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

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

Center for Supercomputing Research and Development
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 12-26%. 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 [9] 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* [16] 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 ranges of addresses and attribute them to the relative lack of loops in the code. Similarly, Chen and Bershad [6] report that systems code has lower locality than application code. They also point out the self-interference in the cache. Other researchers like Ousterhout [14] and Anderson *et al* [2] also indicate the different nature of the operating system activity. Finally, Nagle *et al* [13] 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 careful page placement [10] or via improved code layout in the cache [7, 11, 12]. The former technique is not relevant because the operating system text is usually not paged. The latter technique is based on repositioning or replicating code, usually to reduce cache conflicts. McFarling's technique [11] 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 [5, 7] 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 [7] but find it largely ineffective because code expansion increases cache conflicts. Finally, Mendlson *et al* [12] perform code replication based on static information to eliminate conflicts. In all 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 an 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

[1]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.

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 optimizations and, finally, Section 5 evaluates them.

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 [3] has one probe connected to each of the four processors. The probes collect all the references issued by the processors except those that hit in the per-processor 16 Kbyte 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. More details are presented in [15].

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 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. The performance monitor can capture the addresses read from and interpret them according to an agreed-upon protocol. This methodology was suggested in [16].

To distinguish these *escape accesses* from real accesses, we do as follows. One first type of escapes, used for operating system instrumentation, access 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 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 accessing 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 references 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 application [15].

Using this approach, 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 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 call 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 [4]. Each program is run with 4 processes. The code is predominately composed of matrix multiplies and data interchanges. It is highly parallel

[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.

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.

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

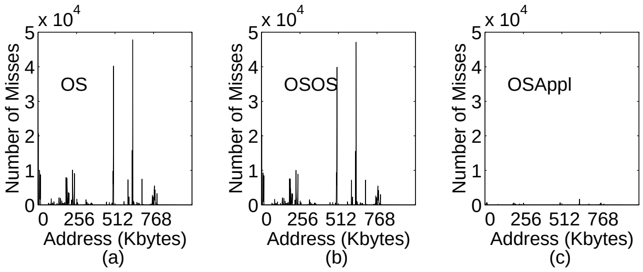

Figure 1: Number of misses on operating system code in one of the Alliant FX/8's 16 Kbyte caches as a function of the virtual address of the code. The data corresponds to *TRFD+Make*. Charts (a), (b) and (c) show the total misses, the component caused by self-interference, and the component caused by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

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. 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. These same 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

The first important characteristic of the operating system is that large sections of its code are rarely accessed. The reason is that the operations performed by the operating system include many infrequent special cases that need to be handled for correctness. This effect is shown in Figure 2, which plots the number of references to operating system instructions as a function of the virtual address of the referenced instruction. From the figure, we can see that many sections are not accessed. The size of the code accessed in each of the workloads is a small fraction of the approximately 1 Mbyte kernel. In fact, combining all workloads, only 18% of the operating system code is ever referenced. Furthermore, only 26% of the routines in the operating system are ever invoked.

Figure 2 also shows the uneven distribution of the references. Some areas are much more popular than others. However, the figure also indicates that there is some consistency across workloads. Overall, the significance of these observations is that most of the code has little impact on cache performance and, therefore, we have more freedom to perform optimizations for the frequently executed routines.

To optimize the cache performance, we simply need to perform transformations that expose the locality in the ar-

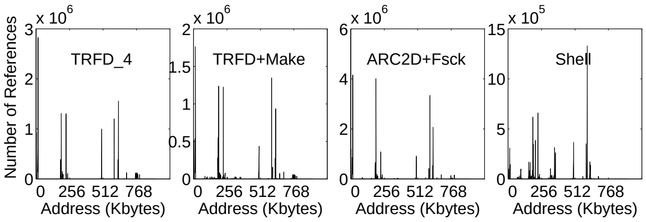

Figure 2: Number of references to operating system code as a function of the virtual address of the referenced instruction. Each data point corresponds to the references to an address range of 1 Kbyte.

eas 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 has 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 in the operating system and most of the misses 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.

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 assembler. We identify four of them: the starting point of interrupt handling, page fault handling, system call servicing, and context switching. Then, starting from the seed, 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 edge out of the basic block with the highest weight. 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 most popular seed's execution count (*ExecThresh*), or all the outgoing edges have less than a certain probability of being taken (*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 [5] 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 are 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 1). 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 1 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 from a core sequence. Furthermore, Column 3 shows that it has probability 0.71-0.77 of being followed by the execution of the next 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. 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 2: 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 3, which shows a distribution of these loops according

Table 1: Characteristics of the sequences of basic blocks. Recall that regular sequences are a superset of core sequences.

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: 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

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.

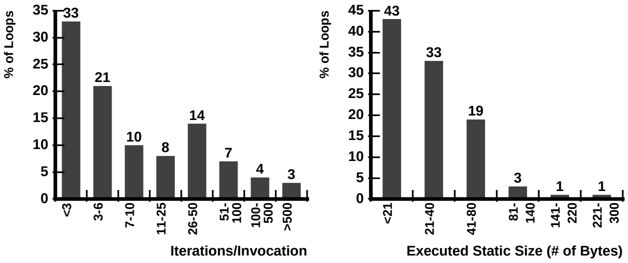

Figure 3: Behavior of the operating system loops that do not call procedures. The leftmost chart shows the distribution of the number of iterations per loop invocation, while the rightmost chart shows the distribution of the static size of the executed part of the loops.

Finally, the static size of the executed part of these loops is small. For example, as shown in the rightmost chart of Figure 3, 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 4 characterizes the 71 such loops that we isolated. They include inner and outer loops. 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.

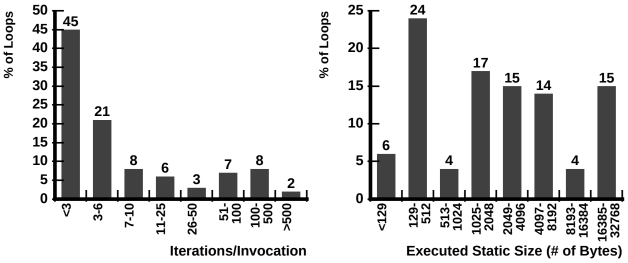

Figure 4: Behavior of the operating system loops that call procedures. The leftmost chart shows the distribution of the number of iterations per loop invocation. The rightmost chart shows the static size of the executed part of the loops including the static size of the executed part of the routines 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.

3.2.3 Temporal Locality in Routines

The final type of locality that we consider is temporal locality in routine invocations. 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 5 shows the normalized value of such count. The figure shows that there is a large skew in the frequency of routine invocations. 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. All of these routines appear in the sequences identified above.

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 op-

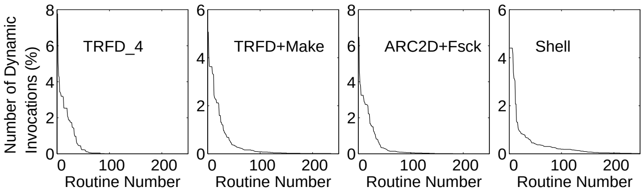

Figure 5: Number of times each operating system routine is invoked dynamically. For a given chart, the routines are ordered from most to least frequently executed. The charts are then normalized to 100 invocations.

erating 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 6 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.

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.

To exploit temporal locality, we ensure that these frequently called routines remain in the cache. This is done by assigning the frequently executed basic blocks of these routines to a contiguous range of addresses (*SelfConfFree*). Then, we make sure that only rarely-executed code is placed in addresses that conflict with the SelfConfFree area. Since there is plenty of rarely-executed code, this is feasible. In practice, we extract the SelfConfFree basic blocks from the most popular sequences described above.

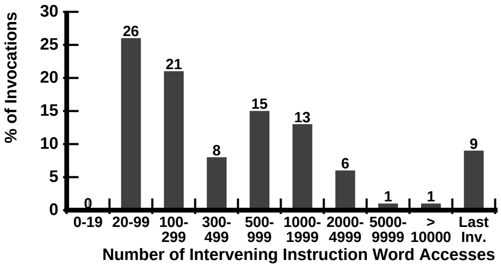

Figure 6: Understanding temporal locality. The chart shows the number of operating system instruction words referenced between two consecutive calls to the same routine in the same operating system invocation. The data corresponds to the 10 most frequently invoked routines and is the average of the four workloads.

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.

4 Instruction Placement Algorithm

Our algorithm uses the experimental evidence discussed above to expose the three types of localities and, as a result, minimize operating system self-interference misses.

We start with the sequences of basic blocks. We repeatedly select a pair of values for *ExecThresh* and *BranchThresh*, generate the resulting sequences, and place them in the cache. In each iteration, we lower the values of *ExecThresh* and *BranchThresh*, therefore capturing more and more rarely-executed segments of code. The code is therefore placed in the cache in segments of decreasing frequency of execution. This effect minimizes the impact of self-interference, since popular sequences will be placed close to other equally popular ones, and therefore cannot conflict with them. In [15], we show how the set of values for *ExecThresh* and *BranchThresh* is selected. Basically, the first set is chosen based on gaps in the curves that plot the distribution of execution frequencies for basic blocks and outgoing edge probabilities for basic blocks respectively. The chosen values are $ExecThresh = 1.4\%$ and $BranchThresh = 40\%$. Then, successive iterations use values for these two parameters that decrease one order of magnitude every time [15]. This is repeated until all operating system code is selected.

We do not simply place all these sequences consecutively. Instead, we leave some empty gaps between some of them so that the first *SelfConfFree* bytes placed do not interfere with any other executed part of the operating system (Figure 7). These gaps are filled with non-executed code. This way, while all sequences exploit spatial locality, the most popular *SelfConfFree* bytes in the sequences will additionally exploit temporal locality. After some experiments [15], we chose the SelfConfFree area to be about 1 Kbyte. Filling the gaps with non-executed code is easy, since the gaps are small, and there is abundant non-executed code.

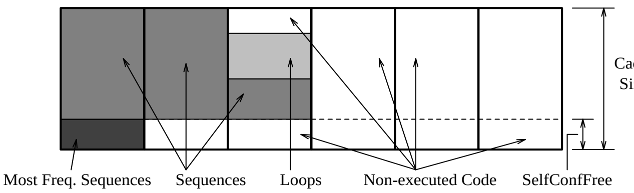

Figure 7: Optimized layout of the code in memory. Addresses increase from bottom to top within a cache-sized chunk and then from left to right.

Finally, to exploit loop locality, we identify all the basic blocks in loops with at least a minimum number of iterations per invocation (currently set to 6). We then remove all these basic blocks from 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. Since the basic blocks extracted belong to a loop, if the loop does not call other procedures, the basic blocks will still suffer few misses: at most one miss in the first iteration of the loop. While we have investigated performing optimizations on the procedures called by loops, we do not perform any; we have

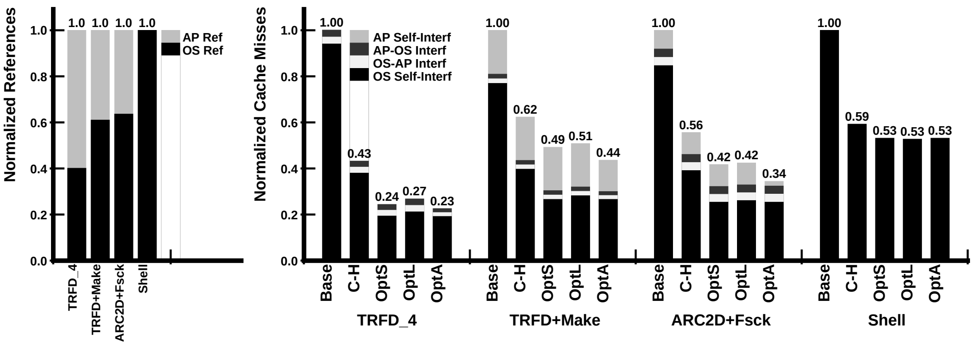

Figure 8: Normalized number of references (leftmost chart) and misses (rightmost chart) for different levels of layout optimization in an 8 Kbyte direct-mapped cache with 32 byte cache lines.

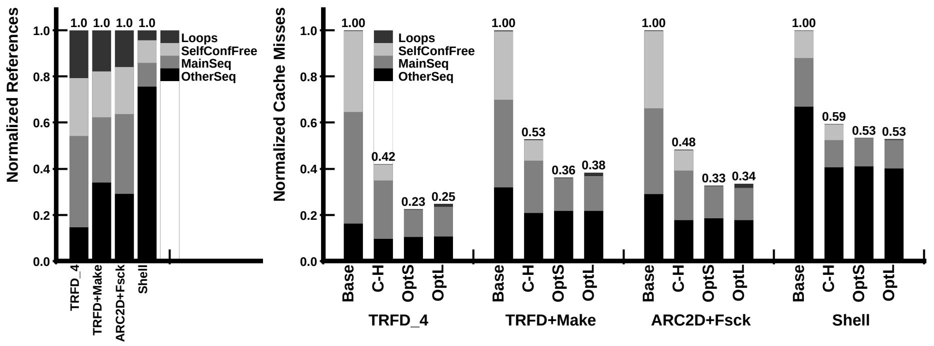

Figure 9: Classification of the operating system references (leftmost chart) and misses (rightmost chart) for different levels of layout optimization in an 8 Kbyte direct-mapped cache with 32 byte cache lines.

found that the added complexity does not pay off.

Overall, the code increases in size because of the extra branches that we have to add when we do basic block motion. However, the increase in dynamic size is as low as 2.0% on average and, therefore, the impact is very small. The algorithm is further described in [15].

5 Evaluation

In this section we summarize the performance impact of our optimization 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 [7] algorithm; *OptS* refers to our layout with SelfConfFree area, sequences, and no loop optimization; *OptL* is *OptS* plus the loop optimization; finally, *OptA* is *OptS* plus the layout of the application optimized with sequences and loops. 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 significantly affected because the number of application references caused 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, 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 8. 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.

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 strip from the code the basic blocks that belong to loops and place them together in an area of the cache. With this we hoped to squeeze the sequences into a smaller number of cache blocks and therefore decrease the chance of sequence interference. In addition, the basic blocks in loops would enjoy loop locality even outside sequences. Unfortunately, loops now cause conflicts in the cache with the code they were extracted from. 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. It needs to be examined further in order to reduce more misses. 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. 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 these results, Figure 9 analyzes the operating system references and misses recorded in this setting. 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* and *Loops* if they belong to the SelfConfFree area or to loops respectively; 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 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 and SelfConfFree basic blocks beyond the number in *C-H*. 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 next bar, *OptL*, we see that it is slightly higher than *OptS*.

5.2 Effect of the Cache Size

Overall, the previous charts do not give us the real impact of these optimizations. For this, we have to examine the miss rates and speedups. Figure 10-(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* monotonically reduces the miss rate by, on average, 39-60%. This is a 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.

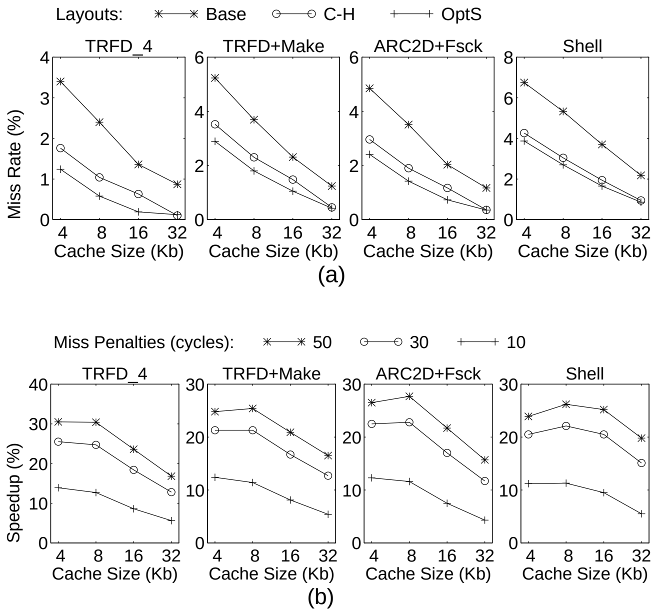

Figure 10: Total instruction miss rates (Chart (a)) and estimated execution time speedups of *OptS* over *Base* using a very simple model (Chart (b)) for different cache sizes. The line size is fixed at 32 bytes.

To get a very rough idea of how these miss rate reductions might translate into execution time speedups, we consider a machine where references take 1 cycle, misses take 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, where the kernel resides in one cluster only. The resulting speedups of *OptS* over the *Base* layout for the previous cache organizations are shown in Figure 10-(b). From the data, we see that our scheme can produce gains in the order of 12-26% with a 30-cycle miss penalty.

5.3 Effect of the Cache Line Size and Associativity

To show that the proposed algorithm outperforms *Base* and *C-H* in a variety of settings, we now consider variations in the line size and associativity of the caches. Figure 11 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 11-(a)) and varying its associativity from direct-mapped to 8-way (Figure 11-(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.

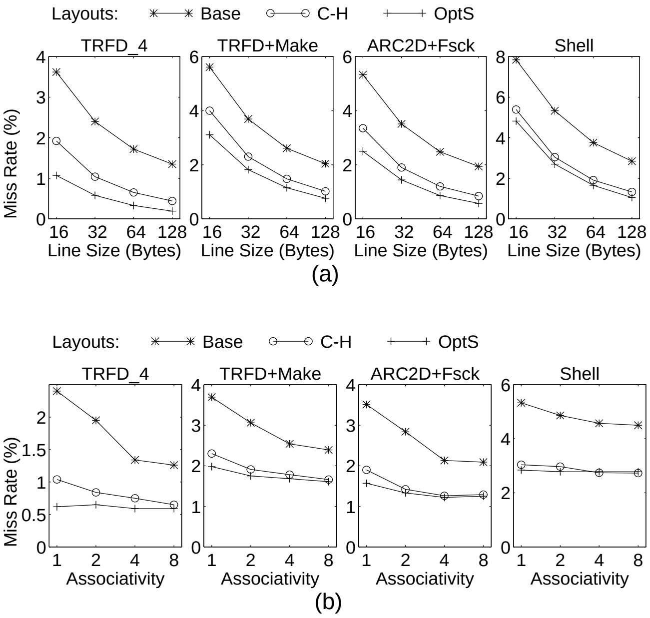

Figure 11: Cache miss rates for different line sizes (Chart (a)) and associativities (Chart (b)). The cache size is fixed at 8 Kbytes.

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 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 decrease in miss rate for *Base* saturates at a high miss rate. This is a result of the small size of the caches considered. Furthermore, the figure also shows that the optimized layouts already achieve most of their potential for direct-mapped caches.

5.4 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 of misses is shown in the bars labeled *Sep* (for separate) in Figure 12. 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.

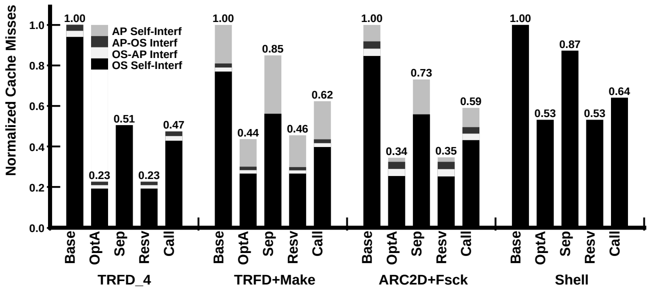

Figure 12: Normalized number of misses for different setups. In all cases, the total cache size is 8 Kbytes and the line size 32 bytes.

An alternative to the previous scheme is to provide a very small cache dedicated to the important sections of the operating system only. Similar approaches have been suggested in the literature [8]. 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 12. 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, setting up a small reserved cache is not as good as cleverly laying out a SelfConfFree area in software.

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. Many of them are actually caused by routines called by loops. To remove these misses, we have extended our algorithm to identify, for each loop, all the routines it calls and their descendants. Then, we assign each loop to one logical cache. Logical caches are memory regions of size equal to the cache that start at addresses that are multiples of the cache size. The loop is placed at address SelfConfFree bytes past the beginning of its logical cache. The loop is then followed by the frequently-executed basic blocks of the routines it calls and their callees. This way, the loop and its callees do not interfere, and there should not be any misses past the first iteration. Unfortunately, two loops may call the same routine. In this case, we place the routine in the logical cache of one of the loops and place non-executed code in the corresponding addresses of the other loop's logical cache. To reduce the likelihood of this problem, we prune out rarely-called routines. More details are given in [15].

The number of misses with this algorithm is shown in the *Call* (for callees) bars of Figure 12. Focusing on the operating system misses, we see that they increase by 20-100% over *OptA*. The increase results from the loss in spatial locality caused by the uprooting of callee routines from sequences. Overall, optimizing loops with the procedures they call is hard because of the many parameters involved. We are studying the issue further. At this point, however, we acknowledge the prevailing importance of exposing spatial locality and recommend the use of the *OptS* optimization only.

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 a large amount of 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 12-26%. Furthermore, our scheme consistently outperforms one of the best existing algorithms by a significant amount. In addition, the effectiveness of our algorithm is not affected by the degree of application layout optimization.

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

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

[13] 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.

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

[15] J. Torrellas, R Daigle, and C. Xia. Optimizing Instruction Cache Performance for Operating System Workloads. Technical Report 1387, Center for Supercomputing Research and Development, November 1994.

[16] 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.