Instruction Prefetching of Systems Codes With Layout Optimized for Reduced Cache Misses[1]

Chun Xia[2] and **Josep Torrellas**

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

Abstract

High-performing on-chip instruction caches are crucial to keep fast processors busy. Unfortunately, while on-chip caches are usually successful at intercepting instruction fetches in loop-intensive engineering codes, they are less able to do so in large systems codes. To improve the performance of the latter codes, the compiler can be used to lay out the code in memory for reduced cache conflicts. Interestingly, such an operation leaves the code in a state that can be exploited by a new type of instruction prefetching: guarded sequential prefetching.

The idea is that the compiler leaves hints in the code as to how the code was laid out. Then, at run time, the prefetching hardware detects these hints and uses them to prefetch more effectively. This scheme can be implemented very cheaply: one bit encoded in control transfer instructions and a prefetch module that requires minor extensions to existing next-line sequential prefetchers. Furthermore, the scheme can be turned off and on at run time with the toggling of a bit in the TLB. The scheme is evaluated with simulations using complete traces from a 4-processor machine. Overall, for 16-Kbyte primary instruction caches, guarded sequential prefetching removes, on average, 66% of the instruction misses remaining in an operating system with an optimized layout, speeding up the operating system by 10%. Moreover, the scheme is more cost-effective and robust than existing sequential prefetching techniques.

1 Introduction

High-performing instruction memory hierarchies are essential to keep current processors highly utilized. Much of the effectiveness of the instruction memory hierarchy depends on the success of on-chip caches at intercepting instruction fetches issued by the processor. Fortunately, it is well-known that instruction references in typical loop-intensive engineering codes have high locality. As a result, on-chip caches are usually able to capture the instruction working set of these codes and keep their instruction misses to a minimum.

Unfortunately, it has recently been shown that on-chip caches do not perform as well for large systems codes [8, 12, 13]. Examples of such codes are operating systems, windows managers, databases, and multimedia codes. These codes have large working sets and, in addition, exhibit lower locality than loop-intensive engineering codes. Since these applications are used frequently in real life, it is important to understand and improve their cache performance.

The first step to improve the instruction cache performance of these codes is to optimize the layout of their instructions in memory [12]. The purpose of this step is to expose more locality in the code and, as a result, minimize conflicts in the cache. The approach taken usually involves building the basic block graph of the code and then, based on profile information, carefully placing the basic blocks in memory. For example, basic blocks that are usually fetched in sequence are laid out in sequence, while basic blocks that form a loop are placed to avoid any conflicts within the loop. Overall, it has been shown that these schemes work well, both for engineering [3, 7] and for systems codes [12]. They work particularly well for systems codes because the original unoptimized layout has poor performance.

After this optimization, the misses that remain tend to be spread out in the code in a uniform manner; there are no obvious hot spot areas of conflict misses. Consequently, no simple changes in the layout algorithm are likely to produce further significant gains. Instead, to hide these spread-out misses, we can try some form of prefetching.

Given that there are only a few misses to be removed, however, any cost-effective prefetching scheme has to be very cheap. Previously proposed schemes, like next-line prefetch on reference or on miss, are indeed cheap. However, they are not as cost-effective as they could be. This is because they do not exploit any information available to the compiler when it generated the code layout. Ignoring this information may not have much impact in loop-intensive engineering codes, but it may not be affordable in codes with a more complex control flow structure like systems codes.

In this paper, we propose a new technique whereby the compiler leaves encoded hints in the code as to how the code was laid out. Then, at run time, the prefetching hardware detects these hints and uses them to optimize sequential prefetching for low misses and traffic. This scheme we call *Guarded Sequential Prefetching*. We show that it can be implemented very cheaply: one bit encoded in control transfer instructions and a prefetch module that requires minor extensions to existing next-line sequential prefetchers. In addition, the scheme can be turned off and on at run time with the toggling of a bit in each TLB entry. In this paper, we use a multiprocessor UNIX with an optimized layout and simulate guarded sequential prefetching. Overall, for 16-Kbyte direct-mapped instruction caches, guarded sequential prefetching removes, on average, 66% of the instruction misses remaining in the optimized layout, speeding up the operating system by 10%. Furthermore, the scheme is more cost-effective and robust than existing sequential prefetching techniques.

This paper is organized as follows: Section 2 discusses the experimental setup used; Section 3 reviews the algorithm

[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; ARPA Contract No. DABT63-95-C-0097; and NASA Contract No. NAG-1-613.

[2]Currently with Sun Microsystems, Inc.

used to optimize the layout of the code; Section 4 discusses the miss patterns in codes with optimized layouts; Section 5 presents our technique to prefetch these misses; Section 6 evaluates the scheme via simulations, and Section 7 presents related work.

2 Experimental Setup

This work is based on the analysis of address traces from a multiprocessor. We use a multiprocessor to capture a larger range of systems activity, including multiprocessor scheduling and cross-processor interrupts. In this section, we discuss the hardware and software setup used and the workloads traced. More details on the setup and workload characteristics can be found in [12, 14].

2.1 Hardware and Software Setup

We gather the traces from a 4-processor bus-based Alliant FX/8 multiprocessor. The operating system running in the machine is a slightly modified version of Alliant's Concentrix 3.0. Concentrix is multithreaded, symmetric, and is based on Unix BSD 4.2. We use a hardware performance monitor that gathers uninterrupted traces in real time for both operating system and applications. The performance monitor has one probe connected to each of the four processors. The probes collect all the references issued by the processors except those that hit in the per-processor 16-Kbyte primary instruction caches. Each probe has a trace buffer that stores over one million references. When one of the four trace buffers nears filling, it sends a non-maskable interrupt to all processors. Upon receiving the interrupt, processors halt in less than ten machine instructions. Then, a workstation connected to the performance monitor dumps the buffers to disk. 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.

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

This instrumentation increases the dynamic size of the code by 30.1% on average. To make sure that we are not perturbing the execution of the operating system noticeably, we compared the relative frequency of invocation of the operating system routines with and without this instrumentation. We find that there is no noticeable difference. In particular, there is no increase in page faulting activity.

2.2 Workloads

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

TRFD_4 is a mix of 4 runs of a hand-parallelized version of the *TRFD* Perfect Club code [2]. Each program runs with 4 processes. The code is composed of matrix multiplies and data interchanges. It is highly parallel yet synchronization intensive. The most important operating system activities present are page fault handling, process scheduling, cross-processor interrupts, processor synchronization, and other multiprocessor management functions. The operating system runs for 42.1% of the time.

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 forces frequent changes of regime in the machine and cross processor interrupts. There is also substantial memory management activity. The operating system runs for 53.6% of the time.

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 that runs with 4 processes. It causes operating system activity like that caused by *TRFD*. *Fsck* is a file system consistency check and repair utility. We run it on one whole file system. It contains a wider variety of I/O code than *Make*. The operating system runs for 45.8% of the time.

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/termination, and I/O- and network-related activity. The operating system runs for 47.0% of the time.

3 Optimizing Code Layouts

In this section, we describe our approach to optimizing the layouts of large systems codes. We first discuss the general approach and then present the algorithm used. More information can be found in [12].

3.1 General Approach

Our approach is to lay out the code to expose the three localities in it: spatial, temporal, and loop locality. If these localities are exposed, cache conflict misses should decrease. To expose spatial locality, we start by observing that address traces of operating system instruction references often contain regular repeatable sequences of basic blocks. A given set of such basic blocks, which we call an *instruction sequence*, may span several routines, and it is not part of any obvious loop. Instruction sequences are the result of complex operating system functions entailing series of fairly deterministic operations with little loop activity. Examples of such functions are the first stages of handling a page fault, processing an interrupt, or servicing a system call. A large fraction of the references and misses in the operating system occur in a set of clearly defined and frequently executed instruction sequences.

Unfortunately, the basic blocks in an instruction sequence are rarely laid out in a contiguous manner in memory. Instead, they are mixed up with other, seldom executed basic blocks. These seldom executed basic blocks appear all over the code because the operating system has to handle all possible, even if highly infrequent, situations. These seldom executed basic blocks are bypassed by conditional branches that are almost always taken. Consequently, the operating system code has low spatial locality. To expose spatial locality, we use code profiling to identify these deterministic sequences of basic blocks. Then, we perform basic block reordering and place these basic blocks in a contiguous area in memory. This step often involves surrounding some of the basic blocks of a callee routine by the basic blocks of its caller routine.

To address temporal locality, we examine the frequency of routine invocation and find that a few routines are invoked much more frequently than the rest. These routines tend to 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. The basic blocks of these routines form the most frequently executed parts of the instruction sequences described before. Unfortunately, temporal locality goes unexploited. Indeed, between two consecutive invocations of one of these popular routines, the operating system tends to execute much loop-less code, displacing the popular routine from the cache before it is reused.

To exploit temporal locality, we ensure that the most popular basic blocks of these routines are not displaced from the cache while the operating system is running. This is done by extracting these sets of basic blocks from the instruction sequences and assigning them to a contiguous range of memory locations called the *SelfConfFree* area. The SelfConfFree area does not conflict with other parts of the operating system code because we place only seldom-executed operating system code in addresses that can conflict with the SelfConfFree area in the cache. Since there is plenty of seldom-executed code, this is feasible. Note that, although this change reduces the spatial locality in the instruction sequences, it is fine because the basic blocks extracted from the instruction sequences will not suffer any conflict misses.

There is also temporal locality resulting from the multi-entry property of the operating system. Indeed, some of the instruction sequences are frequently invoked from several different operating system entry points. To minimize cache conflicts, we place instruction sequences in memory ordered by their frequency of invocation. As a result, a frequently-invoked instruction sequence does not conflict with another frequently-invoked one.

Finally, to assess the potential impact of loop locality, we use dataflow analysis [1] to identify the loops in the code. We divide the loops into those that do not call routines and those that do. The loops that do not call routines have a small weight: they account for only 29-39% of all dynamic instructions in the operating system compared to 69% in *TRFD* and 96% in *ARC2D*. Furthermore, they tend to execute few iterations per invocation, usually 6 or fewer.

The loops that call routines execute complex operations, often spanning several routines. For instance, a loop is executed when the memory allocated by a process has to be freed up after the process dies. The operating system loops over all the process' page table entries, performing many checks and complex operations in each case. These loops also have very few iterations per invocation. Their size, however, including the size of the routines that they call, is huge.

Overall, we would like to layout each loop in the cache so that no two instructions in the loop or the routines it calls conflict with each other. If this is possible, misses will be limited to the first iteration of the loop. Such a layout can be devised by pulling the basic blocks of these loops and the routines they call out of the instruction sequences and putting them, in the same order, in a contiguous area in memory. Unfortunately, since loops do not matter much, we find that, in most cases, the loss of spatial locality suffered in the instruction sequences outweighs the benefits of reducing the misses in the loops. Therefore, we do not make any change to expose loop locality.

3.2 Code Placement Algorithm

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. 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.

To expose spatial locality, we begin the code placement by identifying *seeds*, or basic blocks that start sets of instruction sequences. These seeds are operating system entry points like the interrupt handler, page fault handler and system call handler. Then, starting from the seeds, we use a greedy algorithm to build the instruction sequences. Given a basic block, the algorithm follows the most frequently executed path out of it. This implies visiting the routine called by the basic block or, if there is no callee, following the control transfer out of the basic block with the highest probability of being used. We stop when all the successor basic blocks have already been visited, or they have an execution count smaller than a given fraction of the execution count of the most popular basic block (*ExecThresh*), or all the outgoing arcs have less than a certain probability of being used (*BranchThresh*). When we have to stop, we start again from the seed looking for the next acceptable basic block.

In the algorithm, we have a loop that repeatedly selects a pair of values for *ExecThresh* and *BranchThresh*, generates the several resulting instruction sequences for each of the seeds, and places them in 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 for all the seeds. The overall result is that we place the code in memory in segments of decreasing frequency of execution. As indicated above, this preserves the temporal locality in instruction sequences and, therefore, minimizes cache conflicts.

Finally, to exploit temporal locality in the most popular routines, we consider logical caches. Logical caches are memory regions of size equal to the cache that start at addresses that are multiples of the cache size (Figure 1). In the lowest *SelfConfFree* bytes of each logical cache except the first one, we place seldom-executed code. In the SelfConfFree area of the first logical cache, we place the most frequently executed basic blocks (Section 3.1). We use a SelfConfFree area of about 1 Kbyte [12]. The resulting layout is shown in Figure 1.

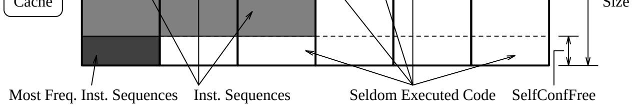


Figure 1: Optimized layout of the code in memory. Addresses increase from bottom to top within a logical cache and then from left to right.

This algorithm improves both the placement of the basic blocks within routines and the relative placement of the routines. These two aspects are already addressed by Hwu and Chang's algorithm [4]. However, our algorithm further exposes spatial locality by generating instruction sequences that cross routine boundaries. For example, an instruction sequence may contain a few basic blocks of the caller routine, then the most important basic blocks of the callee routine, and then a few more basic blocks from the caller routine. This is different from function inlining. In function inlining, the whole callee routine is inserted between the caller's basic blocks, not just the callee's most important basic blocks. As a result, function inlining, even if it is done at a single point, may increase cache conflicts. Indeed, while Chen *et al* [3] limited inlining to frequent routines only, their results reveal that inlining may not be a stable and effective scheme. For this reason, we do not use inlining. Our algorithm also differs from [4] in that we exploit temporal locality: as indicated above, we use a SelfConfFree area and order the placement of the instruction sequences by frequency of invocation.

As a final step, we traverse the resulting code layout and subdivide the instruction sequences into smaller units that we will use as prefetching blocks. These smaller units we call *prefetching sequences* or simply *sequences*. A prefetching sequence ends when two basic blocks that are laid out contiguously in memory are connected by an arc with a probability of being taken that is lower than *CutBranch*. The default value that we will use for *CutBranch* is 0.5.

4 Instruction Misses in Codes with Optimized Layouts

The extra locality exposed by this algorithm significantly decreases the number of cache misses: for 16 Kbyte instruction caches, the misses in the operating system decrease by 30-70%, leaving the miss rate of the operating system at 0.6-1.6%. However, an important consequence of applying the algorithm is that the code is left in a very special way. This is shown in Table 1. The data in the table corresponds to 16-Kbyte direct-mapped caches with 32-byte lines. We use a *CutBranch* of 0.5. In the table, Columns 2 and 3 show the weighted average length of the prefetch sequences, and their average number of misses respectively. To compute the weighted average length, the weight of a prefetch sequence is proportional to the number of misses in the sequence. This is because, in current machines, misses are very expensive and, therefore, we do not care about long prefetch sequences with few misses. From the table, we see that, on average, prefetch sequences span 7 lines and suffer $35 * 10^{-4}$ misses. The figures are fairly similar in all workloads. Obviously, these figures change if we choose a different *CutBranch*.

To understand how these misses are induced, we classify them into *Sequential* and *Transition* misses. The former are misses that occur in a prefetch sequence while the processor is already executing the sequence. This means that the processor has already executed at least one instruction of the prefetch sequence. Transition misses are those that occur when the processor jumps from one prefetch sequence to another. They occur in the first instruction that the processor executes in the new prefetch sequence. Depending on the type of jump to the new prefetch sequence, transition misses are classified into *Unconditional*, *Likely*, and *Unlikely*. In *unconditional* transition misses, the control transfer is caused by an unconditional branch, a procedure call, or a procedure return. In *likely* transition misses, the control transfer is caused by a conditional branch or a fall-through which, according to the profile of the execution, has a probability larger than or equal to 0.5. *Unlikely* transition misses are like *likely* transition misses except that the probability is smaller than 0.5.

The reason why an unconditional branch, or a procedure call or return can be the end of a prefetch sequence is because the target code may already be placed. For example, Figure 2-(a) shows two basic blocks A and B calling subroutine *foo*. If the subroutine is placed after basic block B, forming a combined prefetch sequence with it and B', prefetch sequence A will be terminated with a procedure call. Similarly, Figure 2-(b) shows the merging of the then and else parts of an if statement. Prefetch sequence A terminates with an unconditional branch.

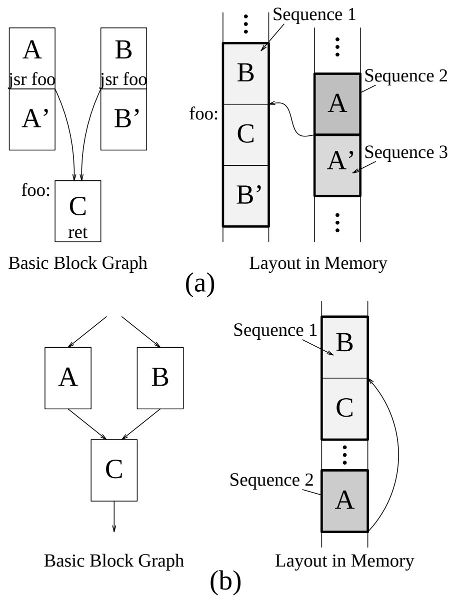


Figure 2: Examples of prefetch sequences terminated by unconditional branches or procedure calls.

The contribution of the different types of misses is shown in Columns 4-7 of Table 1. The table shows that the large majority of the misses (80% on average) are sequential misses. This means that there is more spatial locality to exploit. Cache lines longer than the 32 bytes used can reduce the number of misses [12]. However, longer lines may not be acceptable in systems with a unified secondary cache like the one simulated (Section 6.1) because data caches may not work well with long lines. Furthermore, long lines tend to increase the miss penalty. Another alternative may be to increase the associativity of the cache. This approach, however, does not help as much as expected because, as shown in [12], the layout algorithm has already removed most of the conflict misses. The remaining misses are hard to remove. Furthermore, associative caches are slower. To reduce sequential misses in codes with optimized layouts, instead, we propose a novel and cost-effective approach called *guarded sequential prefetching*. We will describe it in the next section.

The second class of misses, namely transition misses, account for only 20% of the misses on average. *Unconditional* transition misses account for most of these misses. In addition, we see that the *Likely* category accounts for little. This is because our algorithm tends to place two prefetch sequences linked by a *Likely* transition in contiguous locations in memory. Consequently, the prefetching provided by cache lines tends to eliminate these misses. In the next section, we will also suggest how to handle transition misses via prefetching.

Table 1: Characteristics of the prefetch sequences.

Workload	Seq. Length (Cache Lines)	# Misses per Seq. ($*10^{-4}$)	Type of Misses			
			Seq. Misses (%)	Transition Misses (%)		
				Unconditional	Likely	Unlikely
TRFD_4	7.4	26	80.3	12.7	0.0	7.0
TRFD+Make	6.9	36	80.7	12.2	0.3	7.0
ARC2D+Fsck	7.1	51	80.7	13.1	0.2	6.2
Shell	6.9	27	80.0	14.2	0.2	5.8
Average	7.1	35	80.4	13.0	0.2	6.5

5 Prefetching in Optimized Layouts

The experimental data presented above suggests a new prefetching scheme for codes with compiler-optimized layouts. In the following, we first describe the scheme and then discuss its implementation.

5.1 Proposed Scheme

5.1.1 Prefetching Sequential Misses

Our main focus is to eliminate sequential misses, since they are preponderant. We use a new hardware-based sequential prefetching scheme that we call *Guarded Sequential Prefetching*. The general idea is to make available to the prefetch engine some of the information used by the compiler when it generated the optimized layout of the code. In particular, the compiler puts a special mark called the *guard bit* in the last instruction of each prefetch sequence. Then, we let the prefetcher get ahead of the program counter and sequentially prefetch all the instructions until a guard bit is found. At that point, the prefetcher stops. With this scheme, we allow the prefetcher to race quite ahead of the program counter and hide much miss latency. At the same, however, we minimize the chances of prefetching useless instructions by stopping the prefetcher where there is a high probability that the processor will branch off.

The prefetch engine is a simple finite state machine with two states, *Idle* and *Pref*, and two registers, *Start* and *Current* (Figure 3). The *Current* register keeps the address of the most recent memory line that the engine has issued a prefetch for.

Initially, the prefetcher is in the *Idle* state. If an instruction fetch misses (transition 1), the prefetcher performs the following: it enters the *Pref* state, computes the address of the memory line immediately following the one missed on, saves this address in the *Start* and *Current* registers, issues a prefetch for that address and waits for the line to be available in the cache. When the line is finally available, it checks for any guard bits in the line. If at least one is found (transition 2), we have reached the end of the prefetch sequence. The prefetcher stops prefetching and returns to *Idle*. Otherwise (transition 3), the *Current* register is incremented by the size of a memory line and the resulting address is prefetched. We are, therefore, prefetching the next line.

If an instruction fetch misses while the prefetcher is in state *Pref* (transition 4), the prefetcher performs the same operations as in transition 1. Such a miss may be the result of the processor prematurely jumping out of the current prefetch sequence into a new one or simply the result of conflicts within the current prefetch sequence. In the former case, the prefetcher will give up on the current prefetch sequence and follow the processor into the new prefetch sequence. In the latter case, the prefetcher will simply re-prefetch a section of the current prefetch sequence.

When the prefetcher finishes prefetching a prefetch sequence and finally reaches the *Idle* state, it monitors the addresses issued by the fetch unit. If these addresses are within the range of the prefetched sequence (*Start* to *Current* register addresses), the prefetcher remains *Idle* (transition 5). However, when an address A is not within the range (transition 6), the prefetcher performs the same operations as in transition 1 except that the prefetched address is A plus the size of a memory line. This part of the state machine is designed to identify when the processor has jumped off to a new prefetch sequence without suffering a miss.

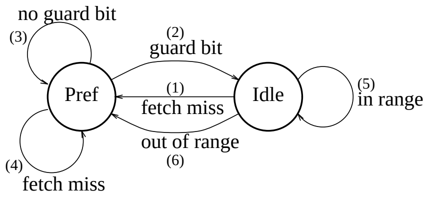
A simplified design of the prefetcher requires neither an address range check in the *Idle* state nor the *Start* register. In such scheme, once the prefetcher finishes prefetching a sequence and finally reaches the *Idle* state, it will perform transition 6 irrespective of the address issued by the processor. This simplified scheme is cheaper and can run faster because it eliminates two register comparisons. However, it causes the prefetcher to initiate more cache accesses.

Clearly, our scheme will work better the longer the prefetch sequences are: the prefetcher will have more time to race ahead of the program counter. Unfortunately, subroutines with multiple callers tend to shorten prefetch sequences. For example, as shown in Figure 2-(a), basic block A is a prefetch sequence by itself: A, C, and A' cannot be combined because C is placed somewhere else. As a result, the prefetcher has to stop at the end of A. To increase the efficiency of the prefetcher, we remove the end-of-sequence guard bit from prefetch sequences like sequence 2 in the figure. Consequently, the prefetcher will race past the procedure call to prefetch the next prefetch sequence in the caller routine. In the best scenario, after the prefetcher finishes prefetching such prefetch sequence, it will realize that the processor is executing callee addresses and will start prefetching the callee. In the example in Figure 2-(a) the result will be the prefetching of A, A', and C with only two steps instead of A, C, and A' in three steps.

In certain corner cases, the prefetcher may prefetch useless data. For example, if the line missed on includes the guard bit, the prefetcher will prefetch past the end of the prefetch sequence. Another example is shown in Figure 2-(a). Indeed, after the prefetcher prefetches A, A', and C, it will continue prefetching B' by default. Overall, while we can devise ways to handle these situations, we feel that the small extra performance gains that can be achieved do not justify the extra complexity involved.

5.1.2 Prefetching Transition Misses

We can refine the technique just described to address transition misses too. Of the transition misses, the categories that matter are *Unconditional* and *Unlikely*. *Unlikely* transition misses are hardly worth prefetching: they occur in the targets of branches that are seldom taken, and they account for only 6.5% of the total misses. *Unconditional* transition misses, on the other hand, are attractive to prefetch because they occur in the target of fully predictable control transfers.



Transition	Description
(1)	Fetch miss; prefetch the next line.
(2)	Guard bit found; stop prefetching.
(3)	No guard bit found; continue prefetching next line.
(4)	Fetch miss; redirect prefetching to the line following the one missed on.
(5)	Instruction fetch in the range of prefetched lines; do not prefetch.
(6)	Instruction fetch out of the range of prefetched lines; redirect prefetching.

Figure 3: Finite state machine to perform guarded sequential prefetching.

We now consider how guarded sequential prefetching needs to be extended to target *unconditional* transition misses as well.

There is a hardware and a software approach to the problem. The least attractive, yet conceptually simplest approach, is the hardware one: when the prefetcher reaches an unconditional control transfer, it partially decodes the instruction, finds the target address, and starts prefetching from the target address. This approach requires a prefetcher with advanced decoding capabilities. Alternatively, we could use a branch target buffer. Of course, some of the unconditional control transfers use a register to specify the target. This makes it harder for the prefetcher to prefetch the correct target.

The software approach involves inserting, as early as possible in the code, a software prefetch instruction with the address of the target instruction. The execution of the instruction by the processor forces the prefetcher to start prefetching from the target address. While this approach does not require that the prefetcher have advanced decoding capabilities, it has two disadvantages. Indeed, it has software overhead and relies on the compiler to move the prefetch instruction up in the code, which may be hard. Furthermore, we do not want to put the prefetch too early in case the prefetcher has not yet reached the end of the prefetch sequence.

In any case, with either the hardware or software approach, the changes to the finite state machine of Figure 3 are minor. The new state machine is shown in Figure 4. The only change for the hardware approach is that transition (4) is also triggered by the prefetcher decoding an unconditional control transfer instruction. The only change for the software approach is that transitions (4) and (1) are also triggered by the processor executing a prefetch instruction.

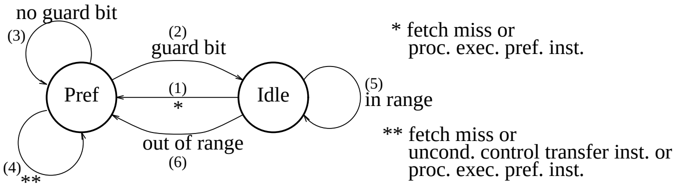


Figure 4: Finite state machine to perform guarded sequential prefetching and transition prefetching. It includes the changes for both the hardware and software approaches.

With the transition prefetching support, we can prefetch basic blocks *A*, *C*, and *A'* in Figure 2-(a) in one shot. In many cases, however, routines that have several callers are small and miss-free. For example, the popular routines in the SelfConfFree area (Section 3.1) are small and cannot conflict with any other operating system routine. In this case, forcing the prefetcher to prefetch the callee does not help save any misses while it only slows down the prefetcher; it would be best if the prefetcher continued prefetching the instructions in the caller.

A heuristic that we can use to optimize this case is that, if the first instruction in the callee routine does not miss, we assume that the remaining instructions in the callee will not miss either. Therefore, when the prefetch for the target of the procedure call instruction is issued, if it hits in the cache, the prefetcher stops prefetching the callee instructions; it goes back to prefetching instructions in the caller. The resulting transition prefetching schemes we call *Probe* transition prefetching. They are like the previous ones except for the prefetching of the callee routines after procedure call instructions. Clearly, probe transition prefetching involves adding more complexity to the prefetch engine.

5.2 Implementation

5.2.1 Guarded Sequential Prefetching

If the processor already supports some form of sequential prefetching, supporting guarded sequential prefetching requires only minor changes. Overall, to support guarded sequential prefetching we need support in the instruction encoding, operating system, and processor microarchitecture. We consider each issue in turn.

The instruction encoding support involves encoding an end-of-sequence guard bit in control transfer instructions. A version of these instructions will have this bit set, while a second version will have it reset. The compiler uses the version with the bit set if the instruction is the last one of a prefetch sequence; otherwise it uses the version with the bit reset. Supporting the guard bit is no more difficult than supporting the "hint" bit(s) for static branch prediction that are present in Hewlett-Packard's PA-8000 processor.

The operating system support involves adding one bit to each entry in the TLB and page tables to mark the pages that contain code with an optimized layout. When several processes are multiprogrammed in a processor, the hardware can then dynamically enable and disable the guarded sequential prefetching hardware on an instruction by instruction basis, depending on the page that the currently-executing instruction is in. For example, if the currently-executing instruction is in a page that has its TLB bit set, guarded sequential prefetching is enabled; otherwise, the machine performs a more traditional form of prefetching or no prefetch at all. A related TLB support can be found in Hewlett-Packard's PA-8000 processor: it uses a TLB bit to choose between two different branch prediction modes.

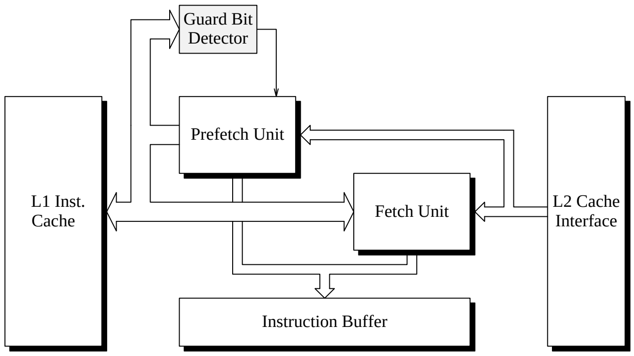
Finally, we consider the microarchitecture support required. We design it as an add-on feature to traditional sequential prefetching schemes. We consider two types of traditional sequential prefetching schemes: non-tagged and tagged [10]. Non-tagged schemes are simple schemes that prefetch line $i+1$ after a reference to line i, or after a miss on line i. In tagged schemes, a bit is added to each cache line to improve the prefetching algorithm. The processor prefetches line $i+1$ only when line i is referenced for the first time since it was brought into the cache. Several of these schemes are currently used by processors. For example, on-miss prefetching is implemented in the Alpha 21164 processor, while tagged prefetching is implemented in the PA-7200 processor.

A design of a guarded sequential prefetcher based on traditional non-tagged and tagged prefetchers is shown in Figures 5-(a) and 5-(b) respectively. The figures show only the datapath, not the control or address signals, except for the output of the Guard Bit Detector (GBD) module. In both designs, the only new module required for guarded sequential prefetching is the GBD module. In addition, of course, the control logic of the prefetch unit needs some minor changes. In the figures, the instruction buffer stores the cache line with the instruction currently being executed. It is important to use the instruction buffer to decouple instruction dispatching from instruction fetching: the instruction cache port is freed up for prefetching. For example, the UltraSparc has a 12-deep instruction buffer, while the Alpha 21164 has two 4-deep instruction buffers.

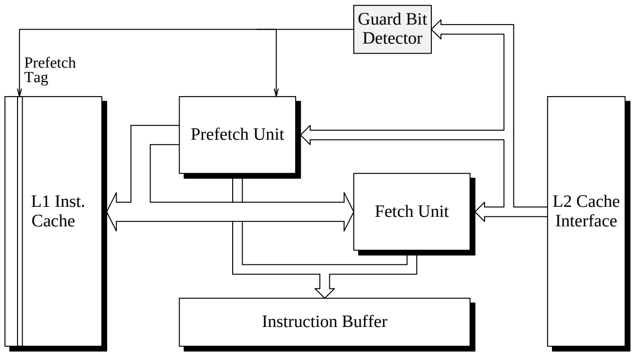
The two figures show the paths along which instructions can be transferred in the traditional prefetching schemes: from the secondary cache interface to the fetch unit and then to the instruction buffer and primary cache; from the primary cache to the fetch unit and then to the instruction buffer; and from the secondary cache interface to the prefetch unit and then to the primary cache. A transfer along the latter path may also involve sending the line to the instruction buffer. This occurs when the fetch unit requests an instruction that is currently being prefetched. In both figures, when the prefetch unit issues a prefetch, it first checks the primary cache. This cache access is performed with lower priority than accesses by the fetch unit. If the check misses in the primary cache, then a prefetch is issued to memory.

We consider the extensions to a non-tagged prefetcher first (Figure 5-(a)). In guarded sequential prefetching, the GBD module needs to snoop on the instructions requested by the prefetcher, both those that are currently in the *L1* cache (on a prefetch hit) and those that are brought in during a cache refill (on a prefetch miss). For the operation of the GBD module to be fast, the connection between *L1* cache and prefetch unit should be wide. If it is as wide as a cache line, the guard bit detection can be finished in one cycle. Otherwise it may take longer. It is better, of course, to perform the detection as fast as possible, so that the cache is freed up quickly.

The GBD module needs to identify control transfer instructions and whether or not their guard bit is set. The encoding of the instructions determines how easy it is to do so. In the best possible case, two opcode bits would identify control transfer instructions and all of these instructions would have one free bit in the same position that could be used as a guard bit. In this case, the GBD could be as simple as a 3-input *and* gate. Clearly, with real instruction set encodings, the GBD will be a bit more complex. However, it will not be more complex than the logic required to identify the "hint" bit(s) for static branch prediction in the PA-8000 processor.



(a) Design based on a non-tagged prefetcher.



(b) Design based on a tagged prefetcher

Figure 5: Design of a guarded sequential prefetcher.

In the tagged prefetcher, the extensions are different (Figure 5-(b)). The traditional tagged prefetcher requires a one-bit prefetch tag per cache line. In guarded sequential prefetching, we will reuse this one-bit prefetch tag for a different purpose. It will now denote whether or not the line has a guard bit set. As indicated before, in the presence of programs with or without optimized layout, the meaning of the tag is determined by a TLB bit. In our scheme, every time that the prefetcher loads a line from the secondary cache, the GBD module checks whether it contains a guard bit set. If it does, as the line is loaded into the cache, the GBD module sets the corresponding one-bit prefetch tag in the cache. Otherwise, it resets it. This operation can be performed in a pre-decoding stage like in the UltraSparc. Then, every time that the prefetcher checks whether a given line is in the *L1* cache, we also check whether the corresponding one-bit prefetch tag is set. Note that this simple check does not need extra time to take place. Therefore, the cache is kept unloaded without the need of a wide port.

In both Figures 5-(a) and 5-(b), once the GBD module determines whether or not a prefetched line contains a guard bit set, it commands the prefetch unit to stop prefetching or prefetch the next line. Obviously, to support guarded sequential prefetching, we need to make some small modifications to the control logic of the prefetch engine. These are minor changes. Furthermore, the prefetch unit prefetches only one memory line at a time. Such single-issue prefetch engine makes the hardware simpler.

Overall, transforming a traditional non-tagged or tagged sequential prefetch engine to support guarded sequential prefetching is very simple and inexpensive.

5.2.2 Transition Prefetching

Given the small gains that transition prefetching will produce (Section 6.4), a detailed design of hardware-based transition prefetching is not worth presenting. Conceptually, the implementation needs to use a decoder [9] or a connection to a branch target buffer [6, 11] in order to determine the address of the branch target. A stack may also be necessary to maintain the addresses of procedure returns. Obviously, traditional sequential prefetchers need non-trivial changes to support it. Support for software-based transition prefetching, however, is easy to add to the schemes of Figure 5. When the processor decodes a prefetch instruction, it should signal the prefetcher to start prefetching from the target address. In addition, the compiler should move the prefetch instruction up in the code as much as possible.

6 Evaluation

After describing the proposed prefetching scheme, we now use simulations to evaluate its impact. In the following, we first describe the machine that we simulate. Then, we evaluate guarded sequential prefetching, compare it to traditional sequential prefetching schemes and, finally, evaluate transition prefetching.

6.1 Architecture Simulated

Since we use instruction and data multiprocessor traces, we simulate a multiprocessor architecture. The architecture simulated has 4 200-MHz processors. Each processor has a 16-Kbyte primary instruction cache, a 32-Kbyte primary data cache, and a 256-Kbyte unified lockup-free secondary cache, all direct-mapped. The primary data cache is write through and, like the instruction cache, has 32-byte lines. The secondary cache is write back and has 32-byte lines. There is a 4-deep 32-byte wide write buffer between the primary data cache and the secondary cache and an 8-deep 32-byte wide write buffer between the secondary cache and bus. Reads bypass writes and release consistency is used. The bus is 8 bytes wide, cycles at 40 MHz, and has split transactions. Each secondary cache line transfer uses the bus for 20 processor cycles. Without resource contention, it takes 1, 14, and 53 cycles for the processor to fetch a line from the primary cache, secondary cache, and memory respectively. Both instruction and data accesses of both applications and operating system are simulated. Contention is fully modeled. For the simulations of guarded sequential prefetching, we use the design based on the tagged prefetcher (Figure 5-(b)).

6.2 Impact of Guarded Sequential Prefetching

To evaluate guarded sequential prefetching, we first look at its impact on misses and then on execution time. To gain a deeper insight, we consider different versions of the code placement algorithm, namely those with a *CutBranch* threshold (Section 3.2) of 0.9, 0.8, 0.6, 0.5, 0.4, 0.2, and 0.0. The closer that *CutBranch* gets to 1, the more conservative the prefetching is. Indeed, for high values of *CutBranch*, the prefetch sequences contain only instructions that have a high probability of being executed in sequence. Therefore, while there is a smaller chance to prefetch useless instructions, the prefetch sequences are shorter. This means that the prefetcher has less time to race ahead of the program counter and, therefore, latency hiding is bound to be less successful. The opposite occurs for low values of *CutBranch*.

Table 2 shows the weighted length of the prefetch sequences for different values of *CutBranch*. The data corresponds to the four workloads combined. The table shows that, as expected, lower *CutBranch* thresholds induce longer prefetch sequences.

Table 2: Weighted length of the prefetch sequences measured in 32-byte cache lines for different values of the *CutBranch* threshold.

CutBranch	0.9	0.8	0.6	0.5	0.4	0.2
Length	3.1	3.8	5.7	7.1	9.5	20.3

Using code layouts generated with these values of *CutBranch*, Figure 6 shows the effect of guarded sequential prefetching on the number of operating system instruction misses in the primary cache. For each workload, the figure shows 9 bars, corresponding to 9 environments. From left to right, *Base* corresponds to the original Concentrix operating system; *LayoutOpt* is *Base* with an optimized layout and no prefetching, and *Seq(i)* is *LayoutOpt* with *CutBranch* equal to *i* and guarded sequential prefetching. For all workloads, the bars are normalized to the misses in *LayoutOpt*. Except for *Base*, each bar is decomposed into unlikely transition misses (*Unlikely*), unconditional transition misses (*Uncond.*), sequential misses (*Seq.*) and likely transition misses (*Likely*). In the figure, misses partially-hidden by prefetches are still considered misses. The *Likely* misses are too few to show in the figure. The instruction miss rate of the operating system for *LayoutOpt* is 0.63%, 1.14%, 1.57%, and 0.82% for *TRFD_4*, *TRFD+Make*, *ARC2D+Fsck*, and *Shell* respectively.

From the figure, we see that guarded sequential prefetching reduces the number of misses over *LayoutOpt* in all cases. The reductions are the largest for a *CutBranch* threshold of 0.5, where the prefetch sequences are as large as possible while always containing basic blocks that are more likely to be executed in sequence than not. For larger thresholds, the prefetching is too conservative and, therefore, some instructions are not prefetched early enough. For smaller thresholds, the prefetching is too aggressive and the large amount of prefetched instructions causes conflicts in the cache that result in misses. A *CutBranch* of 0.0 is obviously undesirable.

Overall, *Seq(0.5)* eliminates 66% of the misses in *LayoutOpt*. Note that the misses in *LayoutOpt* were already few compared to the original code. Furthermore, *Seq(0.5)* eliminates 71% of the sequential misses from *LayoutOpt*. With this optimization, very few instruction misses remain. Indeed, the miss rate of the operating system instructions in the simulated 16-Kbyte caches is now 0.18-0.49%.

To see the complete picture, Figure 7 shows the impact of guarded sequential prefetching on execution time. The figure shows the total execution time of the operating system. For each workload, the figure has the same 9 environments as Figure 6. For each bar, the execution time is divided into time that the processor is stalled due to instruction misses in any of the two caches (*I Miss*), data read misses in any of the two caches (*D Read Miss*), or write buffer overflow (*D Write*), and the time the processor is busy executing operating system instructions (*Exec*). Like in the previous figure, for a given workload, the execution time is normalized

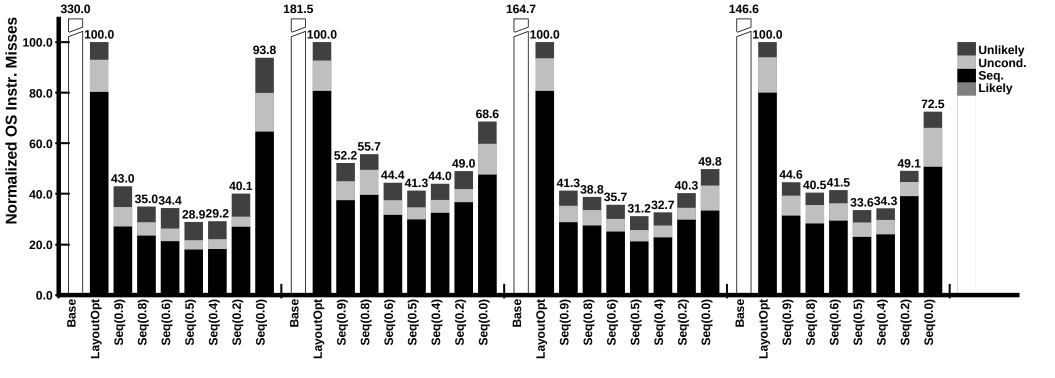


Figure 6: Normalized number of instruction misses in the operating system with and without guarded sequential prefetching and for different code layouts. The data corresponds to the 16-Kbyte primary cache.

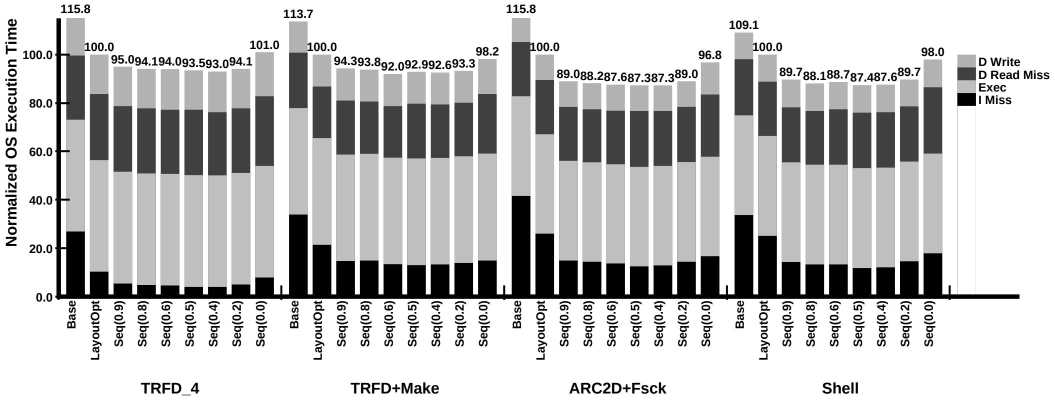


Figure 7: Normalized execution time of the operating system with and without guarded sequential prefetching and for different code layouts.

to that of *LayoutOpt*.

From the figure, we observe that *I Miss* accounts for a small fraction (20.7%) of the execution time in *LayoutOpt*. Indeed, *Exec* and *D Read Miss* each accounts for as much or even more. Therefore, any optimization aimed at reducing the instruction misses necessarily has a modest impact on execution time. Aside from this, we note that the *I Miss* times and total execution times show a similar U-shaped curve as that exhibited by misses: the lowest execution time corresponds to a 0.5-0.4 *CutBranch* threshold. For these values of *CutBranch*, the execution time is, on average, 10.2% smaller than for *LayoutOpt*. Therefore, we conclude that guarded sequential prefetching is an attractive optimization.

6.3 Comparison to Traditional Sequential Prefetching

To put the impact of guarded sequential prefetching in the proper context, we need to compare it to what other sequential prefetching schemes with comparable hardware complexity achieve. We first consider three traditional schemes described in [10]: next-line prefetch on access (*NLalways*), next-line prefetch on miss (*NLmiss*), and tagged prefetch as discussed in Section 5.2.1 (*NLtagged*). We further extend *NLalways* to next N-line prefetch on access: when the processor accesses line i, the prefetcher sequentially prefetches lines $i+1$ to $i+N$; later, when the processor accesses a line out of this range, the prefetcher prefetches the next block of lines. We choose N to be 4, 8, and 16 and call the systems *N4Lalways*, *N8Lalways*, and *N16Lalways*, respectively. To compare all these prefetching schemes, we examine cache misses and execution time.

Figure 8 compares the number of primary cache misses in all of these environments to the number in *Seq(0.5)* and *LayoutOpt*. As usual, for each workload, the bars are normalized to *LayoutOpt*. The figure shows that, in all applications, *Seq(0.5)* has quite fewer misses than the other sequential prefetching schemes. For example, *Seq(0.5)* has, on average, 38% fewer misses than *NLmiss*. This is because *NLmiss* has to suffer at least as many misses as it saves. Similarly, *Seq(0.5)* has, on average, 20% fewer misses than *NLalways* and 24% fewer misses than *NLtagged*. This is because, in *NLalways* and *NLtagged*, the prefetcher only gets one line ahead of the processor. Therefore, it is hard to hide long-latency misses. *NLtagged* is slightly worse than *NLalways* because it is a bit more conservative.

Next N-line schemes significantly outperform the three next-line schemes because the prefetcher can get several lines ahead of the processor. The results, however, are still worse than *Seq(0.5)*. To see why, consider the average prefetch sequence length. As indicated in Table 1, it is 7.1. For small N (*N4Lalways*), the prefetcher often stops before reaching the end of the prefetch sequence. Therefore, it is less able to hide latency than *Seq(0.5)*, which reaches it. For large N (*N16Lalways*), however, the prefetcher often prefetches past the end of the prefetch sequence, bringing useless instructions and causing cache pollution. The best case is where N is very close to the average prefetch sequence length (*N8Lalways*). However, even in this case, the results are still 19% worse than *Seq(0.5)*. The reason is that few prefetch sequences have a length equal to the average; instead, there are long and short prefetch sequences. Therefore, we do not recommend fixed N-line schemes. In addition, a fixed N-

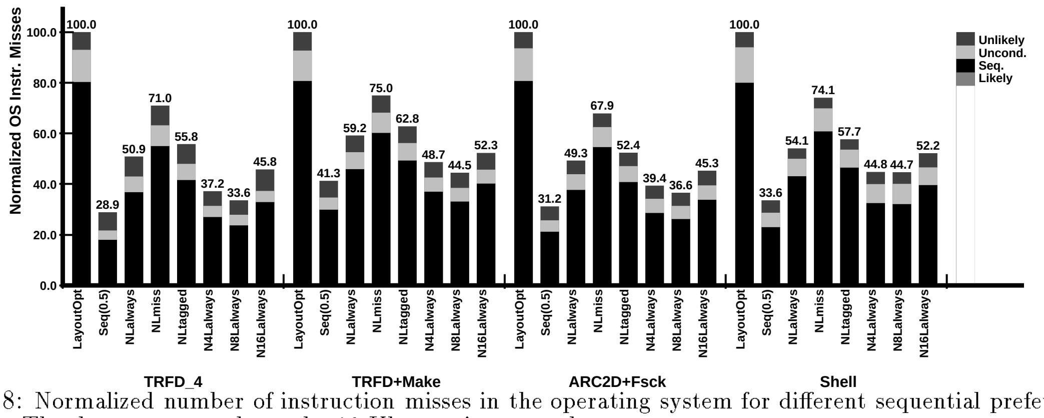


Figure 8: Normalized number of instruction misses in the operating system for different sequential prefetching schemes. The data corresponds to the 16-Kbyte primary cache.

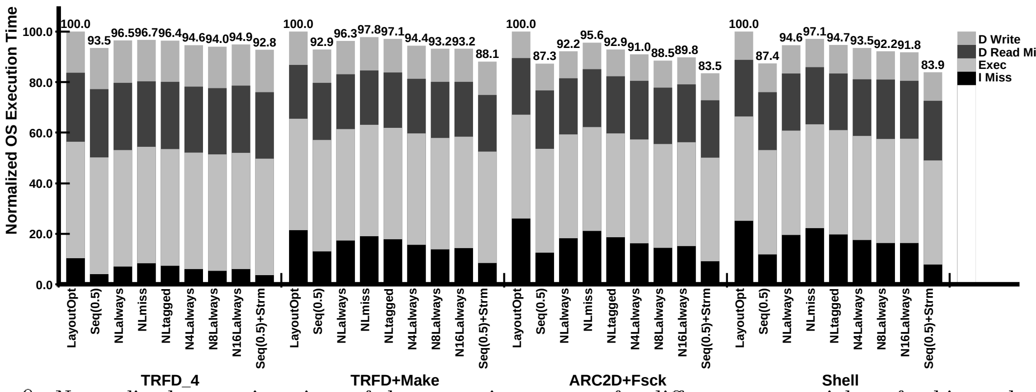


Figure 9: Normalized execution time of the operating system for different sequential prefetching schemes.

line prefetch scheme that works well for systems code is not likely to work well for loop-intensive codes. Indeed, for the latter codes, the best N is probably the one that matches the size of the body of the most important inner loop in the program.

The effect of these techniques on the execution time of the operating system is shown in Figure 9. The figure is organized like Figure 7 and shows the environments in Figure 8 plus another one that we consider later. As expected, *Seq(0.5)* is faster than the other schemes considered. In particular, it is 5% faster than *NLalways* and *NLtagged*, 7% faster than *NLmiss*, and 2% faster than *N8Lalways*. While 2% is a small number, N=8 works well here because it matches the average length of the prefetch sequences; in another workload, N=8 will perform worse.

6.3.1 Enhancing Guarded Sequential Prefetching with Stream Buffers

An analysis of the *Seq(0.5)* system shows that a sizable fraction of its *I Miss* time comes from secondary cache misses that occur *in sequence*. These misses slow down the guarded sequential prefetcher. This suggests that we can further reduce the execution time by placing stream buffers [5] between the secondary caches and the memory bus to complement the guarded sequential prefetcher. We add two 8-entry direct-access stream buffers per processor. The buffers are allowed to look up the secondary cache, thus avoiding unnecessary bus accesses when the line is in the secondary cache. We select two buffers to be able to intercept references to the caller and callee procedures in a procedure call. The execution time of the resulting system is shown in Figure 9 as *Seq(0.5)+Strm*. The new system is 3.5% faster than *Seq(0.5)*. We feel that it is a good combination.

We consider a range of parameters for the stream buffers. We find that increasing the depth of the buffers to 16 results in no gain. This is probably because the average length of the prefetch sequences is 7 lines. Furthermore, adding one or two extra stream buffers to our configuration consistently worsens performance. This is because these extra buffers are mostly useless and, in addition, create bus traffic.

While we use stream buffers between secondary caches and the bus, we think that using stream buffers between primary and secondary caches is not as good a choice as using a guarded sequential prefetcher. Indeed, stream buffers often prefetch useless instructions, while guarded sequential prefetchers have a high efficiency. In addition, stream buffers require much more silicon than guarded sequential prefetchers.

6.4 Impact of Transition Prefetching

Finally, we examine the impact of adding support for transition prefetching to our guarded sequential prefetcher. We consider the four schemes described in Section 5.1.2. We compare *Seq(0.5)* to *Seq(0.5)* plus hardware-based transition prefetching (*S+T(Hard)*), *Seq(0.5)* plus software-based transition prefetching (*S+T(Soft)*), *Seq(0.5)* plus hardware-based probe transition prefetching (*S+PT(Hard)*), and *Seq(0.5)* plus software-based probe transition prefetching (*S+PT(Soft)*). For the software-based schemes, since we do not have compiler support, we place the prefetch instruction at the beginning of the basic block that contains the unconditional control transfer instruction. As before, we examine misses in the primary cache and execution time.

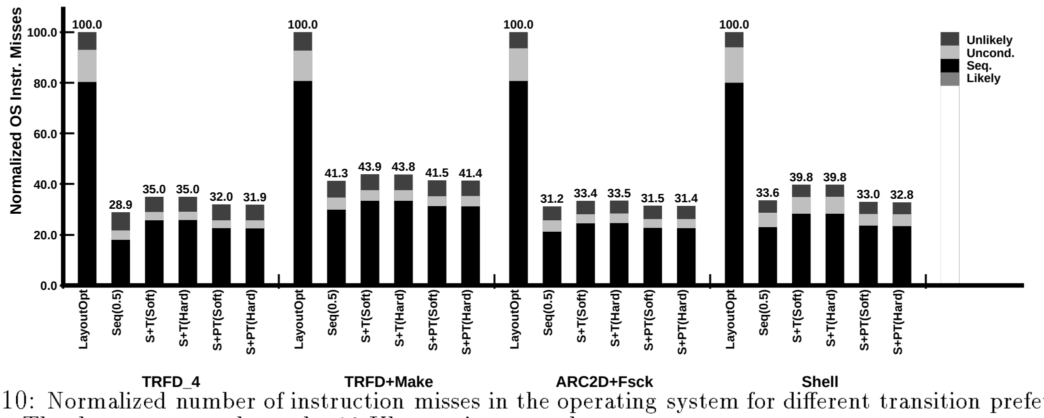


Figure 10: Normalized number of instruction misses in the operating system for different transition prefetching schemes. The data corresponds to the 16-Kbyte primary cache.

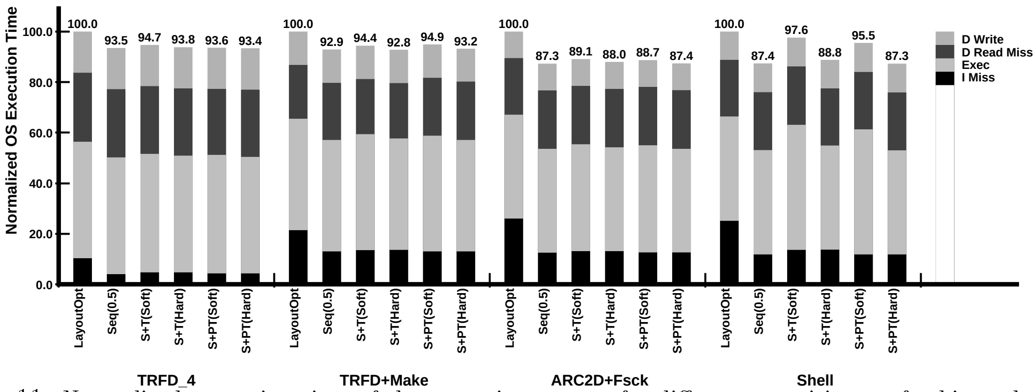


Figure 11: Normalized execution time of the operating system for different transition prefetching schemes.

Figure 10 shows how adding transition prefetching affects the number of operating system instruction misses. As usual, for each application, the bars are normalized to *LayoutOpt*. Overall, adding transition prefetching is undesirable: it decreases the number of unconditional transition misses only slightly, while it increases the number of sequential misses by a larger amount. The reason why the decrease in unconditional transition misses is so small is that *Seq(0.5)* alone already removed 63% of them and, therefore, few remain for the transition prefetcher to eliminate. The second effect, namely an increase in sequential misses, results from transition prefetching disrupting sequential prefetching by polluting the cache.

Another observation is that probe transition prefetching schemes save some misses over the plain transition prefetching schemes. While the miss reductions are small, they confirm the hypothesis that suggested probe transition prefetching. Finally, we note that both hardware and software schemes tend to have a similar impact on misses.

To see the effect on the execution time, we now examine Figure 11. The figure shows the same environments as in Figure 10 and is organized like Figure 9. We can see that, overall, none of the transition prefetching schemes improves the performance of the machine. In fact, the software-based schemes even slow down the machine due to the overhead of the prefetch instructions. Such overhead appears in the *Exec* category. Overall, therefore, we do not recommend transition prefetching: guarded sequential prefetching alone is the most cost-effective scheme.

7 Related Work

To our knowledge, none of the previous work has focused on prefetching codes with optimized layouts, especially systems codes. This is our main contribution. There is, however, a large body of related work. First, McFarling [7], Hwu and Chang [4], and Torrellas *et al* [12] studied code layout optimization for cache performance. However, they did not investigate instruction prefetching on the optimized codes.

Instruction prefetching without layout optimization has been a topic frequently addressed in the past. For example, Smith examined the three next-line sequential prefetching schemes described in Section 5.2.1 [10]. Our results agree with his on the relative performance of the schemes. These schemes do not tolerate long latencies because they prefetch only one line ahead. An effective improvement is the stream buffer proposed by Jouppi [5]. Stream buffers prefetch successive lines after a miss. We do not compare stream buffers to guarded sequential prefetching because our scheme is a much cheaper addition to next-line prefetchers. Instead, we add stream buffers between the secondary caches and the bus. Prefetching the next N lines after a miss was evaluated by Uhlig *et al* for large codes [13] and found effective. To reduce cache pollution, they suggested caching prefetched lines only if they are used. This optimization, however, actually reduced performance. Smith and Hsu [11] investigated the *fetch-ahead distance* in a next-line prefetch scheme. This distance is the number of instructions that remain to be issued in a line before a prefetch request for the next line should be issued. Next-line prefetching has also been studied by Lee *et al* [6] in the context of speculative execution. Overall, several authors have pointed out that smallish cache lines with sequential prefetching are better than long lines [10, 11, 13].

Smith and Hsu [11] studied target prefetching with the support of a target prediction table. They found that target prefetching performed slightly worse than next-line sequential prefetching. However, the combination of both schemes was best. A different approach taken by Pierce and Mudge [9] is the *wrong-path* prefetching. This scheme prefetches both paths of a conditional branch. They found that 70-80% of the performance gain came from prefetching the fall-through path instead of prefetching the target. We do not compare these target prefetching schemes to our transition prefetching schemes because our optimized layout changes many of the frequently taken branches to fall-throughs.

8 Conclusions

This paper presented a new scheme to hide instruction misses on systems codes whose layout was optimized for reduced cache conflicts. We call the scheme guarded sequential prefetching. In this scheme, compilers leave hints in the code as to how the code was laid out; then, at run time, the prefetching hardware uses these hints to improve prefetching. In the actual scheme presented here, the code is laid out in sets of basic blocks with high spatial locality called prefetch sequences. The end of each prefetch sequence is marked with a set guard bit. At run time, the prefetcher sequentially prefetches until it finds a guard bit set. This scheme is very cheap: we presented a detailed design based on minor extensions to existing sequential prefetchers. Of course, the scheme can be applied to applications code as well. In addition, the scheme can be turned off and on at run time with one TLB bit.

The scheme was evaluated with simulations using complete traces from a 4-processor machine. Overall, for 16-Kbyte primary instruction caches, guarded sequential prefetching removes an average of 66% of the instruction misses remaining in an operating system with an optimized layout, speeding up the operating system by 10%. Furthermore, the scheme is more cost-effective and robust than existing sequential prefetching techniques. Finally, extensions to guarded sequential prefetching that involve prefetching across prefetch sequence transitions are found not to be cost-effective.

Acknowledgments

We thank Liuxi Yang for his help with trace collection and the design of prefetchers. We also thank Russell Daigle, Tom Murphy and Perry Emrath. We thank the referees and the graduate students in the I-ACOMA group for their feedback. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

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

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

[3] 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.

[4] 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.

[5] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[6] D. Lee, J. Baer, B. Calder, and D. Grunwald. Instruction Cache Fetch Policies for Speculative Execution. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 357–367, June 1995.

[7] 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.

[8] 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.

[9] J. Pierce and T. Mudge. Wrong-Path Instruction Prefetching. Technical Report CSE-222-94, University of Michigan, 1994.

[10] A. J. Smith. Cache Memories. In *Computing Surveys*, pages 473–530, September 1982.

[11] J. E. Smith and W.-C. Hsu. Prefetching in Supercomputer Instruction Caches. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 588–597, July 1992.

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

[13] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction Fetching: Coping with Code Bloat. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 345–356, June 1995.

[14] C. Xia and J. Torrellas. Improving the Data Cache Performance of Multiprocesor Operating Systems. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 85–94, February 1996.