1991.

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

[19] 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.

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

[21] K. Pettis and R. C. Hansen.
Profile Guided Code Positioning.
In *Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[22] A. D. Samples and P. N. Hilfinger.
Code Reorganization for Instruction Caches.
Technical Report CSD-88-447, University of California, Berkeley, October 1988.

[23] 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.

[24] Y. Wu.
Ordering Functions for Improving Memory Reference Locality in a Shared Memory Multiprocessor System.
In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 218–221, December 1992.