Architectural and Compiler Support for Effective Instruction Prefetching: A Cooperative Approach

CHI-KEUNG LUK
Compaq Computer Corporation
and
TODD C. MOWRY
Carnegie Mellon University

Instruction cache miss latency is becoming an increasingly important performance bottleneck, especially for commercial applications. Although instruction prefetching is an attractive technique for tolerating this latency, we find that existing prefetching schemes are insufficient for modern superscalar processors, since they fail to issue prefetches early enough (particularly for nonsequential accesses). To overcome these limitations, we propose a new instruction prefetching technique whereby the hardware and software *cooperate* to hide the latency as follows. The hardware performs aggressive sequential prefetching combined with a novel *prefetch filtering* mechanism to allow it to get far ahead without polluting the cache. To hide the latency of nonsequential accesses, we propose and implement a novel compiler algorithm which automatically inserts *instruction-prefetch instructions* into the executable to prefetch the targets of control transfers far enough in advance. Our experimental results demonstrate that this new approach hides 50% or more of the latency remaining with the best previous techniques, while at the same time reduces the number of useless prefetches by a factor of six. We find that both the *prefetch filtering* and *compiler-inserted prefetching* components of our design are essential and complementary, and that the compiler can limit the code expansion to only 9% on average. In addition, we show that the performance of our technique can be further increased by using profiling information to help reduce cache conflicts and unnecessary prefetches. From an architectural perspective, these performance advantages are sustained over a range of common miss latencies and bandwidth. Finally, our technique is cost effective as well, since it delivers performance comparable to (or even better than) that of larger caches, but requires a much smaller hardware budget.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; *Optimization*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms: Performance, Design, Experimentation

Additional Key Words and Phrases: Instruction prefetching, compiler optimization

---

Authors' addresses: C.-K. Luk, Alpha Development Group, Compaq Computer Corporation, 334 South Street, Shrewsbury, MA 01545; email: chi-keung.luk@compaq.com; T. C. Mowry, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: tcm@cs.cmu.edu.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2001 ACM 0734-2071/01/0100–0071 $5.00

1. INTRODUCTION

The latency of fetching instructions is a key performance bottleneck in modern systems, and the problem is expected to get worse as the gap between processor and memory speeds continues to grow. While instruction caches are a crucial first step, they are not a complete solution. For example, a study conducted by Maynard et al. [1994] demonstrates that many commercial applications suffer from relatively large instruction cache miss rates (e.g., over 20% in an 8KB cache) due to their large instruction footprints and poor instruction localities. To further tolerate this latency, one attractive technique is to automatically *prefetch* instructions into the cache before they are needed.

1.1 Previous Work on Instruction Prefetching

Several researchers have considered instruction prefetching in the past. We will begin by discussing and then quantitatively evaluating four of the most promising techniques that have been proposed to date, all of which are purely hardware-based: *next-N-line* prefetching [Smith 1978; 1982], *target-line* prefetching [Smith and Hsu 1992], *wrong-path* prefetching [Pierce and Mudge 1996], and *Markov* prefetching [Joseph and Grunwald 1997].

Before we begin our discussion, we briefly introduce some prefetching terminology. The *coverage factor* is the fraction of original cache misses that are prefetched. A prefetch is *unnecessary* if the line is already in the cache (or is currently being fetched), and is *useless* if it brings a line into the cache which will not be used before it is displaced. An ideal prefetching scheme would provide a coverage factor of 100% and would generate no unnecessary or useless prefetches. In addition, the *timeliness* of prefetches is also crucial. The *prefetching distance* (i.e., the elapsed time between initiating and consuming the result of a prefetch) should be large enough to fully hide the miss latency, but not so large that the line is likely to be displaced by other accesses before it can be used (i.e., a useless prefetch).

The idea behind *next-N-line prefetching* [Smith 1978; 1982] is to prefetch the N sequential lines following the one currently being fetched by the CPU. A larger value of N tends to increase the prefetching distance, but also increases the likelihood of polluting the cache with useless prefetches. The optimal value of N depends on the line size, the cache size, and the behavior of the application itself. Next-N-line prefetching captures sequential execution as well as control transfers where the target falls within the next N lines. It is usually included as part of other more complex instruction prefetching schemes, and based on our experiments, it accounts for most of the performance benefit of these previously existing schemes.

To further expand the scope of prefetching to capture more control transfer targets, Smith and Hsu [1992] proposed *target-line prefetching* which uses a prediction table to record the address of the line which most recently followed a given instruction line, thus enabling hardware to prefetch targets whenever an entry is found in this table. They observed

that combining target-line prefetching with next-1-line prefetching produced significantly better results than either technique alone.

Rather than relying on history tables, Pierce and Mudge [1996] proposed *wrong-path prefetching* which combines next-N-line prefetching with always prefetching the target of control transfers with static target addresses. Hence, for conditional branches, both the target and fall-through lines will always be prefetched. However, since target addresses cannot be determined early, this scheme only outperforms next-N-line prefetching when a conditional branch is initially untaken but later taken (assuming that enough time has passed to hide the latency but not so much that the line has been displaced). Their results indicated that wrong-path prefetching performed slightly better than next-1-line prefetching on average.

Joseph and Grunwald [1997] proposed *Markov prefetching*, which correlates consecutive miss addresses. These correlations are stored in a *miss-address prediction table* which is indexed using the current miss address, and which can return multiple predicted addresses. The Joseph and Grunwald study focused primarily on data cache misses, and did not compare Markov prefetching with techniques designed specifically for prefetching instructions.

Reinman et al. [1999a] recently proposed a prefetching technique for architectures with a decoupled front-end [Stark et al. 1997; Reinman et al. 1999b] in which the branch predictor can run ahead of the instruction fetcher upon instruction cache misses, thus providing predicted instruction addresses to prefetch. In our study, we consider conventional front-ends—i.e., where branch prediction stops when the predicted path suffers an instruction cache miss—which are representative of existing commercial microprocessors. One fundamental difference between our approach and approaches that rely upon accurate branch prediction to determine prefetching addresses is that our scheme (which we will describe later in Section 2) can effectively hide miss latencies even in the face of inherently unpredictable branches, since it will prefetch *both* paths following the unpredictable branch in such cases. Rather than trying to predict a single likely control path, our approach considers prefetching far enough ahead along *all* possible paths, and it suppresses prefetches only for cases that have demonstrated themselves not to be useful in the past. Hence, although we do model a realistic branch predictor in our experiments, our approach is not limited by the effectiveness of branch prediction.

Finally, we note that while Xia and Torrellas [1996] considered instruction prefetching for codes where the layout has already been optimized using profiling information, we focus only on techniques which do not require changes to the instruction layout in this study.

1.1.1 *Performance of Existing Instruction Prefetching Techniques.* To quantify the performance benefits and limitations of the four prefetching techniques described above, we implemented each of them within a detailed, cycle-by-cycle simulator which models an out-of-order four-issue superscalar processor based on the MIPS R10000 [Yeager 1996]. We model

Table I. Parameters Used in the Evaluation of Existing Instruction Prefetching Techniques

Technique	Number of Sequential Lines Prefetched	Target Prefetching Parameters		
		Number of Targets	Table Size	Table Indexing Method
Next-N-Line	$N = 2, 4, 8$	0	0	N/A
Target-Line	2	1	64 entries	direct-mapped with tags
Wrong-Path	2	1	0	N/A
Markov	2	2	512KB	direct-mapped with tags

a two-level cache hierarchy with split 32KB, two-way set-associative primary instruction and data caches and a unified 1MB, four-way set-associative secondary cache. Both levels use 32–byte lines. The penalty of a primary cache miss that hits in the secondary cache is at least 12 cycles, and the total penalty of a miss that goes all the way to memory is at least 75 cycles (plus any delays due to contention, which is modeled in detail). To provide better support for instruction prefetching, we further enhanced the primary instruction cache relative to the R10000 as follows: we divide it into four separate banks, and we add an eight-entry victim cache [Jouppi 1990] and a 16-entry prefetch buffer [Joseph and Grunwald 1997]. Further details on our experiments will be presented later in Section 5.

Table I summarizes the prefetching parameters used throughout our experiments. These parameters were chosen through experimentation in an effort to maximize the performance of each scheme. All schemes effectively include next-2-line prefetching. (Although next-2-line prefetching was not in the original Markov prefetching design [Joseph and Grunwald 1997], we added it, since we found that it improves performance.) When a target is to be prefetched, we prefetch two consecutive lines starting at the target address.

Figure 1 shows the performance impact of each prefetching scheme on a collection of seven nonnumeric applications (which are discussed more in Section 5). We show three different versions of next-N-line prefetching (where $N = 2$, 4, and 8) in Figure 1, along with the original case without prefetching (**O**) and the case with a perfect instruction cache (**P**). Each bar represents execution time normalized to the case without prefetching, and is broken down into three categories corresponding to all potential graduation slots.[1] The bottom section (*Busy*) is the number of slots when instructions actually graduate. The top section (*I-Miss Stall*) is any nongraduating slots that would not occur with a perfect instruction cache, and the middle section (*Other Stall*) is all other slots where instructions do not graduate.

We observe from Figure 1 that despite significant differences in complexity and hardware cost, the various prefetching schemes offer remarkably similar performance, with no single scheme clearly dominating. Perhaps

[1]The number of graduation slots is the issue width (4 in this case) multiplied by the number of cycles. We focus on graduation rather than issue slots to avoid counting speculative operations that are squashed.

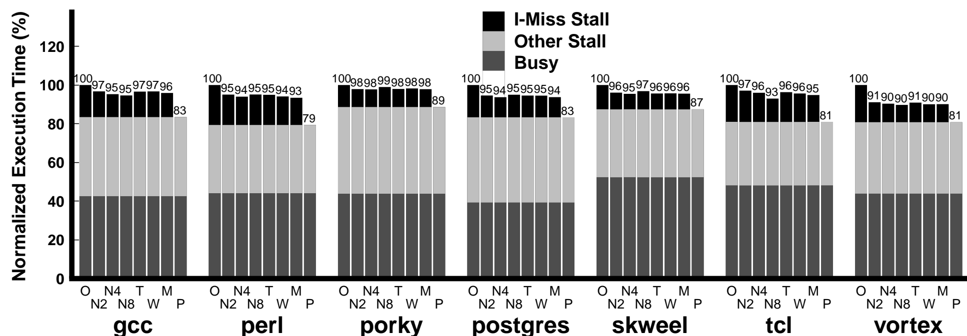


Fig. 1. Performance of existing instruction prefetching techniques (**O** = original, **N***x* = next-*x*-line prefetching, **T** = target-line prefetching, **W** = wrong-path prefetching, **M** = Markov prefetching, **P** = perfect instruction cache).

surprisingly, the best performance is achieved by either next-4-line or next-8-line prefetching in all cases except `perl`; even in `perl`, next-4-line prefetching is still within 1% of the best case. The reason for this is that the bulk of the benefit offered by each of these schemes is due to prefetching sequential accesses.

Finally, we see in Figure 1 that these schemes are hiding no more than half of the stall time due to instruction cache misses. Through a detailed analysis of why these schemes are not more successful (further details are presented later in Section 6.1), we observe, that although the coverage is generally quite high, the real problem is the *timeliness* of the prefetches—i.e., prefetches are not being launched early enough to hide the latency. Hence there is significant room for improvement over these existing schemes.

1.2 Our Solution

To hide instruction cache miss latency more effectively in modern microprocessors, we propose and evaluate a new fully automatic instruction prefetching scheme whereby the compiler and the hardware cooperate to launch prefetches earlier (therefore hiding more latency) while at the same time maintaining high coverage and actually *reducing* the impact of useless prefetches relative to today's schemes. Our approach involves two novel components. First, to enable more aggressive sequential prefetching without polluting the cache with useless prefetches, we introduce a new *prefetch filtering* hardware mechanism. Second, to enable more effective prefetching of nonsequential accesses, we introduce a novel compiler algorithm which inserts explicit *instruction-prefetch instructions* into the executable to prefetch the targets of control transfers far enough in advance. Our experimental results demonstrate that our scheme provides significant performance improvements over existing schemes, eliminating roughly 50% or more of the latency that had remained with the best existing scheme.

This paper is organized as follows. We begin in Section 2 with an overview of our approach, and then present further details on the architectural

and compiler support in Sections 3 and 4. Sections 5 and 6 present our experimental methodology and our experimental results, and finally we conclude in Section 7.

2. COOPERATIVE INSTRUCTION PREFETCHING

We begin this section with a high-level overview of our prefetching scheme. To make our approach concrete, we also present an example illustrating prefetch insertion.

2.1 Overview of the Prefetching Algorithm

As we mentioned earlier, the key challenge in designing a better instruction prefetching scheme is to be able to launch prefetches earlier—i.e., to achieve a larger *prefetching distance*. Let us consider the sequential and nonsequential portions of instruction streams separately.

2.1.1 *Prefetching Sequential Accesses*. Since the addresses within sequential access patterns are trivial to predict, they are well-suited to a purely hardware-based mechanism such as next-N-line prefetching. To get far enough ahead to fully hide the latency, we would like to choose a fairly large value for N (e.g., $N = 8$ in our experiments). However, the problem with this is that larger values of N increase the probability of overshooting the end of the sequence and polluting the cache with useless prefetches. For example, next-8-line prefetching performs worse than next-4-line prefetching for four cases in Figure 1 (`perl`, `porky`, `postgres`, and `skweel`) due to this effect.

The ideal solution would be to prefetch ahead aggressively (i.e., with a large N) but to stop upon reaching the end of the sequence. Xia and Torrellas [1996] proposed a mechanism for doing this which uses software to explicitly mark the likely end of a sequence with a special bit. In contrast, we achieve a similar effect using a more general *prefetch filtering* mechanism which automatically detects and discards useless prefetches before they can pollute the instruction cache. We will explain how the prefetch filter works in detail later in Section 3.3.1, but the basic idea is to use two-bit saturating counters stored in the secondary cache tags to dynamically detect cases where lines have been repeatedly prefetched into the primary instruction cache but were not accessed before they were displaced (i.e., *useless* prefetches). When prefetches for such lines subsequently arrive at the secondary cache, they are simply dropped. One advantage of our approach is that it adapts to the dynamic branching behavior of the program, rather than relying on static predictions of likely control flow paths. In addition, our filtering mechanism is equally applicable to *nonsequential* as well as sequential prefetches.

2.1.2 *Prefetching Nonsequential Accesses*. In contrast with sequential access patterns, purely hardware-based prefetching schemes are far less successful at prefetching *nonsequential* instruction accesses early enough. Wrong-path prefetching does not attempt to predict the target address of a

given branch early, but instead hopes that the same branch will be revisited sometime in the not-too-distant future with a different branch outcome. Both target-line and Markov prefetching rely on building up history tables to predict addresses to prefetch along control targets. However, if a control transfer is encountered for the first time or if its entry has been displaced from the finite history table, then its target will not be prefetched. (Note, that although our prefetch filtering mechanism can also potentially suffer from the limitations of learning within a finite table, we find it is far more important to prefetch target addresses early enough rather than filtering out all useless prefetches.) Perhaps more importantly, even if a valid entry is found in the history table, it is often too late to fully hide the latency of prefetching the target, since the processor is already accessing the line containing the branch.

To overcome these limitations, we rely on *software* rather than hardware to launch nonsequential instruction prefetches early enough. To avoid placing any burden on the programmer, we use the compiler to insert these new instruction-prefetching instructions automatically. As we describe in further detail later in Section 4, our compiler algorithm moves prefetches back by a specified prefetching distance while being careful not to insert prefetches that would be redundant with either next-N-line prefetching or other software instruction prefetches. Since many control transfers within procedures have targets within the N lines covered by our next-N-line prefetcher, the bulk of the instructions inserted by our compiler algorithm are for prefetching *across* procedure boundaries (as we show later in Section 5). Hence, although it is an oversimplification, one could think of our scheme as being primarily hardware-based for *intraprocedural* prefetching, and primarily software-based for *interprocedural* prefetching.

While direct control transfers (i.e., ones where the target address is statically known) are handled in a straightforward way by our algorithm, *indirect jumps* require some additional support. We consider two separate cases of indirect jumps: procedure returns, and all other indirect jumps. Since procedure return addresses can be easily predicted through the use of a *return address stack* [Webb 1988], we simply use a special prefetch instruction which implicitly uses the top of the return address stack as its argument. Although one could also imagine using the return address register as an explicit argument to the prefetch instruction, this may complicate the processor by creating a new datapath from the register file to the instruction fetcher. In general, we would like to avoid instruction-prefetch instructions that have register arguments.

To predict the target addresses of other indirect jumps, we use a hardware structure called an *indirect-target table* which records past target addresses of individual indirect jump instructions, and which is indexed using the instruction addresses of indirect jumps themselves. A prefetch instruction designed to prefetch the target of an indirect jump i conceptually stores the instruction address of i, which is then used to index the indirect-target table to retrieve the actual target addresses to prefetch.

Note that an indirect-target table is considerably smaller than the tables used by either target-line or Markov prefetching, since it only contains entries for active indirect jumps other than procedure returns. In recent work on indirect branch prediction [Chang et al. 1997; Driesen and Holzle 1998a; 1998b], the proposed techniques typically combine the address of the indirect branch with some history information to predict the *single* most likely target address. In contrast, our technique uses the indirect branch address to predict *multiple* target addresses. The difference lies in the fact that one and only one target must be selected in the case of branch prediction while in our case multiple targets can be prefetched simultaneously.

While the advantage of software-controlled instruction prefetching is that it gives us greater control over issuing prefetches early, the potential drawbacks are that it increases the code size and effectively reduces the instruction fetch bandwidth (since the prefetch instructions themselves consume part of the instruction stream). Fortunately, our experimental results demonstrate that this advantage outweighs any disadvantages.

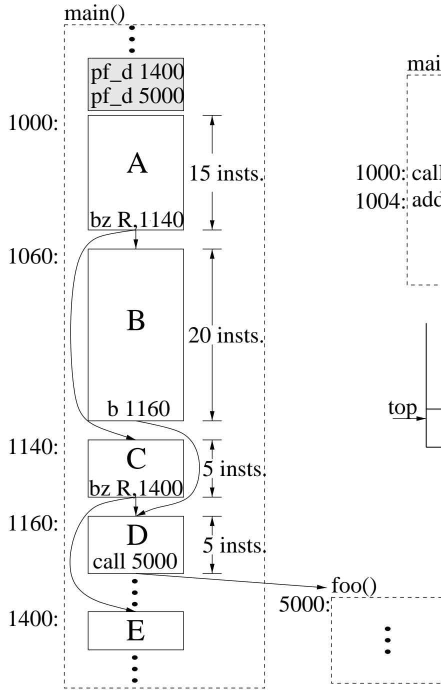
2.2 Example of Prefetch Insertion

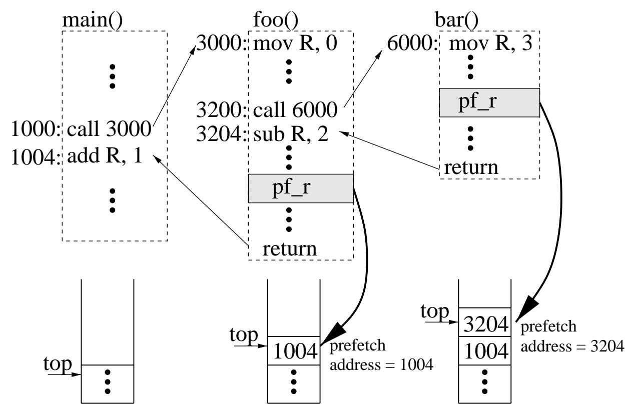
To make our discussion more concrete, Figure 2 contains three examples of how different types of prefetches are inserted. We assume the following in these examples: a cache line is 32 bytes long; an instruction is four bytes long (hence one cache line contains eight instructions); hardware next-8-line prefetching is enabled; and the prefetching distance is 20 instructions.

Figure 2(a) shows two procedures, `main()` and `foo()`, where `main()` contains five basic blocks (labeled `A` through `E`). Two prefetches have been inserted at the beginning of basic block `A`: one targeting block `E`, and the other targeting procedure `foo()`. There is no need to insert software prefetches for blocks `B`, `C` or `D` at `A`, since they will already be handled by next-8-line prefetching. The prefetch targeting `E` is inserted in block `A` rather than in block `C` in order to guarantee a prefetching distance of at least 20 instructions. Although there are two possible paths from `A` to `foo()` (i.e., `A` → `B` → `D` → `foo( )` and `A` → `C` → `D` → `foo( )`), the compiler inserts only a single prefetch of `foo()` in `A` (rather than inserting one in `A` and one in `B`) because (i) `A` *dominates*[2] both paths, and (ii) the compiler determines that these prefetched instructions are not likely to be displaced by other instructions fetched along the path `A` → `B` → `D` → `foo( )`.

Figure 2(b) shows an example of prefetching return addresses. The prefetches in procedures `bar()` and `foo()` get their addresses from the top of the return address stack—i.e., `3204` and `1004`, respectively. Finally, Figure 2(c) shows an example where a prefetch is inserted to prefetch the target address of the indirect jump at address `8192` before the actual target address is known (i.e., the value register R has not been determined yet).

[2]Node d of a flow graph dominates node n if every path from the initial node of the flow graph to n goes through d [Aho et al. 1986].





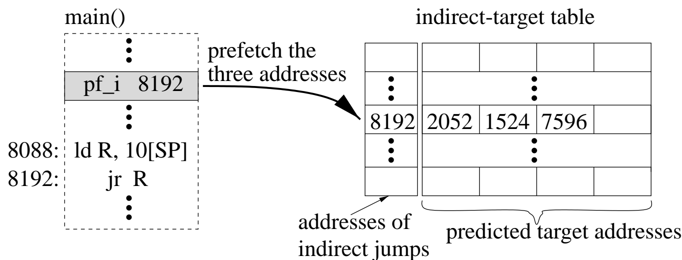


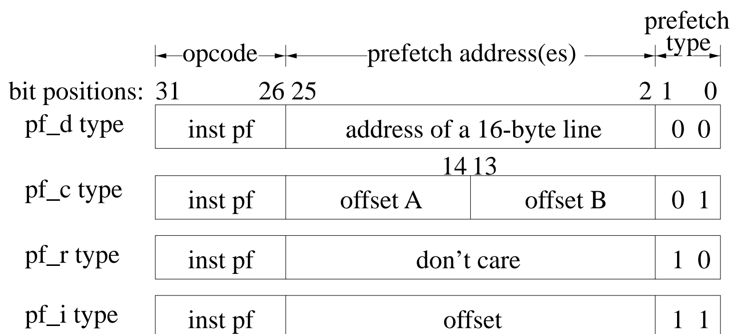
Fig. 2. Examples of prefetch insertion for different types of target addresses. (`pf_d` = prefetch a direct address, `pf_r` = prefetch a return address, `pf_i` = prefetch an indirect-jump target address).

Hence the prefetch has `8192` as its address operand to serve as an index into the indirect-target table. Three target addresses are predicted for this indirect jump, and all of them will be prefetched.

3. ARCHITECTURAL SUPPORT

Our prefetching scheme requires new architectural support. In this section, we describe our extensions to the instruction set, how these new instructions affect the pipeline, and the new hardware that we add to the memory system (including the prefetch filter).

(a) Adding instruction prefetches to the ISA



(b) Data vs. instruction prefetch pipelines

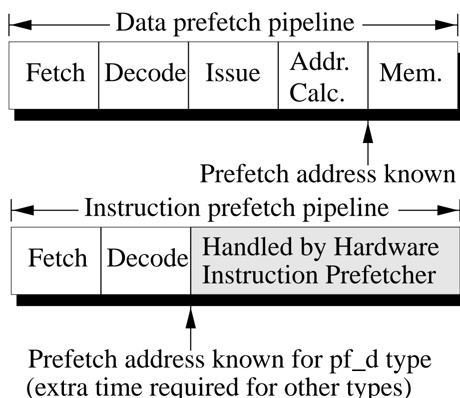


Fig. 3. Possible extensions to the ISA and the CPU pipeline for instruction prefetches.

3.1 Extensions to the Instruction Set Architecture

Without loss of generality, we assume a base instruction set architecture (ISA) similar to the MIPS ISA [Kane and Heinrich 1992]. Within a 32-bit MIPS instruction, the high-order six bits contain the opcode. For the jump-type instructions which implement static procedure calls, the remaining 26 bits contain the low-order bits of the target word address. We will use this same instruction format as our starting point.

There are many ways to encode our new instruction-prefetch instructions, and Figure 3(a) shows just one of the possibilities. An opcode is designated to identify instruction-prefetch instructions. In contrast with the standard jump-type instruction format, we assume that 24 bits (bits 2 through 25) contain information for computing the prefetch address(es); bits 1 and 0 indicate one of the four prefetch types. The prefetch type `pf_d` stores a single prefetch address in a format similar to a MIPS jump address. The only difference is, that since the lower two bits are ignored, it effectively encodes a 16-byte-aligned address. (Since most machines have at

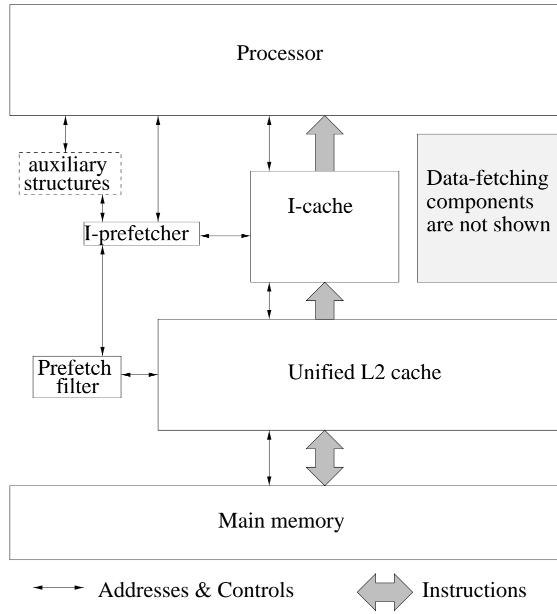
least 16−byte instruction lines, this is not a limitation.) The `pf_c` type is a *compact* format which encodes two target addresses within the 24-bit field in the form of offsets between the target address lines and the prefetch instruction line itself (again, a single offset bit represents 16 bytes); each offset is 12 bits wide. The remaining two types are for prefetching indirect targets—`pf_r` is for procedure returns, and `pf_i` is for general indirect-jump targets. A `pf_r` prefetch does not require an argument, since it implicitly uses the top of the return address stack as its address. A `pf_i` prefetch encodes the word offset between itself and the indirect-jump instruction that it is prefetching. To look up the prefetch address(es), this offset is added to the current program counter to create an index into the indirect-target table.

3.2 Impact on the Processor Pipeline

Many recent processors have implemented instructions for data prefetching [Bernstein et al. 1995; Santhanam et al. 1997; Yeager 1996]. With respect to pipelining, our *instruction* prefetches differ in two important ways from data prefetches: (i) the pipeline stage in which the prefetch address is known, and (ii) the computational resources consumed by the prefetches. Figure 3(b) contrasts the pipeline for data prefetches in the MIPS R10000 [Yeager 1996] with the pipeline for our instruction prefetches in an equivalent machine. As we see in Figure 3(b), the prefetch address of a `pf_d` instruction prefetch is known immediately after the *Decode* stage (the other three prefetch types would require some additional time), while the address for a data prefetch is not known until it is computed in the *Address Calculate* stage. Hence a `pf_d` instruction prefetch can be initiated two cycles earlier than a data prefetch. In addition, since instruction prefetches do not go through the latter three pipeline stages of a data prefetch (instead they are handled directly by the hardware instruction prefetcher after they are decoded), they do not contend for processor resources including functional units, the reorder buffer, register file, etc. In effect, the instruction prefetches are removed from the instruction stream as soon as they are decoded, thereby having minimal impact on most computational resources.

3.3 Extensions to the Memory Subsystem

Figure 4(a) shows our memory subsystem (only the instruction fetching components are displayed). The *I-prefetcher* is responsible for generating prefetch addresses and launching prefetches to the unified L2 cache for both hardware- and software-initiated prefetching. Prefetch-address generation involves simple extraction of prefetch addresses from `pf_d` prefetches, adding constant offsets to the current program counter (for next-N-line prefetching and `pf_c` prefetches), or retrieving prefetch targets from some hardware structures (for `pf_r` and `pf_i` prefetches). The I-prefetcher will not launch a prefetch to the L2 cache if the line being prefetched is already in the primary instruction cache (*I-cache*) or has an



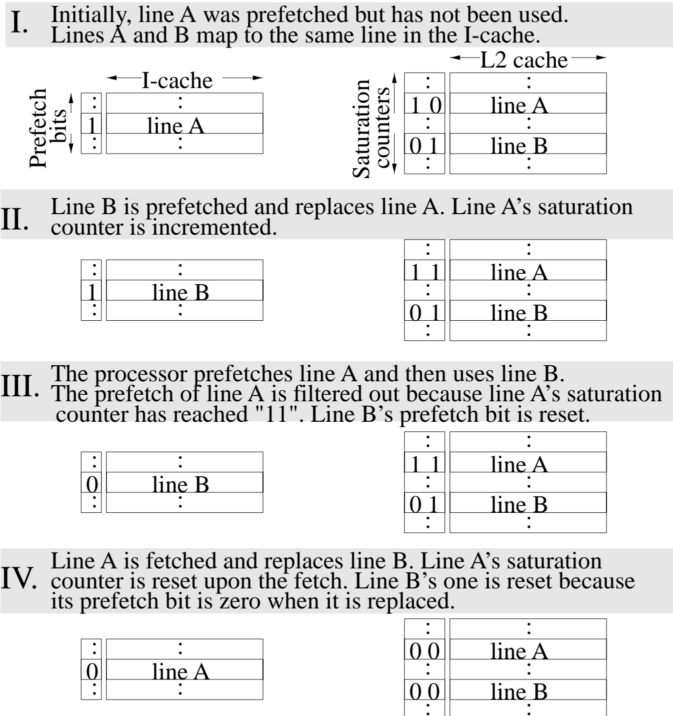


Fig. 4. The memory subsystem and an example of the prefetch filtering mechanism.

outstanding fetch or prefetch for the same line address. The *auxiliary structures* shown in Figure 4(a) include the return address stack and the indirect-target table used by `pf_r` and `pf_i` prefetches, respectively. These structures are not necessary if these two types of prefetches are not implemented.

3.3.1 *Prefetch Filtering Mechanism*. The *prefetch filter* sits between the I-prefetcher and the L2 cache to reduce the number of useless prefetches. In addition, a *prefetch bit* is associated with each line in the I-cache to remember whether the line was prefetched but not yet used, and a two-bit saturating counter value is associated with each line in the L2 cache to record the number of *consecutive* times that the line was prefetched but not used before it was replaced. In essence, our prefetch filter is a specific form of confidence counters [Grunwald et al. 1998] that assess the quality of prefetching. The width of these counters was selected through experimentation.

The prefetch filtering mechanism works as follows. When a line is *fetched* from the L2 cache to the I-cache, both the prefetch bit and the saturating counter value are reset to zero. When a line is *prefetched* from the L2 cache to the I-cache, its prefetch bit is *set* to one, and its saturation counter does not change. When a prefetched line is actually used by a fetch, its prefetch bit is *reset* to zero. When a prefetched line l in the I-cache is replaced by another line, then if the prefetch bit of line l is set, its saturation counter is incremented (unless it has already saturated, of course); otherwise, the counter is reset to zero. When the prefetch filter receives a prefetch request for line l, it will either respond normally if the counter value is below a threshold T, or else it will drop the prefetch and send a "prefetch canceled" signal to the processor if the counter has reached T (in our experiments, $T = 3$). Figure 4(b) shows an example of how the prefetch filtering mechanism works, and Figure 5 summarizes the states and transitions of the prefetch bit and the saturation counter for a particular cache line.

4. COMPILER SUPPORT

The compiler is responsible for automatically inserting prefetch instructions into the executable. Since prefetch insertion is most effective if it begins after the code is otherwise in its final form, this new pass occurs fairly late in the compilation: perhaps at link time, or as in our case, implemented as a binary rewrite tool. Further, the compiler should schedule prefetches so as to achieve high coverage and satisfactory prefetching distances while minimizing the static and dynamic instruction overhead. Hence our compiler algorithm has two major phases, *prefetch scheduling* and *prefetch optimization*, which are described in the following subsections. In addition, the compiler also determines how many lines should be brought into the cache by a prefetch. A complete implementation of this algorithm was used throughout our experiments.

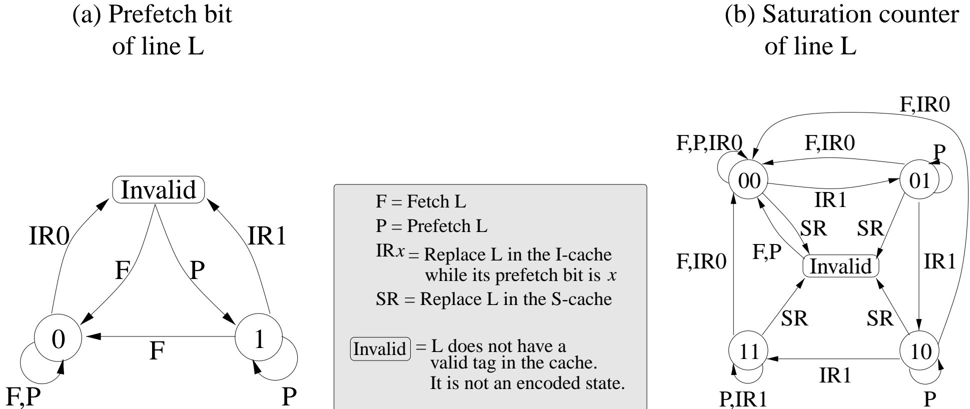


Fig. 5. The states and transitions of (a) prefetch bits and (b) saturation counters under prefetch filtering.

4.1 Preprocessing

This step prepares the information that will be used by the scheduling and the optimization phases. One important task here is to identify the interesting control structures in the input executable, in particular *loops*. We used the algorithm given in Aho et al. [1986, Section 10.4] to find all the natural loops. This algorithm requires us to first compute the *dominators*. In addition to this use, dominator information is also needed in other parts of our compiler algorithm.

4.2 Prefetch Scheduling

Figure 6 shows our main prefetch scheduling algorithm `Schedule_Prefetches`, which inserts direct and indirect prefetches into a given executable using two similar algorithms: `Schedule_Direct_Prefetches` and `Schedule_Indirect_Prefetches`.

4.2.1 *Schedule_Direct_Prefetches.* Algorithm `Schedule_Direct_Prefetches` (also shown in Figure 6) attempts to move prefetches for a given target basic block back by a distance of at least *PF_DIST* instructions. `Schedule_Direct_Prefetches` also tries to avoid inserting unnecessary prefetches by checking whether another prefetch with the same target already exists. This check is made by algorithm `Hardware_Prefetched`, which determines if the target basic block is covered by hardware sequential prefetching, and by algorithm `Software_Prefetched`, which tests whether an identical software prefetch is present in the same block. Algorithm `Locality_Likely`, which we will discuss shortly, checks whether the target basic block is likely to be already in the I-cache, in which case no prefetch is required. Also, as implied by the control flow of `Schedule_Direct_Prefetches`, if `prefetched` is *False* but `Locality_Likely` is *True*, we will keep attempting to insert prefetches further back, since, even though locality exists *within* a particular loop, we

```
// This algorithm schedules prefetches for every basic block in the input executable.
algorithm Schedule_Prefetches
            (E: executable) // the input executable
    return ();              // nothing to return

  foreach B in E do
    Schedule_Direct_Prefetches(B, B, 0, {}); // schedule direct prefetches for B
  end foreach;
  foreach B in E do
    if (B ends with an indirect jump or a procedure return) then
      // Schedule_Indirect_Prefetches() (code shown in Figure 7) inserts prefetches
      // for the target of the indirect control transfer located in B.
      Schedule_Indirect_Prefetches(B, B, Instruction_Count(B.instructions), {});
    end if;
  end foreach;
end algorithm;

// This algorithm inserts direct prefetches targeting a given basic block T.
algorithm Schedule_Direct_Prefetches
            (B: basic block,             // current basic block
             T: basic block,             // prefetch-target basic block
             D: integer,                 // the prefetching distance between B and T
             S: set of basic blocks)     // basic blocks considered so far
    return (); // nothing to return

  if (B ∉ S) then // continue only if B hasn't been considered
    S := S ∪ {B}; // B is now being considered
    // First, determine if a prefetch for T, either launched by hardware or software,
    // is already present in B.
    prefetched: boolean := Hardware_Prefetched(B, T) or Software_Prefetched(B, T);
    if (not prefetched) then
      // not prefetched yet, attempt to insert a prefetch at B if it is sufficiently early
      // and necessary
      if (D ≥ PF_DIST and not Locality_Likely(B, T)) then
        // PF_DIST is the desirable prefetching distance.
        // Locality_Likely() determines the likelihood that B and T coexist in the cache.
        Attach_Direct_Prefetch(B, T); // attach a direct prefetch for T onto B
        prefetched := True; // a prefetch for T is just inserted into B
      end if;
    end if;
    if (not prefetched) then
      // still not prefetched yet, attempt to insert prefetches at the predecessors of B
      foreach B's predecessor block P do
        increment: integer;
        if (B is the fall-through block of P and P ends with a procedure
            call (static or dynamic)) then
          // also count the length of the called procedure
          increment := Shortest_Path(P.instructions);
        else // P reaches B via a static control transfer or a fall-through path
             // that does not end with a procedure call
          increment := Instruction_Count(P.instructions);
        end if;
        D' := D + increment; // update prefetching distance conservatively
        Schedule_Direct_Prefetches(P, T, D', S); // consider P next
      end foreach;
    end if;
  end if;
end algorithm;
```

Fig. 6. Main prefetch scheduling algorithm and the algorithm for scheduling direct prefetches.

```
// This algorithm inserts prefetches targeting the indirect control transfer located in
// a given basic block T.
algorithm Schedule_Indirect_Prefetches
            (B: basic block,          // current basic block
             T: basic block,          // the block that contains the indirect control transfer
             D: integer,              // the prefetching distance between B and
                                      // the target of the indirect control transfer
             S: set of basic blocks)  // basic blocks considered so far
    return (); // nothing to return

  if (B ∉ S) then // continue only if B hasn't been considered
    S := S ∪ {B}; // B is now being considered
    // First, determine if there is another indirect software prefetch for the same target in B.
    prefetched: boolean := Software_Prefetched(B, T);
    if (not prefetched) then
      // not prefetched yet, attempt to insert a prefetch at B if it is sufficiently early
      if (D ≥ PF_DIST) then
        // PF_DIST is the desirable prefetching distance.
        Attach_Indirect_Prefetch(B, T); // attach a prefetch targeting the indirect
                                        // control transfer
        prefetched := True; // a prefetch is just inserted into B
      end if;
    end if;
    if (not prefetched) then
      // still not prefetched yet, attempt to insert prefetches at the predecessors of B
      foreach B's predecessor basic block P do
        increment: integer;
        if (B is the fall-through block of P and P ends with a procedure call
          (static or dynamic)) then
          increment := Shortest_Path(P.instructions);
        else // P reaches B via a static control transfer or a fall-through that does not
             // end with a procedure call.
          increment := Instruction_Count(P.instructions);
        end if;
        D' := D + increment; // update prefetching distance conservatively
        Schedule_Indirect_Prefetches(P, T, D', S); // consider P then
      end foreach;
    end if;
  end if;
end algorithm;
```

Fig. 7. Algorithm for scheduling indirect prefetches.

still need to prefetch *before* entering that loop. Note that when we consider inserting prefetches into a predecessor block, the prefetching distance must be updated. If there is a procedure invocation between the predecessor block and the current block, the new prefetching distance should include the length of the called procedure, which is estimated by `Shortest_Path`; otherwise, the new prefetching distance is simply obtained by adding the instruction count of the predecessor block to the old one. Finally, `Schedule_Direct_Prefetches` *terminates* along a particular path in one of the following four situations: (i) a prefetch is successfully inserted on that path; (ii) prefetching is discovered to be unnecessary for that path; (iii) there are

```
// This algorithm determines the likelihood that two given basic blocks A and B coexist in
// the I-cache.
algorithm Locality_Likely
            (A: basic block,
             B: basic block)
    return (boolean); // true if A and B are likely to exist in the I-cache simultaneously

  L_A: set of loops := My_Loops(A);      // find the set of loops containing A
  L_B: set of loops := My_Loops(B);      // find the set of loops containing B
  L_common: set of loops := L_A ∩ L_B;  // find the set of loops that are common to A and B
  foreach l ∈ L_common do
     if (Largest_Volume(l.body, True) < Effective_CacheSize) then
        // Largest_Volume() computes the largest possible volume of instructions to be fetched.
        // Effective_CacheSize is the effective I-cache size, after taking other factors such as
        // cache conflicts into account.
        // A and B are likely to coexist in the I-cache if their common loop can be entirely
        // held into the cache.
        return True;
     end if;
  end foreach;
  return False;
end algorithm;
```

Fig. 8. Algorithm for determining the likelihood of two basic blocks existing in the cache simultaneously.

no more predecessors to move further back; or (iv) a basic block is considered again to insert a prefetch (i.e., the path constitutes a cycle).

4.2.2 *Schedule_Indirect_Prefetches*. Working in a similar fashion to `Schedule_Direct_Prefetches`, `Schedule_Indirect_Prefetches` (shown in Figure 7) inserts prefetches for indirect jumps and procedure returns. However, unlike `Schedule_Direct_Prefetches`, the second input parameter of `Schedule_Indirect_Prefetches` is the basic block containing the indirect control transfer instead of the target basic block itself, since the target basic block is unknown statically. Therefore, when we pass this basic block as the first basic block to consider (i.e., how `Schedule_Indirect_Prefetches` is invoked in `Schedule_Prefetches`), the initial prefetching distance is not zero but the length of this basic block. Another difference between `Schedule_Direct_Prefetches` and `Schedule_Indirect_Prefetches` is that `Hardware_Prefetched` and `Locality_Likely` are not called in `Schedule_Indirect_Prefetches`, since they cannot get much information without knowing the exact prefetch target address. While `Schedule_Indirect_Prefetches` handles both indirect jumps and procedure returns in mostly the same way, it has to make sure that prefetches for a return located in a procedure—say, P—will not be issued prior to the invocation of P. This is accomplished by the `Attach_Indirect_Prefetch` routine.

4.2.3 *Locality_Likely*. Given two basic blocks, the algorithm shown in Figure 8 estimates how much the presence of one block in the I-cache implies the other's. The likelihood of both appearing would be high if all the

```
// This algorithm computes the shortest path through an instruction sequence.
algorithm Shortest_Path
            (I: list of instructions) // instructions at a given control-flow graph level
      return (shortestLength: integer);

   shortestLength: integer := 0;
   foreach i ∈ I do
      if (i is a conditional branch) then
         // pick the one with shorter path length
         shortestLength := shortestLength + min(Shortest_Path(i.then_part),
                                                Shortest_Path(i.else_part));
      else if (i is an unconditional branch) then
         shortestLength := shortestLength + Shortest_Path(i.target_part);
         // instructions following i won't be executed after the branch and hence they
         // shouldn't be counted then
         return (shortestLength);
      else if (i is a loop) then
         if (isConstant(i.num_iterations)) then
            shortestLength := shortestLength + i.num_iterations * Shortest_Path(i.loop_body);
         else // assume at least one iteration when iteration count is unknown
            shortestLength := shortestLength + Shortest_Path(i.loop_body);
         end if;
      else if (i is a static, non-recursive procedure call) then
         shortestLength := shortestLength + Shortest_Path(i.procedure_body);
      else if (i is a dynamic procedure call) then
         // assume small length for dynamic procedure calls
         shortestLength := shortestLength + Small_ProcedureLength;
      end if;
      shortestLength := shortestLength + 1;
   end foreach;
   return (shortestLength);
end algorithm;
```

Fig. 9. Algorithm for computing the shortest path through an instruction sequence.

following three conditions are true: (i) both blocks belong to a common loop; (ii) both blocks are fetched or prefetched before entering the loop; and (iii) the entire loop (of course including the two blocks in question) will stay in the I-cache from the first to the last loop iteration. Condition (ii) is ensured automatically by the caller of `Locality_Likely` (e.g., `Schedule_Direct_Prefetches`), and therefore we only need to check conditions (i) and (iii) in `Locality_Likely`. Condition (i) is checked by finding the sets of loops that contain the two blocks and then intersecting them. For condition (iii), we estimate it to be true if the largest possible volume of instructions to be fetched by the loop is smaller than the "effective" size of the I-cache (the effective I-cache size is only a fraction of the actual I-cache capacity in order to account for effects due to cache conflicts and instruction alignment, etc.). In our implementation, the effective cache size is one-eighth of the actual size. Similar to this test for locality within loops, our implementation also includes a test for locality within *recursive procedure calls*.

4.2.4 *Shortest_Path*. Algorithm `Shortest_Path` in Figure 9 estimates the minimum number of instructions to be executed for a given control-flow

```
// This algorithm computes the largest possible volume of instructions fetched in an
// given instruction sequence.
algorithm Largest_Volume
            (I: list of instructions // instructions at a given control-flow graph level
             combine: boolean)       // whether need to accumulate the volume of
                                     // instructions on disjoint paths
    return (largestVolume: integer);

  largestVolume: integer := 0;
  foreach i ∈ I do
    if (i is a conditional branch) then
      if (combine) then
        // accumulate the volume of both paths
        largestVolume := largestVolume + Largest_Volume(i.then_part, combine)
                         + Largest_Volume(i.else_part, combine);
      else
        // pick the one with larger volume
        largestVolume := largestVolume + max(Largest_Volume(i.then_part, combine),
                                             Largest_Volume(i.else_part, combine));
      end if;
    else if (i is an unconditional branch) then
      largestVolume := largestVolume + Largest_Volume(i.target_part, combine);
      // instructions following i won't be executed after the branch and hence they
      // shouldn't be counted then
      return (largestVolume);
    else if (i is a loop) then
      // conservatively assume that all instructions in the loop are used (in different iterations)
      largestVolume := largestVolume + Largest_Volume(i.loop_body, True);
    else if (i is a static, non-recursive procedure call) then
      largestVolume := largestVolume + Largest_Volume(i.procedure_body, combine);
    else if (i is a dynamic procedure call) then
      // assume large volume for dynamic procedure calls
      largestVolume := largestVolume + Large_ProcedureVolume;
    end if;
    largestVolume := largestVolume + 1;
  end foreach;
  return (largestVolume);
end algorithm;
```

Fig. 10. Algorithm for computing the largest possible volume of instructions accessed.

structure. The interesting cases in this algorithm are: (i) conditional branches, where we choose the shorter of the "*then*" and "*else*" paths; (ii) unconditional branches, where we return the length of the target paths; (iii) loops, where we use the iteration count if it is known and otherwise assume that at least a single iteration is executed; (iv) static procedure calls, where we use the length of the procedure body, unless there is recursion; and (v) dynamic procedure calls, where we conservatively assume unknown procedure length to be small (*Small_ProcedureLength* = 10 in our implementation).

4.2.5 *Largest_Volume*. This algorithm estimates the largest possible volume of instructions to be fetched in a given instruction sequence. As shown in Figure 10, `Largest_Volume` has a similar structure to `Shortest_Path`. The major difference of them is that while `Shortest_Path` computes

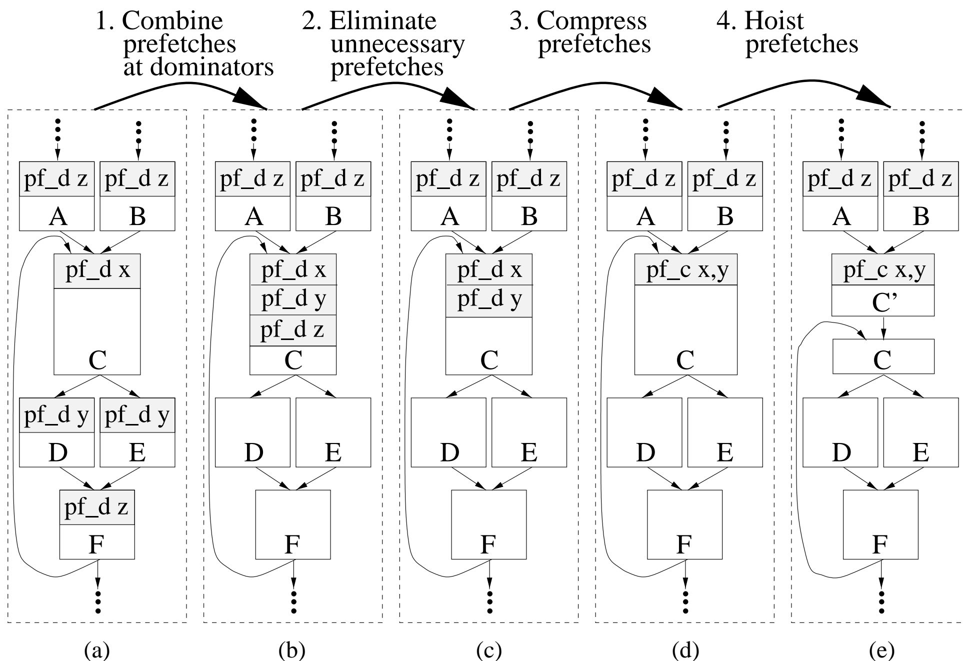


Fig. 11. Example of prefetch optimization. A to F are basic blocks; x, y and z are cache line addresses. C is a dominator of D, E, F, and C itself. Part (a) is the initial schedule, and part (e) is the final optimized schedule.

in a *minimal* notion, Largest_Volume does in a *maximal* one. As a result, Largest_Volume needs an additional parameter combine to indicate whether we should sum up instructions accessed in *disjoint* paths or simply select the path accessing more instructions; the former case is used to conservatively count all possible instructions accessed by a loop. Note that when we encounter a dynamic procedure call, we assume a fairly large procedure body ($Large_ProcedureVolume = 1024$ in our implementation).

4.3 Prefetch Optimization

After generating an initial prefetch schedule, the compiler then performs a number of optimizations attempting to minimize both static and dynamic prefetching overheads. We devised our optimizations in light of existing compiler optimizations [Aho et al. 1986], including *code motion*, *common-subexpression elimination*, and *extraction of loop invariants*. However, our optimizations differ from them in two aspects. First, we need to take *locality* information into account to make sure that these optimizations would not degrade the effectiveness of prefetching. Second, since prefetches are mainly performance hints, we can simply ignore the information that we do not know statically (e.g., indirect jump targets, mapping to cache sets); the worst case is to affect performance but not break the program, as

it would in classical compiler optimizations. Our optimization algorithm is composed of the following four passes. To make our discussion more concrete, we use the example in Figure 11 to help explain these passes.

4.3.1 *Pass 1: Combining Prefetches at Dominators.* This pass boosts prefetches that have been attached to a basic block b in the prefetch scheduling phase to b's nearest dominator (other than b itself) if the boosting is not considered to be harmful (it is so if the volume of instructions accessed between the dominator and the prefetch target, as estimated by the algorithm shown in Figure 10, exceeds the effective cache size. In that case, the boosted prefetches may displace other useful instructions from the cache before the prefetch target is reached). After this boosting process, the compiler could combine redundant prefetches at dominators. For example, Figure 11(b) shows the result of combining the two prefetches of line `y` into one after boosting prefetches from basic blocks `D`, `E`, and `F` into their dominator `C`.

4.3.2 *Pass 2: Eliminating Unnecessary Prefetches.* A prefetch instruction u targeting a line l is *unnecessary* if l resides in the I-cache on *all* possible paths reaching u. To eliminate unnecessary prefetch instructions, we devise a data-flow analysis algorithm to estimate which instruction lines reside in the I-cache at a particular program point. This algorithm, called `Compute_Cached_Instructions`, resembles the one that computes *available expressions* in classical compiler optimizations [Aho et al. 1986]. It is shown in Figure 12.

Algorithm `Compute_Cached_Instructions` performs a *forward* data-flow analysis. For a given block B other than the initial block B_0, $in[B]$ obtains its value by intersecting $out[P]$'s for all predecessors P of B. We can then compute $out[B]$ by uniting $use[B]$, $pf[B]$, and $(in[B] - displace[B])$. This union captures the notion that the instruction lines that reside in the cache upon exiting a block are equal to the lines resident upon entering the block plus those that are newly brought into the cache by the block (i.e., $use[B] \cup pf[B]$) minus those that are displaced. The set $displace[B]$ is initialized using `Conflicting_Instructions`, which estimates the instruction lines that could possibly conflict with $use[B] \cup pf[B]$. Like many other iterative data-flow analyses, `Compute_Cached_Instructions` repeats computing $in[]$ and $out[]$ until $out[]$ remains the same in two consecutive iterations.

In the example shown in Figure 11(b), we assume, that once line `z` is prefetched before entering the loop, it will stay in the cache for the whole execution of the loop. Then line `z` will definitely be in the I-cache when we enter basic block `C` regardless of whether we came from `A`, `B`, or `F`. Therefore, the prefetch of line `z` in `C` is unnecessary and can be eliminated, as shown in Figure 11(c).

4.3.3 *Pass 3: Compressing Prefetches.* The compiler checks whether multiple `pf_d` prefetches in the same basic block can be compressed into a

```
// This algorithm estimates the instruction lines that reside in the I-cache at each basic block
// of the input executable.
// Explanation for the data-flow analysis variables used in this algorithms:
//          in[B] = the set of instruction lines residing in the I-cache upon entering block B
//          out[B] = the set of instruction lines residing in the I-cache upon exiting block B
//          use[B] = the set of instruction lines used in block B
//          pf[B] = the set of instruction lines prefetched in block B
//          displace[B] = the set of instruction lines displaced from the I-cache in block B

algorithm Compute_Cached_Instructions
          (E: executable)    // the input executable
          (B0: basic block)  // the initial block
    return (in); // in[B] for each block B

  in, out, use, pf, displace: array of set of instruction lines;
  // some initialization follows
  in[B0] := {};
  use[B0] := the set of instruction lines used in B0;
  pf[B0] := the set of instruction lines prefetched in B0;
  out[B0] := use[B0] ∪ pf[B0]; // in[B0] and out[B0] will never change then
  foreach B ≠ B0 in E do
    use[B] := the set of instruction lines used in B;
    pf[B] := the set of instruction lines prefetched in B;
    // Find out which instruction lines in U could be conflicting with those that are used
    // or prefetched in B, where U includes all instruction lines in E.
    displace[B] := Conflicting_Instructions(U, use[B] ∪ pf[B]);
    // assume that all instruction lines other than those that are displaced are cached initially
    out[B] := U - displace[B];
  end foreach;
  change: boolean := True;
  while change do // iterate until converged
    change := False;
    foreach B ≠ B0 in E do
      // We are sure that an instruction line resides in the cache upon entering B if it
      // does upon leaving every predecessor of B.
      in[B] :=        ∩            out[P];
              P a predecessor of B
      oldout: set of instruction lines := out[B];
      out[B] := use[B] ∪ pf[B] ∪ (in[B] − displace[B]);
      if (out[B] ≠ oldout) then
        change := True;
      end if;
    end foreach;
  end while;
  return (in);
end algorithm;
```

Fig. 12. Our data-flow analysis algorithm for estimating which instruction lines reside in the I-cache.

single compact prefetch. For each basic block b, the compiler needs to compute the offsets between the starting address of b and the target addresses of all `pf_d` prefetches scheduled in b. It then attempts to fit these offsets into a minimum number of compact prefetch instructions. Our example assumes that the address offsets of both lines `x` and `y` are

representable within 12 bits, and therefore the two `pf_d` prefetches in `C` are compressed into a single `pf_c` prefetch, as shown in Figure 11(d).

4.3.4 *Pass 4: Hoisting Prefetches*. Finally, the compiler hoists prefetches scheduled inside a loop up to the nearest basic block that dominates but is not part of the loop, if the prefetches do not need to be reexecuted at every iteration (which may not be the case if each iteration can access a large volume of instructions). If such a block is not found (most probably because these outside-loop dominators are too far away from the loop), a *preheader* block will be created for the loop to hold the hoisted prefetches. For example, in Figure 11(e), a preheader `C'` is created to immediately precede the header (i.e., `C`) of the loop containing `C`, `D`, `E`, and `F` to hold the hoisted `pf_c` prefetch. While this optimization does not reduce the code size, it can reduce the number of *dynamic* prefetches.

4.4 Determining the Software Prefetch Size

Since instruction accesses are fairly sequential, it is usually helpful to bring in more than one cache line by a software prefetch. Currently, our compiler decides a constant prefetch size for all software prefetches and passes it to the hardware prior to program execution. If hardware prefetch filtering is enabled, the software prefetch size is set to four; otherwise, it is set to two. Alternatively, the compiler can use variable prefetch sizes. However, this would require extra operands or more complex encoding in instruction-prefetch instructions in order to specify the prefetch size.

5. EXPERIMENTAL FRAMEWORK

We performed our experiments on seven nonnumeric applications which were chosen because their relatively large instruction footprints result in poor instruction cache performance. These applications are described Table II, and all of them were run to completion. Table III shows the number of software prefetches inserted into the executable, broken down into the interprocedural and intraprocedural cases. As we see in Table III, the software component of our scheme mainly targets interprocedural prefetching.

We performed detailed simulations of our applications on a dynamically scheduled, superscalar processor similar to the MIPS R10000 [Yeager 1996]. Table IV shows the parameters used in our model for the bulk of our experiments (we vary the latency and bandwidth later in Section 6.7). As shown in Table IV, we enhanced the memory subsystem in a few ways relative to the R10000 to provide better support for instruction prefetching—e.g., we added an eight-entry victim cache [Jouppi 1990] and a 16-entry prefetch buffer [Joseph and Grunwald 1997]. We considered using more sophisticated branch predictors than the 2-bit predictors in the R10000; however, we found, that when we performed the same experiments with even a *perfect* branch predictor, it did not affect our conclusions. Hence, we decided to use the R10000's branch prediction scheme throughout our experiments.

Table II. Application Characteristics. Note the "combined" miss rate is the fraction of instruction fetches which suffer misses in both the 32KB I-cache *and* the 1MB L2 cache. All programs were run to completion.

Name	Description	Input Data Set	Instructions Graduated	Miss Rate	
				I-Cache	Combined
Gcc	The GNU C compiler drawn from SPEC92	The stmt.i in the reference input set	136.0M	2.63%	0.10%
Perl	The interpreter of the Perl language drawn from SPEC95	A Perl script a2ps.pl which converts ascii to postscript	41.4M	5.03%	0.06%
Porky	A SUIF compiler pass for simplifying and rearranging codes	The compress95.c program in SPEC95 (default optimizations)	86.8M	2.38%	0.06%
Postgres	The PostgreSQL database management system [Yu and Chen 1996]	A subset of queries in the Postgres Wisconsin benchmark	46.0M	3.76%	0.16%
Skweel	A SUIF compiler pass for loop parallelization	A program that computes Simplex (with all optimizations)	68.1M	2.22%	0.06%
Tcl	An interpreter of the script language Tcl version 7.6	Tcltags.tcl which makes Emacs-style TAGS file for Tcl source	37.5M	2.78%	0.02%
Vortex	The Vortex object-oriented database program in SPEC95	A reduced SPEC95 input set	193.0M	6.48%	0.08%

We compiled each application as a "nonshared" executable with `-O2` optimization using the standard MIPS C compilers under IRIX 5.3. We implemented our compiler algorithm as a standalone pass which reads in the MIPS executable and modifies the binary. However, since we did not have access to a complete set of binary rewrite utilities, we tightly integrated our compiler pass with our simulator so that rather than physically generating a new executable, we instead pass a logical representation of the new binary to the simulator which it can then model accurately. For example, the simulator fetches and executes all of the new instruction prefetches as though they were in a real binary, and it remaps all

Table III. Number of Software Prefetches Inserted into the Executable. Note that the "static prefetch count" is normalized to the size of the original executable. Prefetches are classified as either *interprocedural* or *intraprocedural*, depending on whether the prefetch target and the prefetch itself are in the same procedure.

Name	Static Prefetch Count (% of Original Executable Size)	
	Interprocedural	Intraprocedural
Gcc	6.3%	1.6%
Perl	8.3%	1.7%
Porky	7.1%	0.4%
Postgres	8.3%	0.6%
Skweel	7.5%	0.6%
Tcl	7.2%	1.0%
Vortex	10.3%	0.7%

Table IV. Simulation Parameters for the Baseline Architecture

Pipeline Parameters	
Fetch AND Decode Width	8 aligned sequential instructions
Issue AND Graduate Width	4
Functional Units	2 Integer, 2 FP, 2 Memory, 2 Branch
Reorder Buffer Size	32
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integers	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FPs	2 cycles
Branch Prediction Scheme	2-bit Counters
Memory Parameters	
Line Size	32B
I-Cache	32KB, 2-way set-associative, 4 banks
Inst.ruction Prefetch Buffer	16 entries
D-Cache	32KB, 2-way set-associative, 4 banks
Victim Buffers	8 entries each for data and inst.
Miss Handlers (MSHRS)	32 each for data and inst.
Unified S-Cache	1MB, 4-way set-associative
Primary-to-Secondary Miss Latency	12 cycles (plus any delays due to contention)
Primary-to-Memory Miss Latency	75 cycles (plus any delays due to contention)
Primary-to-Secondary Bandwidth	32 bytes/cycle
Secondary-to-Memory Bandwidth	8 bytes/cycle

instruction layouts and addresses to correspond to what they would be in the modified binary. Hence we truly emulate the physical insertion of prefetches at the expense of decreased simulation speed.

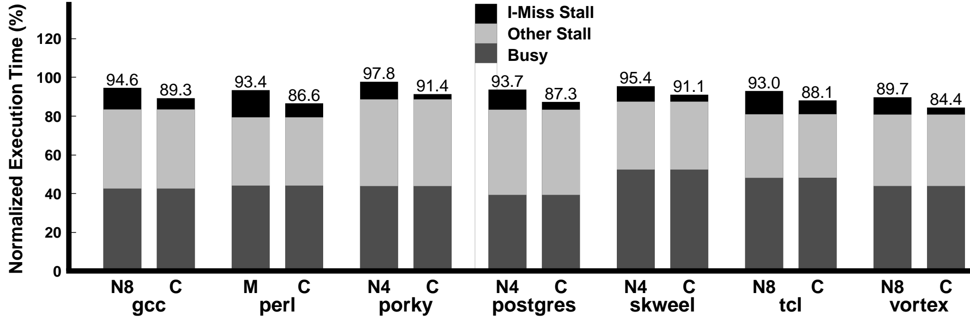


Fig. 13. Performance comparison of our basic cooperative prefetching and the best-performing existing schemes of individual applications (**N**x = next-x-line prefetching, **M** = Markov prefetching, **C** = cooperative prefetching).

6. EXPERIMENTAL RESULTS

We now present results from our simulation studies. We start by evaluating the overall performance of our basic cooperative prefetching scheme (with only direct prefetches), and then evaluate the benefit of also adding indirect prefetches (i.e., `pf_r` and `pf_i`). Next, we examine the relative importance of the two key components of our scheme: prefetch filtering and compiler-inserted prefetching. We also quantify the impact of our compiler's prefetch optimizations, and of varying the prefetching distance parameter, on the code size and performance. We then investigate if cooperative prefetching could benefit from profiling information. After that, we explore the impact of varying cache latencies and bandwidth on the performance of our scheme. Finally, we evaluate whether cooperative prefetching is cost effective.

6.1 Performance of Basic Cooperative Prefetching

Our basic cooperative prefetching scheme includes compiler-inserted `pf_d` and `pf_c` prefetches, hardware-based next-8-line prefetching, and prefetch filtering. No `pf_r` or `pf_i` prefetches (and hence the required hardware structures) are used. A prefetching distance of 20 instructions is used for all applications. (We will discuss the impact of the prefetching distance more in Section 6.5.)

Figure 13 shows the performance impact of cooperative instruction prefetching. For each application, we show two cases: the bar on the left is the best previously existing prefetching scheme (seen earlier in Figure 1), and the bar on the right is cooperative prefetching (**C**). Note that the number of instructions that actually graduate (i.e., the *busy* section) is equal in both cases because instruction prefetches are removed from the instruction stream once they are decoded (see Section 3.2). As we see in Figure 13, our cooperative prefetching scheme offers significant speedups over existing schemes (6.4% on average) by hiding a substantially larger fraction of the original instruction cache miss stall times (71% on average, as opposed to an average reduction of 36% for the best existing schemes).

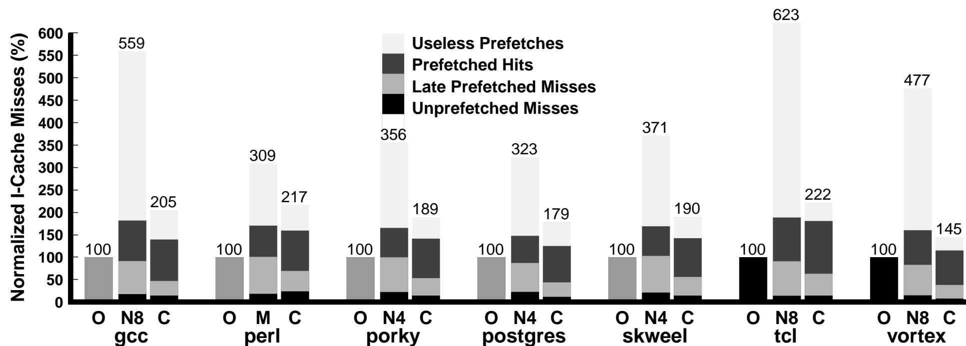


Fig. 14. Breakdown of all I-cache misses (**O** = original, **N***x* = next-*x*-line prefetching, **M** = Markov prefetching, **C** = cooperative prefetching).

To understand the performance results in greater depth, Figure 14 shows a metric which allows us to evaluate the coverage, timeliness, and usefulness of prefetches all on a single axis. This figure shows the total I-cache misses (including both fetch and prefetch misses) normalized to the original case (i.e., without prefetching) and broken down into the following four categories. The bottom section is the number of fetch misses that were not prefetched (this accounts for 100% of the misses in the original case, of course). The next section (*Late Prefetched Misses*) is where a miss has been prefetched, but the prefetched line has not returned in time to fully hide the miss (in which case the instruction fetcher stalls until the prefetched line returns, rather than generating a new miss request). The *Prefetched Hits* section is the most desirable case, where a prefetch fully hides the latency of what would normally have been a fetch miss, converting it into a hit. Finally, the top section is useless prefetches which bring lines into the cache that are not accessed before they are replaced.

Figure 14 shows that both cooperative prefetching and the best existing prefetching schemes achieve large coverage factors, as indicated by the small number of unprefetched misses. The main advantage of our scheme is that it is more effective at launching prefetches early enough. This is demonstrated in Figure 14 by the significant reduction in *late prefetched misses*, the bulk of which have been converted into *prefetched hits*. We also observe in Figure 14 that both cooperative prefetching and existing schemes experience a certain amount of *cache pollution*, since the sum of the bottom three sections of the bars adds up to over 100%. However, the *prefetch filtering* mechanism used by cooperative prefetching helps to reduce this problem, thereby resulting in a smaller total for the bottom three sections than the best existing scheme in all of our applications. In addition, Figure 14 shows another benefit of prefetch filtering: it dramatically reduces the number of useless prefetches. The reduction in total useless prefetches ranges from 2.4 in `perl` to 10.6 in `tcl`—on average, cooperative prefetching has achieved a sixfold reduction in useless prefetching.

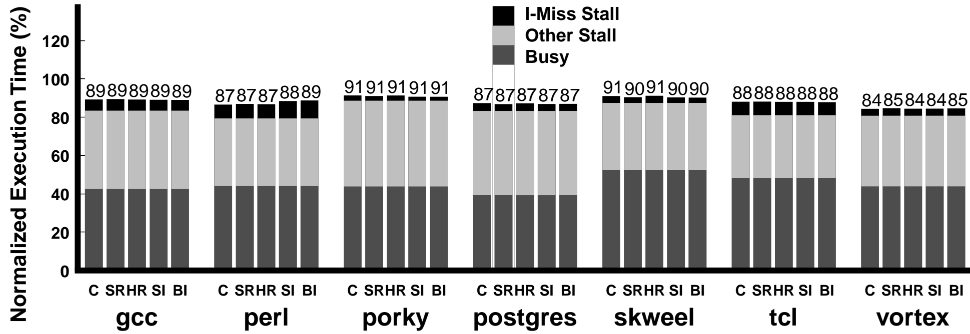


Fig. 15. Impact of adding prefetches for procedure returns and indirect jumps (**C** = basic cooperative prefetching, **SR** = basic plus `pf_r` prefetches, **HR** = basic plus using hardware to prefetch the next three return addresses at each return, **SI** = basic plus `pf_i` prefetches with a smaller indirect-target table, **BI** = basic plus `pf_i` prefetches with a bigger indirect-target table).

6.2 Adding Prefetches for Procedure Returns and Indirect Jumps

Having seen the success of our basic cooperative prefetching scheme, we now evaluate the performance benefit of extending it to include the *indirect* prefetches—i.e., `pf_r` and `pf_i` prefetches for procedure returns and indirect jumps, respectively. Figure 15 shows the performance of five variations of cooperative prefetching: the basic scheme (**C**); the basic scheme plus `pf_r` prefetches (**SR**); the basic scheme plus using hardware to prefetch the top three addresses on the stack at each procedure return (**HR**); and two cases which include the basic scheme plus `pf_i` prefetches (**SI** and **BI**). Both schemes **SR** and **HR** use a 12-entry return address stack. While scheme **HR** has no instruction overhead, scheme **SR** has a better control over the prefetching distance via compiler scheduling. Scheme **SI** uses a 1KB, 2-way set-associative indirect-target table where each entry holds up to four target addresses; scheme **BI** uses a 16KB, 4-way set-associative indirect-target table with 16 targets per entry.

As we can see in Figure 15, the marginal benefit of supporting indirect prefetches is quite small for these applications. Part of the limitation is that only a relatively small fraction (roughly 15%) of the remaining misses which are not handled by our basic scheme are due to either procedure returns or indirect jumps, and therefore the potential for improvement is small. In addition, since some indirect jumps can have a fairly large number of possible targets—e.g., more than eight, as we observe in `perl` and `gcc`—prefetching all of these targets could result in cache pollution. Prefetching indirect jump targets may become more important in applications where they occur more frequently—e.g., object-oriented programs that make heavy use of virtual functions, or applications that use shared libraries. Although two of our applications are written in C++ (`porky` and `skweel`), they rarely use virtual functions. Since our applications show little benefit from `pf_r` and `pf_i` prefetches, we do not use them in the remainder of our experiments.

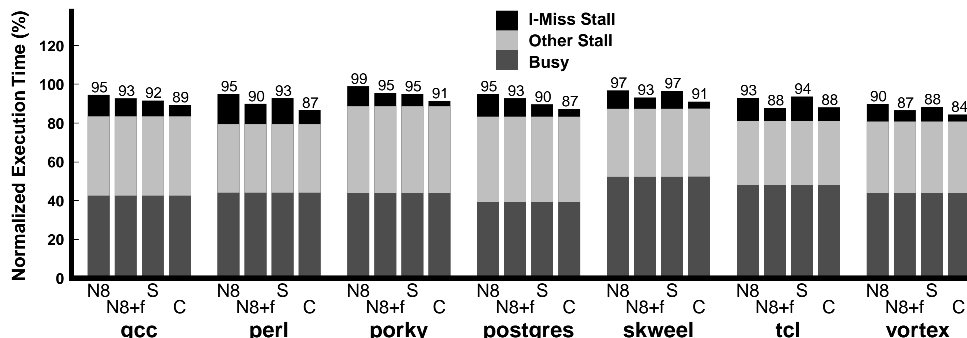


Fig. 16. Performance of four different combinations of prefetch filtering and compiler-inserted prefetching (**N8** = next-8-line prefetching alone, **N8+f** = next-8-line prefetching with prefetch filtering, **S** = compiler-inserted prefetching alone without prefetch filtering, **C** = cooperative prefetching).

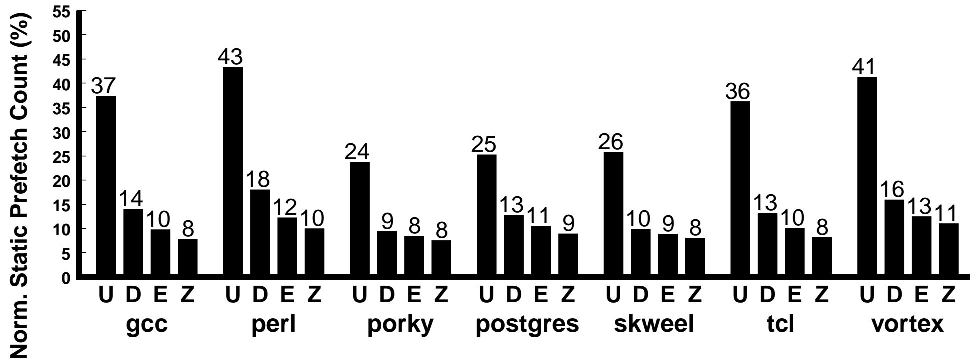
6.3 Importance of Prefetch Filtering and Software Prefetching

Two components of the cooperative prefetching design contribute to its performance advantages: prefetch filtering and compiler-inserted software prefetching. To isolate the contributions of each component, Figure 16 shows their performance individually as well as in combination. The relative importance of prefetch filtering versus compiler-inserted prefetching varies across the applications: in `tcl`, prefetch filtering is more important, and in `postgres`, compiler-inserted prefetching is more important. In all cases, the best performance is achieved when both techniques are combined, and in all but one case this results in a significant speedup over either technique alone. Intuitively, the reason for this is that the benefits of prefetch filtering (i.e., avoiding cache pollution) and software prefetching (i.e., issuing nonsequential prefetches early enough) are *orthogonal*. Hence both components of our design are clearly important for performance and are complementary in nature.

6.4 Impact of Prefetching Optimizations

To evaluate the effectiveness of the compiler optimizations discussed earlier in Section 4 in reducing the number of prefetches, we measured their impact both on code size and performance. Figure 17(a) shows the number of static prefetches remaining as each optimization pass is applied incrementally, normalized to the original code size. Without any optimization (**U**), the code size can increase by over 40%. Combining prefetches at dominators (**D**) dramatically reduces the prefetch count by more than half in all applications except `postgres`. Eliminating unnecessary prefetches and compressing prefetches further reduces the prefetch count by a moderate amount. (Prefetch hoisting has no effect on the static prefetch count, and therefore is not shown in Figure 17(a).) Altogether, the prefetch optimizations limit the prefetch count to only 9% of the original code size on average.

(a) Static prefetch count



(b) Performance

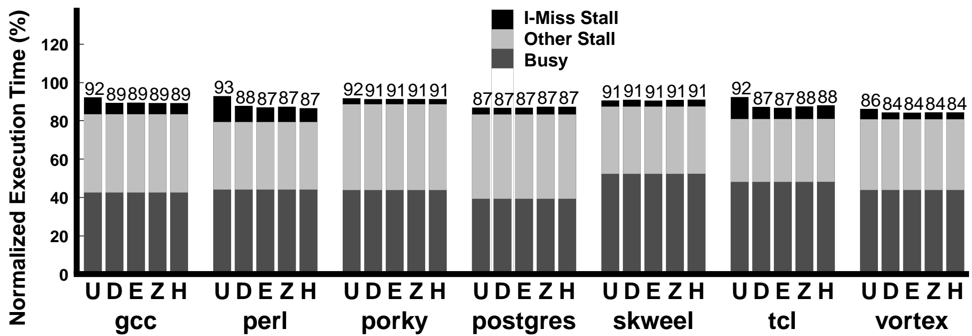


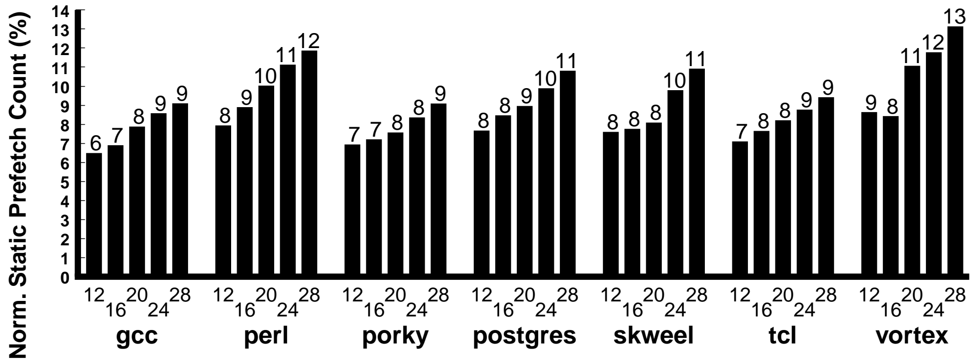
Fig. 17. Impact of prefetch optimization on (a) the static prefetch count and (b) the performance of cooperative prefetching (**U** = unoptimized, **D** = combining prefetches at dominators, **E** = case **D** plus eliminating unnecessary prefetches, **Z** = case **E** plus compressing prefetches, **H** = case **Z** plus hoisting prefetches). The y-axis of (a) is normalized to the number of instructions in the original executable.

Figure 17(b) shows the impact of these optimizations on performance. As we see in this figure, combining prefetches at dominators results in a noticeable performance improvement in several cases (e.g., `gcc`, `perl`, and `tcl`). The other optimizations have a negligible performance impact. In fact, prefetch compression and hoisting sometimes degrade performance by a very small amount by changing the order in which prefetches are launched.

6.5 Varying the Prefetching Distance

A key parameter in our prefetch scheduling compiler algorithm is the *prefetching distance* (i.e., *PF_DIST* in Figure 6). When choosing a value for this parameter, we must consider the following tradeoffs: we would like the parameter to be large enough to hide the expected miss latency, but setting the parameter too high can increase the code size (since more prefetches

(a) Static prefetch count



(b) Performance

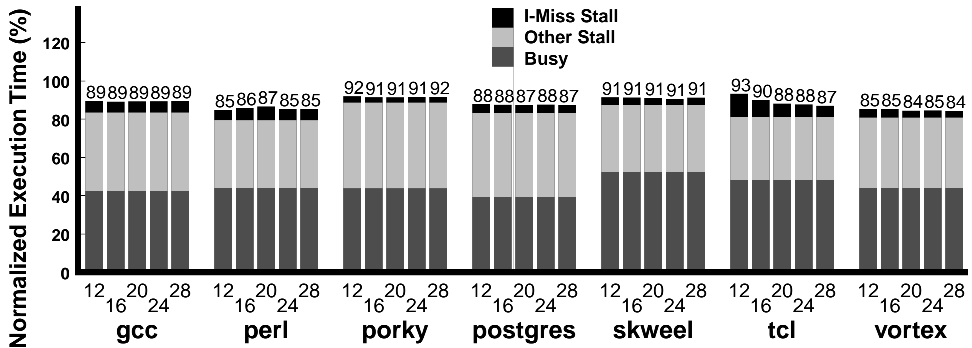


Fig. 18. Impact of the prefetching distance on (a) the static prefetch count and (b) the performance of cooperative prefetching (x = a prefetching distance of x instructions is used in the compiler scheduling; the case **20** is the default for our basic cooperative prefetching). The y-axis of (a) is normalized to the number of instructions in the original executable.

must be inserted to cover a larger number of unique incoming paths) and increase the likelihood of polluting the cache. In our experiments so far, we have used a prefetching distance of 20 instructions, which is roughly equal to the product of the expected IPC ($\sim$ 1.6) and the primary-to-secondary miss latency ($\geq$ 12 cycles). To determine the sensitivity of cooperative prefetching to this parameter, we varied the prefetching distance across a range of five values from 12 to 28 instructions, and measured the resulting impact on both code size and performance (shown in Figures 18(a) and 18(b), respectively).

As we observe in Figure 18(a), increasing the prefetching distance can result in a noticeable increase in the code size. Fortunately, even with a prefetching distance as large as 28 instructions, the compiler is still able to limit the code expansion to less that 11% on average, due to the optimizations discussed in the previous section. In contrast, the *performance* offered by cooperative prefetching is less sensitive to the prefetching distance, as we see in Figure 18(b). While `tcl` enjoys a 6% speedup as we increase this

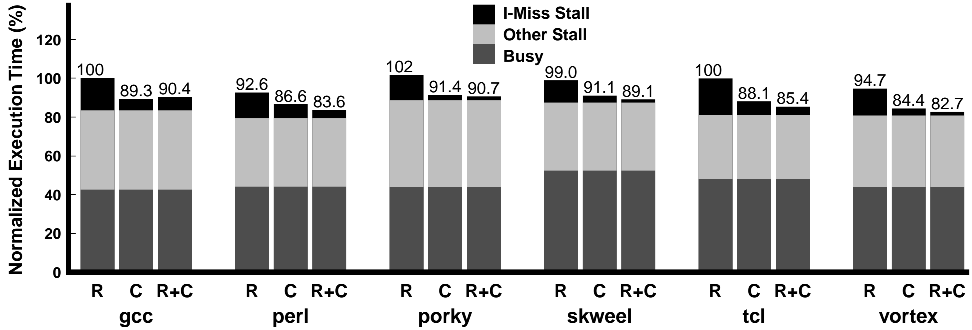


Fig. 19. Performance impact of code reordering guided by profiling information (**R** = code reordered, **C** = cooperative prefetching, **R+C** = code reordered plus cooperative prefetching).

parameter from 12 to 28 cycles, the other applications experience no more than a 2% fluctuation in performance across this range of values. Hence we observe that performance is not overly sensitive to this parameter.

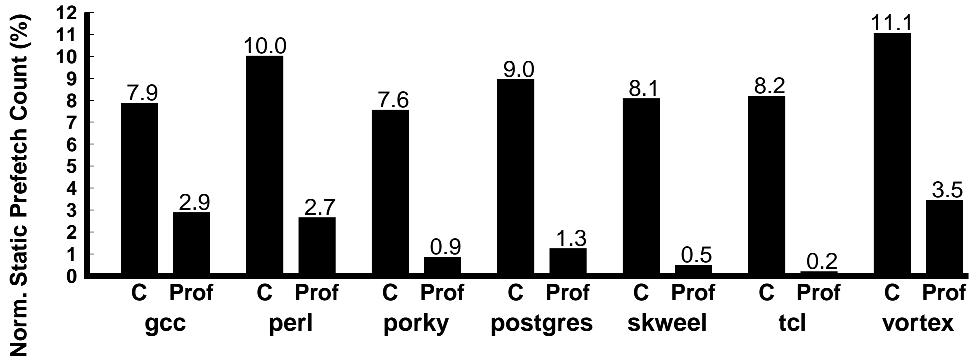
6.6 Impact of Profiling Information

One advantageous feature of cooperative prefetching is that it requires no profiling information. In this section, we study how well our technique performs relative to some profiling-driven techniques. In addition, we investigate whether cooperative prefetching could be further improved by using profiling information. We present below the results of two experiments in which profiling information is used in different ways to improve instruction cache performance.

In the first experiment, control-flow profiling information is used to guide *code reordering* for better instruction locality. Code reordering can be done at the granularity of either basic blocks [Hwu and Chang 1989; Muth et al. 2001; Pettis and Hansen 1990] or procedures [Bahar et al. 1999; Gloy et al. 1997; Hashemi et al. 1997; Hwu and Chang 1989; Pettis and Hansen 1990]. In our experiments, we used the best commercial code reordering tool that was compatible with our system: the `cord` utility in IRIX, which happens to reorder code at a procedure granularity.

Figure 19 shows the results of our experiments, where there are three cases per application: with code reordering (**R**), with cooperative prefetching (**C**), and with both code reordering and cooperative prefetching (**R+C**). For cases **R** and **R+C**, we experimented with two different code orders, one based on a profile where both the actual and training runs use the same input and the other based on a profile where the two runs use different inputs. The better-performing code order is reported in Figure 19. (We present the better-performing code order rather than the cross-training code order for sake of maximizing the benefit of code reordering; our conclusions do not change if we use the cross-training order instead.) `Postgres` is not included in Figure 19 because `prof` is unable to handle this application. We observe from Figure 19 that the performance of the code-reordered cases (**R**) is somewhat disappointing—it performs even

(a) Static prefetch count



(b) Performance

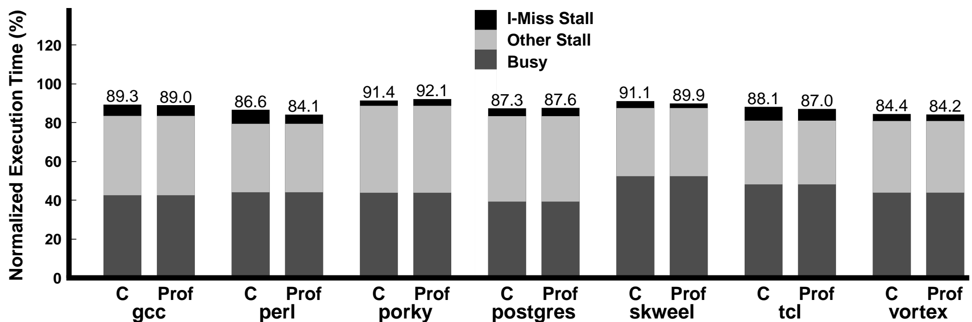
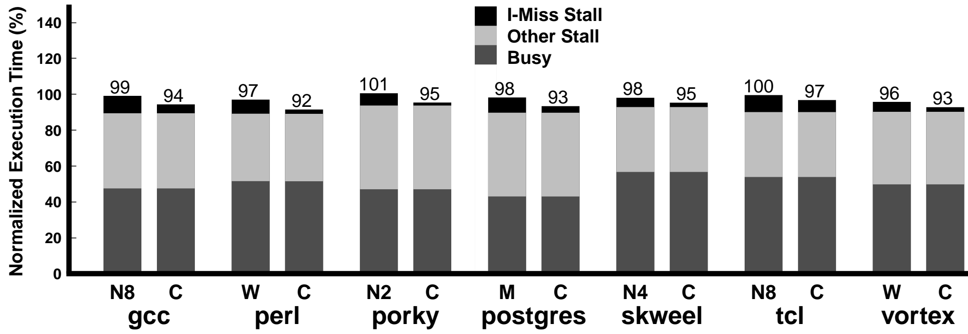


Fig. 20. Impact of profiling-guided prefetch selection on (a) the static prefetch count and (b) the performance of cooperative prefetching (**C** = original cooperative prefetching, **Prof** = cooperative prefetching with prefetches selected using profiling information). The y-axis of (a) is normalized to the number of instructions in the original executable.

worse than the original case in `porky`. The problem is, that since the code order determined by `prof` mainly reduces the cache misses caused by *conflicts* in procedure mapping, there is little improvement in *compulsory* or *capacity* misses (they may get even worse in the new code order). Fortunately, by applying cooperative prefetching to the reordered code (**R+C**), these compulsory or capacity misses are largely eliminated, while at the same time the new code order helps reduce the conflict misses that are not handled by cooperative prefetching alone. As a result, the **R+C** cases have better performance than the **C** cases in all applications except `gcc`. Although performing code reordering at a finer granularity (e.g., at a basic-block level) may help reduce the performance gap between code reordering and cooperative prefetching, we still expect the two techniques to be complementary for the same reason that they are complementary for data accesses: locality optimizations (in this case code reordering) help reduce the number of caches misses that need to be prefetched, and prefetching helps to hide the latency of the remaining misses [Mowry et al.

(a) Miss latency = 6 cycles



(b) Miss latency = 24 cycles

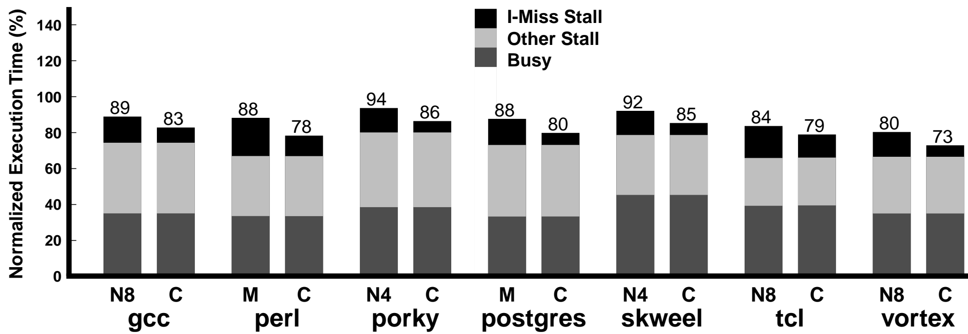


Fig. 21. Impact of varying the cache miss latency (**C** = cooperative prefetching, best-performing existing schemes: **N**x = next-x-line prefetching, **T** = target-line prefetching, **W** = wrong-path prefetching, **M** = Markov prefetching). Note that the best-performing scheme can vary for a given benchmark with different latencies.

1992]. In addition, cooperative prefetching has the advantage that it does not require profiling information.

In the second experiment, we attempted to reduce software prefetching overheads by using profiling information. We first recorded the average I-cache miss rates of individual prefetch instructions in a training run. Then we decided to insert a prefetch instruction in the actual run whenever the average miss rate of that prefetch was higher than a given threshold—that is, we wish to eliminate prefetches that are *unnecessary* most of the time. The impact of this profiling-guided prefetch selection on both the static prefetch count and the performance is shown in Figure 20. We varied the threshold miss rates from 10^{-6} to 10^{-3}, and the **Prof** case of each application in Figure 20 used the threshold that resulted in the best performance. To be optimistic, we used the same inputs for both training and actual runs. It is obvious from Figure 20(a) that this profiling information is very effective at reducing the static prefetch count even after our prefetch optimizations are applied. However, as shown in Figure 20(b), the performance impact is less clear. While the **Prof** cases achieve 1%–3%

speedups over the **C** cases in `perl`, `skweel`, and `tcl`, they also perform a little worse than the **C** cases in two applications because some discarded static prefetches turn out to be useful occasionally.

Overall, while the two kinds of profiling information studied in this section could further improve the performance of cooperative prefetching, they do not appear to be indispensable, since cooperative prefetching alone already offers most of the same performance advantages.

6.7 Impact of Latency and Bandwidth Variations

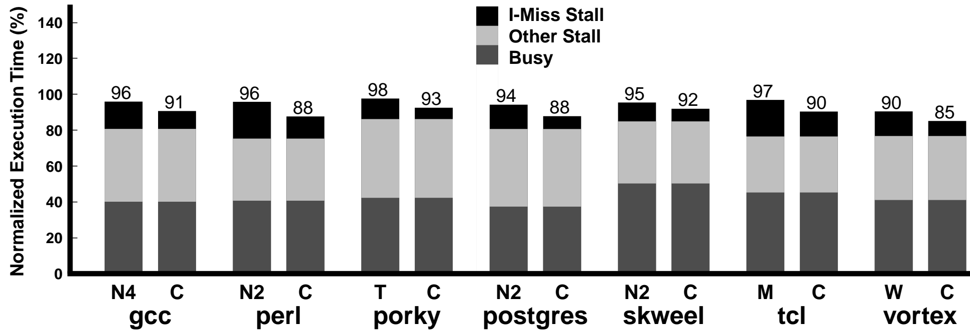
We now consider the impact of varying miss latencies and available bandwidth between the primary and secondary caches on the performance of cooperative prefetching. Recall, in our experiments so far, that the primary-to-secondary miss latency has been 12 cycles (plus any delays due to contention). Figure 21 shows the performance of the best-performing existing schemes and cooperative prefetching when the primary-to-secondary latency is decreased to 6 cycles and increased to 24 cycles. (Note that the compiler's prefetching distance was set to 12 and 28 instructions, respectively, for the 6-cycle and 24-cycle cases.) As we see in Figure 21, cooperative prefetching still performs well under both latencies, and results in even larger improvements as the latency grows. In the 24-cycle case, cooperative prefetching results in an average speedup of 24.4%, which is significantly larger than the 14.2% speedup offered by the best existing scheme.

Turning our attention to bandwidth, recall that our experiments so far have assumed a bandwidth of 32 bytes/cycle between the primary instruction cache and the secondary cache. Figure 22 shows the impact of decreasing this bandwidth to 8 bytes/cycle, and of increasing it to unlimited bandwidth. There are two things to note from these results. First, we see in Figure 22(a) that while reducing the bandwidth does degrade the performance of cooperative prefetching somewhat—from an average speedup of 13.3% to 11.9%—the overall performance gain still remains high. Hence cooperative prefetching can achieve good performance with the range of bandwidth that is common for recent processors. (Note that this bandwidth includes servicing data cache misses as well.) Second, we observe in Figure 22(b) that *increasing* the bandwidth beyond 32 bytes/cycle does not significantly improve the performance of cooperative prefetching (the average speedup only increases from 13.3% to 13.7%). Therefore cooperative prefetching is not bandwidth-limited, and it is more likely that it is limited by other factors (e.g., cache pollution, achieving a sufficient prefetching distance, etc.).

6.8 Cost Effectiveness

Having demonstrated the performance advantages of cooperative prefetching, we now focus on whether the additional hardware support is cost effective. One alternative to cooperative prefetching would be to simply increase the cache sizes by a comparable amount. (Note that this is overly

(a) Bandwidth = 8 bytes/cycle



(b) Bandwidth = infinite

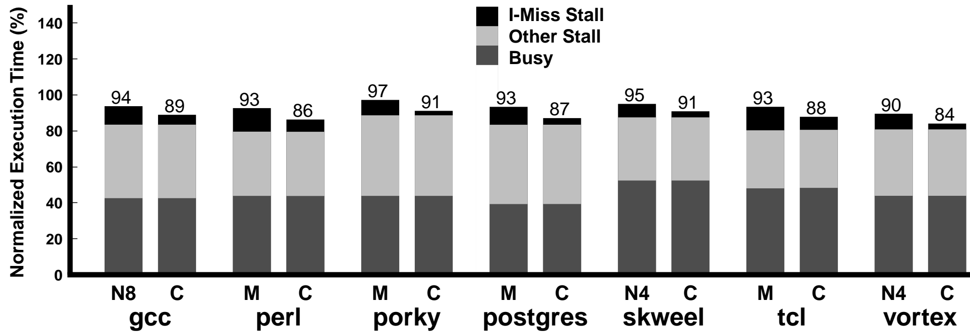


Fig. 22. Impact of varying the bandwidth between the I-cache and L2 cache (**C** = cooperative prefetching, best performing existing schemes: **N**x = next-x-line prefetching, **T** = target-line prefetching, **W** = wrong-path prefetching, **M** = Markov prefetching). Note that the best-performing scheme can vary for a given benchmark with different bandwidths.

simplistic, since the primary cache sizes are often limited more by access time than the amount of silicon area available.) For our baseline architecture, the additional storage necessary to support basic cooperative prefetching is 640 bytes at the level of the primary I-cache (128 bytes for the prefetch bits used by prefetch filtering, and 512 bytes for the prefetch buffer), and 8KB for the 2-bit saturating counters added to the L2 cache.

Figure 23 compares the performance of a 32KB I-cache with cooperative prefetching with that of three larger I-caches, ranging from 64KB to 256KB, without prefetching. It is encouraging that the average speedup achieved by cooperative prefetching (13.3%) is greater than that obtained by doubling the cache size from 32KB to 64KB (10.8%) despite the substantially higher hardware cost of the larger cache. In addition, cooperative prefetching outperforms the 128KB I-cache in three of the seven applications, and is within 2% of the performance with a 256KB I-cache in five cases. Overall, cooperative prefetching appears to be a more cost-effective method of improving performance than simply increasing the I-cache size.

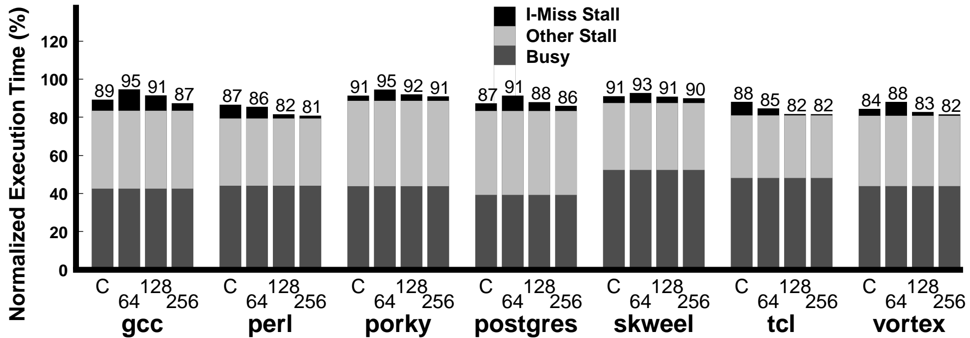


Fig. 23. Performance comparison of cooperative prefetching and larger I-caches (**C** = a 32KB I-cache with cooperative prefetching, x = an xKB I-cache without prefetching). The y-axis is normalized to the execution time of a 32KB I-cache without prefetching.

7. CONCLUSIONS

To overcome the disappointing performance of existing instruction prefetching schemes on modern microprocessors, we have proposed and evaluated a new prefetching scheme whereby the hardware and software cooperate as follows: the hardware performs aggressive next-N-line prefetching combined with a novel *prefetch filtering* mechanism to get far ahead on sequential accesses without polluting the cache, and the compiler uses a novel algorithm to insert explicit *instruction-prefetch instructions* into the executable to prefetch nonsequential accesses. Our experimental results demonstrate that our scheme significantly outperforms existing schemes, eliminating 50% or more of the latency that had remained with the best existing scheme. This reduction in latency translates into a 13.3% average speedup over the original execution time on a state-of-the-art superscalar processor, which is more than double the 6.5% speedup achieved by the best existing scheme, and much closer to the maximum 20% speedup (for these applications and this architecture) in the ideal instruction prefetching case. These improvements are the result of launching prefetches earlier (thereby hiding more latency), while at the same time reducing the cache-polluting effects of useless prefetches dramatically. Given these encouraging results, we advocate that future microprocessors provide instruction-prefetch instructions along with the prefetch filtering mechanism.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

BAHAR, I., CALDER, B., AND GRUNWALD, D. 1999. A comparison of software code reordering and victim buffers. *SIGARCH Comput. Arch. News*.

BERNSTEIN, D., COHEN, D., AND FREUND, A. 1995. Compiler techniques for data prefetching on the PowerPC. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques* (PACT '95, Limassol, Cyprus, June 27–29), L. Bic, P. Evripidou, W. Böhm, and J.-L. Gaudiot, Chairs. IFIP Working Group on Algol, Manchester, UK, 19–26.

CHANG, P.-Y., HAO, E., AND PATT, Y. N. 1997. Target prediction for indirect jumps. In *Proceedings of the 24th International Symposium on Computer Architecture* (ISCA '97, Denver, CO, June 2–4), A. R. Pleszkun and T. Mudge, Chairs. ACM Press, New York, NY, 274–283.

DRIESEN, K. AND HOLZLE, U. 1998a. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (ISCA '98, Barcelona, Spain, June 27–July 1), D. DeGroot, Ed. IEEE Press, Piscataway, NJ, 167–178.

DRIESEN, K. AND HÖLZLE, U. 1998b. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (MICRO-31, Dallas, TX, Nov. 30–Dec. 2), J. Bondi and J. Smith, Chairs. IEEE Computer Society Press, Los Alamitos, CA, 249–258.

GLOY, N., BLACKWELL, T., SMITH, M. D., AND CALDER, B. 1997. Procedure placement using temporal ordering information. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 30, Research Triangle Park, NC, Dec. 1–3), M. Smotherman and T. Conte, Chairs. IEEE Computer Society Press, Los Alamitos, CA, 303–313.

GRUNWALD, D., KLAUSER, A., MANNE, S., AND PLESZKUN, A. 1998. Confidence estimation for speculation control. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (ISCA '98, Barcelona, Spain, June 27–July 1), D. DeGroot, Ed. IEEE Press, Piscataway, NJ, 122–131.

HASHEMI, A. H., KAELI, D. R., AND CALDER, B. 1997. Efficient procedure mapping using cache line coloring. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '97, Las Vegas, NV, June 15–18), A. M. Berman, Ed. ACM Press, New York, NY, 171–182.

HWU, W. AND CHANG, P. 1989. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture* (ISCA '89, Jerusalem, Israel, May 28–June 1), J.-C. Syre, Chair. ACM Press, New York, NY, 242–251.

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching using Markov predictors. In *Proceedings of the 24th International Symposium on Computer Architecture* (ISCA '97, Denver, CO, June 2–4), A. R. Pleszkun and T. Mudge, Chairs. ACM Press, New York, NY, 252–263.

JOUPPI, N. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture* (ISCA '90, Seattle, WA, May). IEEE Press, Piscataway, NJ, 364–373.

KANE, G. AND HEINRICH, J. 1992. *MIPS RISC Architectures*. Prentice-Hall, Inc., Upper Saddle River, NJ.

MAYNARD, A., DONNELLY, C., AND OLSZEWSKI, B. 1994. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS VI, San Jose, CA, Oct. 4–7), F. Baskett and D. Clark, Chairs. ACM Press, New York, NY, 145–156.

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-V, Boston, MA, Oct. 12–15), S. Eggers, Chair. ACM Press, New York, NY, 62–73.

MUTH, R., DEBRAY, S., WATTERSON, S., AND BOSSCHERE, K. D. 2001. Alto: A link-time optimizer for the Compaq Alpha. *Softw. Pract. Exper.*.

PETTIS, K. AND HANSEN, R. 1990. Profile guided code positioning. In *Proceedings of the Conference on Programming Language Design and Implementation* (SIGPLAN '90, White Plains, NY, June 20–22), B. N. Fischer, Chair. ACM Press, New York, NY, 16–27.

PIERCE, J. AND MUDGE, T. 1996. Wrong-path instruction prefetching. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture* (MICRO 29, Paris, France, Dec. 2–4), S. Melvin and S. Beaty, Chairs. IEEE Computer Society Press, Los Alamitos, CA, 165–175.

REINMAN, G., CALDER, B., AND AUSTIN, T. 1999a. Fetch directed instruction prefetching. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture* (Nov.). 16–27.

REINMAN, G., CALDER, B., AND AUSTIN, T. 1999b. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th Annual International Symposium on Computer Architecture* (June). 234–245.

SANTHANAM, V., GORNISH, E. H., AND HSU, W.-C. 1997. Data prefetching on the HP PA-8000. In *Proceedings of the 24th International Symposium on Computer Architecture* (ISCA '97, Denver, CO, June 2–4), A. R. Pleszkun and T. Mudge, Chairs. ACM Press, New York, NY, 264–273.

SMITH, A. 1978. Sequential program prefetching in memory hierarchies. *IEEE Computer 11*, 2, 7–21.

SMITH, A. J. 1982. Cache memories. *ACM Comput. Surv. 14*, 3 (Sept.), 473–530.

SMITH, J. E. AND HSU, W.-C. 1992. Prefetching in supercomputer instruction caches. In *Proceedings of the 1992 Conference on Supercomputing* (Supercomputing '92, Minneapolis, MN, Nov. 16–20), R. Werner, Ed. IEEE Computer Society Press, Los Alamitos, CA, 588–597.

STARK, J., RACUNAS, P., AND PATT, Y. N. 1997. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 30, Research Triangle Park, NC, Dec. 1–3), M. Smotherman and T. Conte, Chairs. IEEE Computer Society Press, Los Alamitos, CA, 34–43.

WEBB, C. F. 1988. Subroutine call/Return stack. *IBM Tech. Discl. Bull. 30*, 11 (Apr.).

XIA, C. AND TORRELLAS, J. 1996. Instruction prefetching of system codes with layout optimized for reduced cache misses. In *Proceedings of the 23rd International Symposium on Computer Architecture* (Philadelphia, PA, May). ACM Press, New York, NY, 271–282.

YEAGER, K. C. 1996. The MIPS R10000 superscalar microprocessor. *IEEE Micro 16*, 2 (Apr.), 28–40.

YU, A. AND CHEN, J. 1996. *The Postgres95 User Manual v1.0*. University of California at Berkeley, Berkeley, CA.

Received: January 2000; revised: October 2000; accepted: October 2000