Compiler and Hardware Support for Automatic Instruction Prefetching: A Cooperative Approach

Todd C. Mowry   Chi-Keung Luk[†]

June 1998

CMU-CS-98-140

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, M5S 3G4.

Abstract

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 non-sequential 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 non-sequential 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 results in speedups ranging from 9.4% to 18.5% (13.3% on average) over the original execution time on an out-of-order superscalar processor, which is more than double the average speedup of the best existing schemes (6.5%). This is accomplished by hiding an average of 71% of the original instruction stall time, compared with only 36% for the best existing schemes. We find that both the *prefetch filtering* and *compiler-inserted prefetching* components of our design are essential and complementary, that the compiler can limit the code expansion to less than 10% on average, and that our scheme is robust with respect to variations in miss latency and bandwidth.

Todd C. Mowry is partially supported by a Faculty Development Award from IBM. Chi-Keung Luk is partially supported by a Canadian Commonwealth Fellowship.

Keywords: B.3.2 Cache Memories, C.4 Performance of Systems (Measurement Techniques, Performance Attributes), D.3.4 Compilers

1 Introduction

Memory latency is a key performance bottleneck in modern microprocessor-based systems. The relative importance of memory latency is expected to increase as the gap between processor and memory speeds continues to grow, and as wider-issue processors increase the effective performance penalty of each cycle of latency. While techniques for coping with data access latency have received considerable attention, it is also important to address the latency of fetching *instructions*. Although instruction cache hierarchies are an essential first step toward coping with this problem, they are not a complete solution. For example, a study conducted by Maynard *et al.* [7] 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

There has been a long history of research on instruction prefetching. 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 [10, 11], *target-line* prefetching [12], *wrong-path* prefetching [8], and *Markov* prefetching [3].

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 when prefetches are launched is also crucial. The *prefetching distance* is the elapsed time between when the prefetch is initiated and when the prefetched instruction is used. The prefetching distance should be large enough to fully hide the cache 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).

As its name implies, the idea behind *next-N-line prefetching* [10, 11] 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. To increase the likelihood that these prefetched sequential lines will be used, the hardware can postpone launching a prefetch until the current instruction falls within a specified distance (called the *fetch-ahead distance*) of the end of its line [12]. 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 schemes.

One limitation of next-N-line prefetching is that it does not prefetch control transfer targets which do not fall within the N fall-through lines. To address this limitation, Smith and Hsu [12] 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 a history table to predict likely target addresses, Pierce and Mudge [8] proposed a scheme called *wrong-path prefetching* which combines next-N-line prefetching with always prefetching the target of control transfers with static target addresses (including procedure calls, conditional and unconditional branches). Hence for conditional branches, both the target and fall-through lines will always be prefetched. However, since target addresses cannot be determine 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 in between to hide the latency of fetching the target line, but not so much time that the line has been displaced). Their results indicated that wrong-path prefetching performed slightly better than next-1-line prefetching on average.

Table 1: Parameters used in the evaluation of existing instruction prefetching techniques.

Technique	# of Sequential Lines Prefetched	Target Prefetching Parameters		
		# 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	512 KB	direct-mapped with tags

Joseph and Grunwald [3] proposed *Markov prefetching* which is applicable to both instruction and data cache misses. This mechanism correlates the current cache miss address with the next miss address and stores this information in a *miss-address prediction table* using the current miss address as the index. Multiple predicted addresses can be associated with a given miss address. Upon a cache miss, prefetches are issued for these 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.

Finally, we note that while a previous study by Xia and Torrellas [13] 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 [14]. We model a two-level cache hierarchy with split 32 KB, two-way set-associative primary instruction and data caches and a unified 1 MB, 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 [4] and a 16-entry prefetch buffer [3]. Further details on our experimental framework will be presented later in Section 5.

Table 1 summarizes the parameters used throughout our experiments for each of the prefetching schemes. These parameters were chosen through experimentation in an effort to maximize the performance of each scheme. All schemes effectively include next-2-line prefetching.[1] We do not use the fetch-ahead distance mechanism [12] to throttle back prefetching. 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 non-numeric applications (which are described in more detail later 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 explaining what happened during all potential graduation slots.[2] The bottom section (*Busy*) is the number of slots when instructions actually graduate, the top section (*I-Miss Stall*) is any non-graduating 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

[1] We added next-2-line prefetching to Markov prefetching (despite the fact that this was not in the original design [3]) because this provides better performance than Markov prefetching alone.

[2] 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.

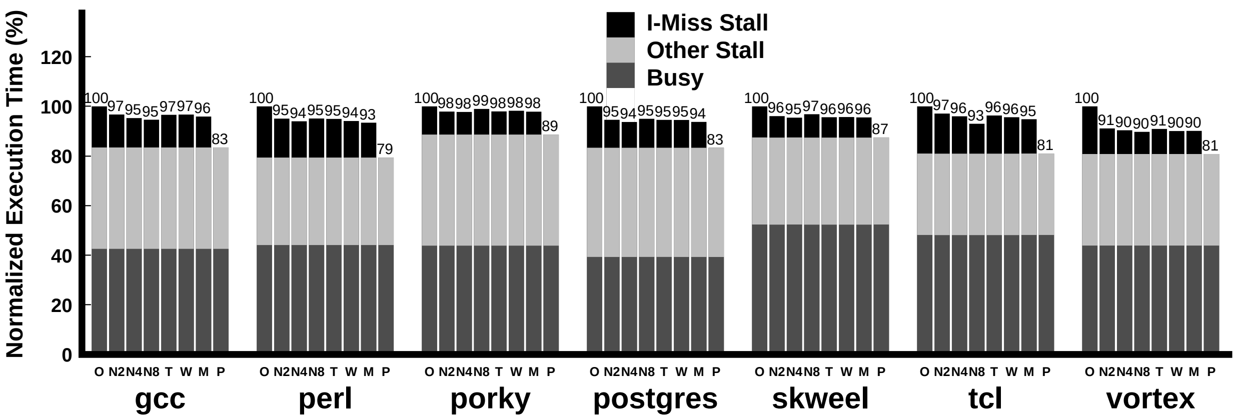

Figure 1: Performance of existing instruction prefetching techniques (**O** = original, **N**x = next-x-line prefetching, **T** = target-line prefetching, **W** = wrong-path prefetching, **M** = Markov prefetching, **P** = perfect instruction cache).

dominating. Perhaps 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 non-sequential 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 how our prefetching scheme works. To make our approach concrete, we also present some examples illustrating how prefetches are inserted.

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 non-sequential 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 once the end of the sequence is reached. Xia and Torrellas [13] proposed a mechanism for doing this which involves having software 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 have a chance to 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 *non-sequential* as well as sequential prefetches.

2.1.2 Prefetching Non-Sequential Accesses

In contrast with sequential access patterns, purely hardware-based prefetching schemes are far less successful at prefetching *non-sequential* 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.[3] 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 non-sequential 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 *prefetch-scheduling 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. 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.

[3] Note that although our prefetch filtering mechanism can also potentially suffer from the limitations of learning within a finite table, we find that it is far more important to prefetch target addresses early enough rather than filtering out all useless prefetches.

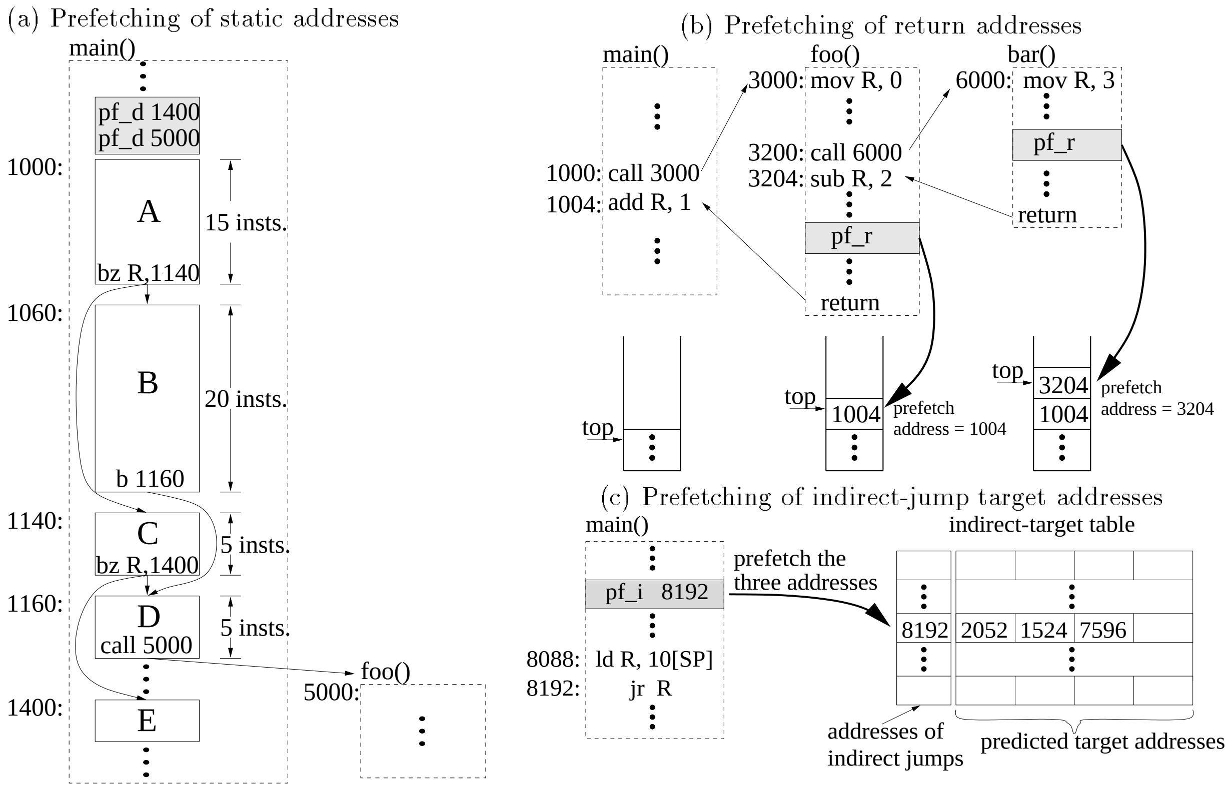

Figure 2: Examples of prefetch insertion for different types of target addresses. (**pf_d** = prefetch a direct address, **pf_r** = prefetch a return address, **pf_i** = prefetch an indirect-jump target address.)

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 in order for software to generate the target addresses early. 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* [5], we simply use a special prefetch instruction which implicitly uses the top of the return address stack as its argument.[4] 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.)

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.

[4] 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 which have register arguments.

2.2 Examples 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 prefetch-scheduling 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*[5] 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). 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 support from the architecture. In this section, we describe how we extend the instruction set architecture, the impact that these new instructions have on the pipeline, and the new hardware that we add to the memory system (including the prefetch filter).

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 [6]. 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.[6] 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

[5] 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 [1].

[6] Since most machines have at least 16 byte instruction lines, this is not a limitation.

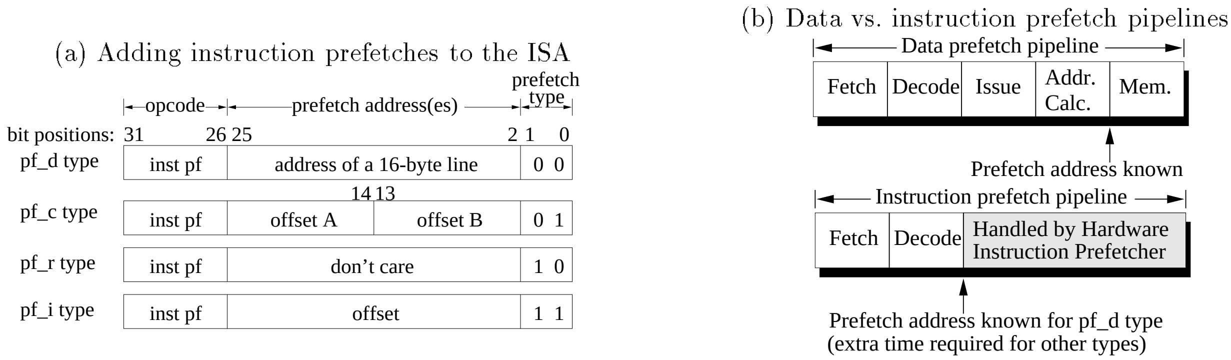

Figure 3: Possible extensions to the ISA and the CPU pipeline for instruction prefetches

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 [2, 9, 14]. 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 [14] 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 (the mostly used type) is known immediately after the *Decode* stage (the other types of instruction prefetches 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 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

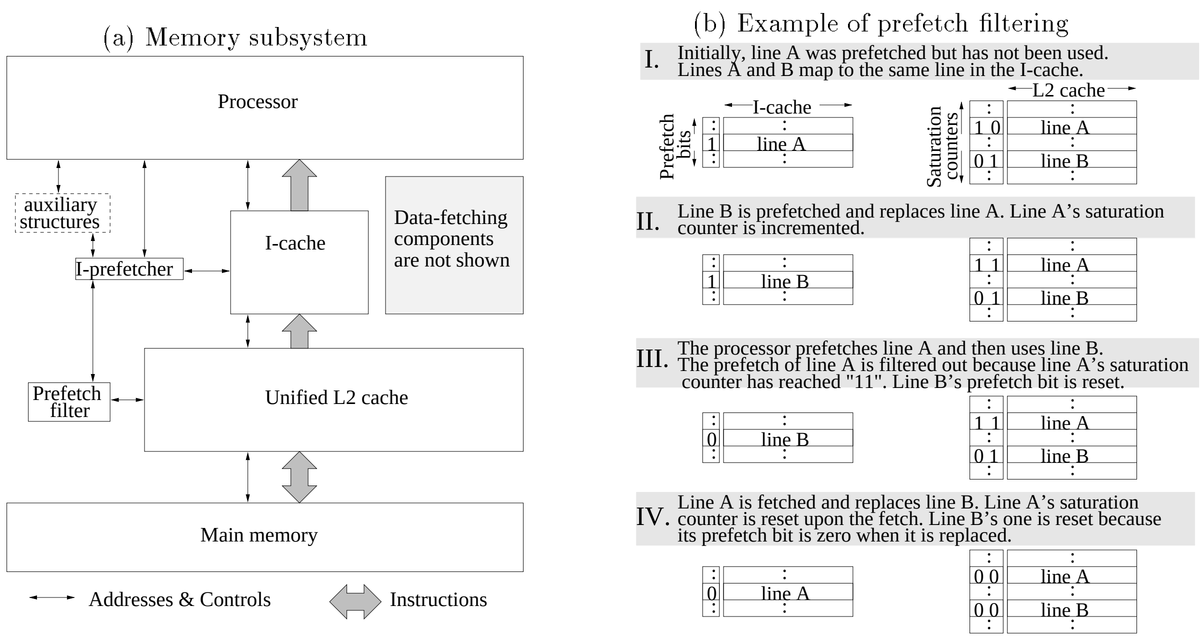

Figure 4: The memory subsystem and an example of the prefetch filtering mechanism.

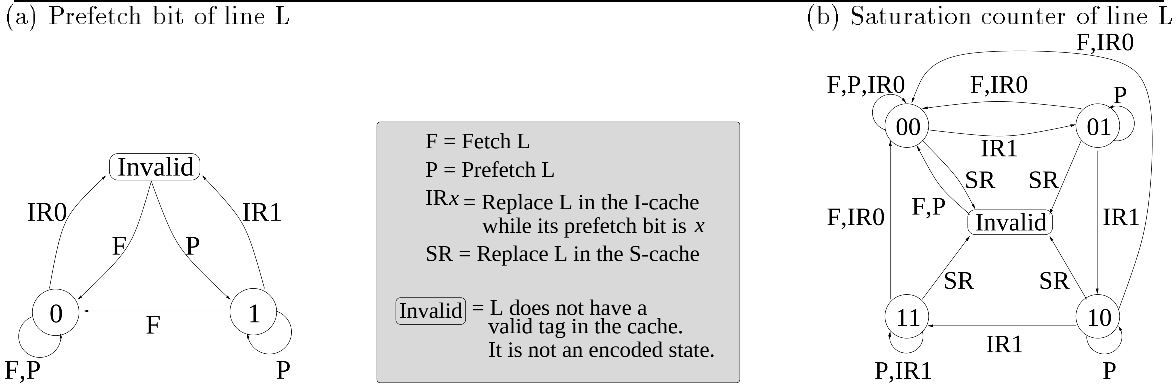

Figure 5: The states and transitions of (a) prefetch bits and (b) saturation counters under prefetch filtering.

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. 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.

```
void schedule_prefetches(E) {
    foreach basic block B in the executable E do
        schedule(B, B, 0, {});
}

// Consider attaching a prefetch for T to B where:
//   B = current basic block, T = prefetch-target basic block,
//   D = the prefetching distance between B and T
//   S = set of basic blocks scheduled so far
//   SCHED_DIST = prefetch-scheduling distance
//   N = the N used in hardware next-N-line prefetching
void schedule(B, T, D, S) {
   if (B ∉ S) { // continue only if B hasn't been scheduled
        S = S ∪ {B};
        // Attach a prefetch if it is sufficiently early and
        // if it is necessary.
        if ((D ≥ SCHED_DIST)
            and not locality_likely(B, T)
            and not nextNline_prefetchable(B, T, N)
            and not prefetch_already_exists(B, T)) {
                attach_prefetch(B, T);
        }
        foreach basic block P which can reach B
            in a single direct control transfer do {
            // update prefetching distance conservatively
            D' = D + min_length(P);
            schedule(P, T, D', S);
        }
   }
}

boolean locality_likely(B, T) {
    // If B and T are in the same loop or recursive procedure
    // chain that accesses a very small volume of instructions
    // relative to the I-cache size, it is likely that T is already
    // in the I-cache when we are executing B. In this case,
    // we return TRUE; otherwise return FALSE.
}

boolean nextNline_prefetchable(B, T, N) {
    // Determine whether T is within N cache lines of B.
}

boolean prefetch_already_exists(B, T) {
    // Check whether a prefetch for T is already attached to B.
}

void attach_prefetch(B, T) {
    // Insert a prefetch of T before the first instruction in B.
}

int min_length(B) {
    // Return the number of instructions executed in basic block
    // B. If B doesn't end with a procedure call, this is simply the
    // number of instructions in B; otherwise, this is the number
    // of instructions in B plus the length of the shortest path
    // from the beginning to the end of the procedure called by B.
}
```

Figure 6: Pseudo-code representation of our prefetch scheduling compiler algorithm.

4 Compiler Support

The compiler is responsible for automatically inserting prefetch instructions into the executable. Note that 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 in our case, we implemented it as a binary rewrite tool. The goal of the compiler is to schedule prefetches to achieve high coverage and satisfactory prefetching distances while at the same time minimizing the static and dynamic instruction overhead. Hence our compiler algorithm has two major phases: *prefetch scheduling* and *prefetch optimization*. Figure 6 shows a pseudo-code representation of our prefetch scheduling algorithm. After generating an initial prefetch schedule, the compiler then performs the four optimization passes described below, using the running example in Figure 7. A complete implementation of this algorithm was used throughout our experiments.

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 harmful (it is harmful when the boosted prefetches will displace other useful instructions from the cache before b is referenced). After this boosting process, the compiler could combine some prefetches at dominators. For example, Figure 7(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`.

Pass 2: *Eliminating Unnecessary Prefetches.* A prefetch instruction targeting a line l is *unnecessary* if l resides in the I-cache on *all* possible paths reaching the prefetch instruction. To eliminate unnecessary prefetch instructions, the compiler estimates which lines reside in the I-cache at each prefetch instruction using an algorithm similar to the one for computing *available expressions* in classical code optimization [1]. In our case, the *gen* set of a basic block b is the set of lines fetched or prefetched by b while the *kill* set is the set of lines displaced by b. In our example, since line `z` will definitely be in the I-cache when we enter basic block `C` regardless of whether we came from `A` or `B`, the prefetch of line `z` in `C` is unnecessary and therefore is eliminated, as shown in Figure 7(c).

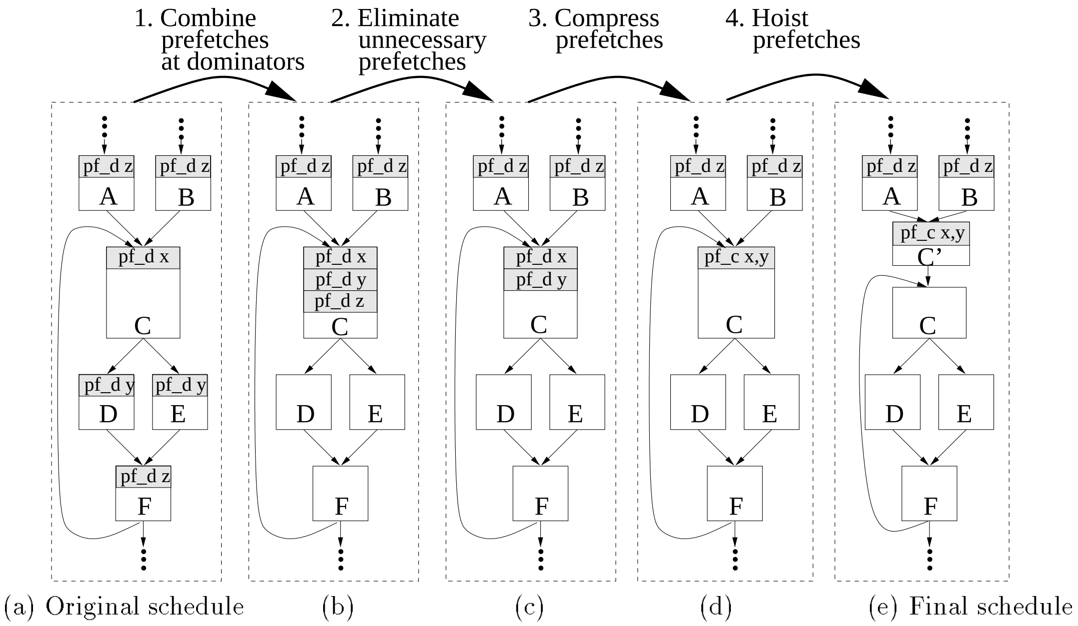

Figure 7: Example of prefetch optimization. (`A` to `F` are basic blocks; `x`, `y` and `z` are cache line addresses. `C` is a dominator of `D`, `E`, `F`, and `C` itself.)

Pass 3: *Compressing Prefetches.* The compiler checks whether multiple **pf_d** prefetches in the same basic block can be compressed into a 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 7(d).

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 re-executed at every iteration (which may not be the case if each iteration can access a large volume of instructions). In some cases, a *pre-header* block will be created for the loop to hold the hoisted prefetches. For example, in Figure 7(e), a pre-header `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.

5 Experimental Framework

We performed our experiments on seven non-numeric applications which were chosen because their relatively large instruction footprints result in poor instruction cache performance. These applications are described Table 2, and all of them were run to completion.

We performed detailed cycle-by-cycle simulations of our applications on a dynamically-scheduled, superscalar processor similar to the MIPS R10000 [14]. Our simulator models the rich details of the processor including the pipeline, register renaming, the reorder buffer, branch prediction, branching penalties, speculative instruction fetching (including incorrect execution paths), the memory hierarchy (including tag, bank, and bus contention), etc. Table 3 shows the parameters used in our model for the bulk of our experiments (we vary the latency and bandwidth later in Section 6.6). As shown in Table 3, 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 [4]

Table 2: Benchmark 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.)

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 called 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 [15]	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 (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 drawn from SPEC95	A reduced SPEC95 input set	193.0M	6.48%	0.08%

Table 3: Simulation parameters for the baseline architecture.

Pipeline Parameters	
Fetch & Decode Width	8 sequential instructions. on the same cache line
Issue & 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 Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction Scheme	2-bit Counters

Memory Parameters	
Line Size	32B
I-Cache	32KB, 2-way set-associative, 4 banks
Inst. 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

and a 16-entry prefetch buffer [3]. Our prefetching buffer is similar to the one used in the Markov prefetching study [3], with the only difference being that when an entry is forced out of this buffer, we place it in the instruction cache rather than dropping it. Hence anything that enters the prefetch buffer eventually enters the instruction cache in our model—its primary purpose is to *delay* filling the instruction cache to help avoid cache conflicts.

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 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.

6 Experimental Results

We now present results from our simulation studies. We start by evaluating the 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

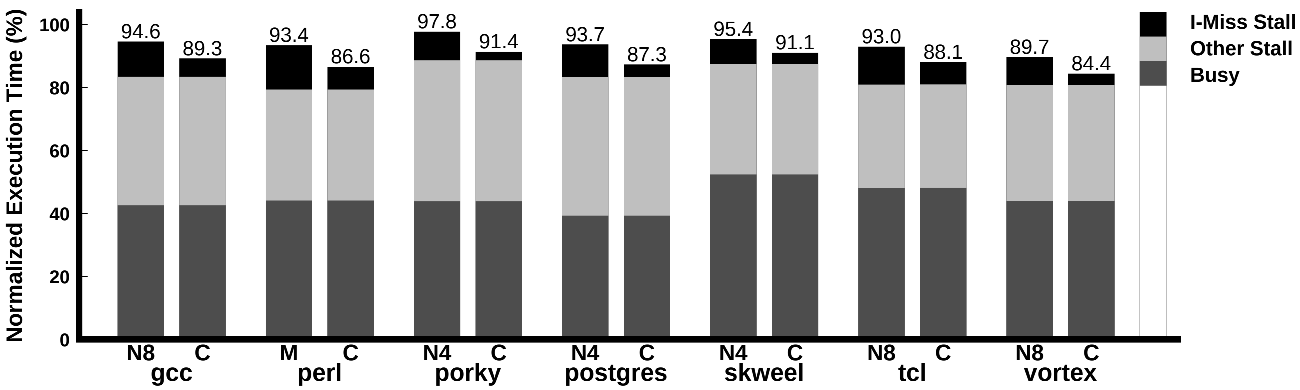

Figure 8: Performance comparison of our basic cooperative prefetching scheme and the best performing existing schemes of individual applications (**N**x = next-x-line prefetching, **M** = Markov prefetching, **C** = cooperative prefetching)

two key components of our scheme: prefetch filtering and software-initiated prefetching. We then measure the impact of varying the prefetch-scheduling distance used by the compiler, and of our compiler's prefetch optimizations, on the code size and performance. We also quantify the impact of varying cache latencies and bandwidths on the performance of our scheme. Finally, we justify the hardware cost of cooperative prefetching.

6.1 Performance of the Basic Cooperative Prefetching Scheme

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 prefetch-scheduling distance of 20 instructions is used for all applications.

Figure 8 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**). As we see in Figure 8, 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).

To understand the performance results in greater depth, Figure 9 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 9 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 9 by the significant reduction in *late prefetched misses*, the bulk of which have been converted into *prefetched hits*. We also observe in Figure 9 that both cooperative prefetching

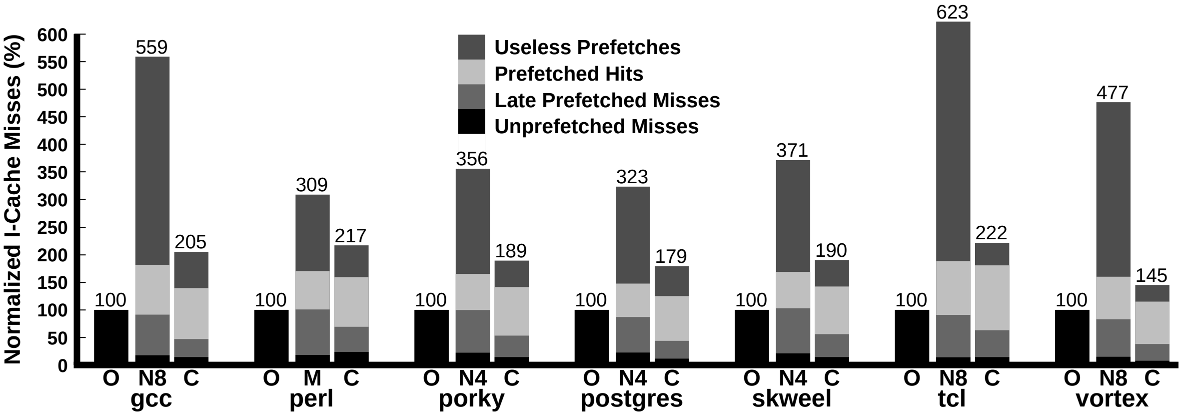

Figure 9: Breakdown of all I-cache misses, normalized to the original case. (**O** = original, **N**x = next-x-line prefetching, **M** = Markov prefetching, **C** = cooperative prefetching).

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 9 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.

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 10 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 1 KB, 2-way set-associative indirect-target table where entry holds up to four target address; scheme **BI** uses a 16 KB, 4-way set-associative indirect-target table with 16 targets per entry.

As we can see in Figure 10, 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.

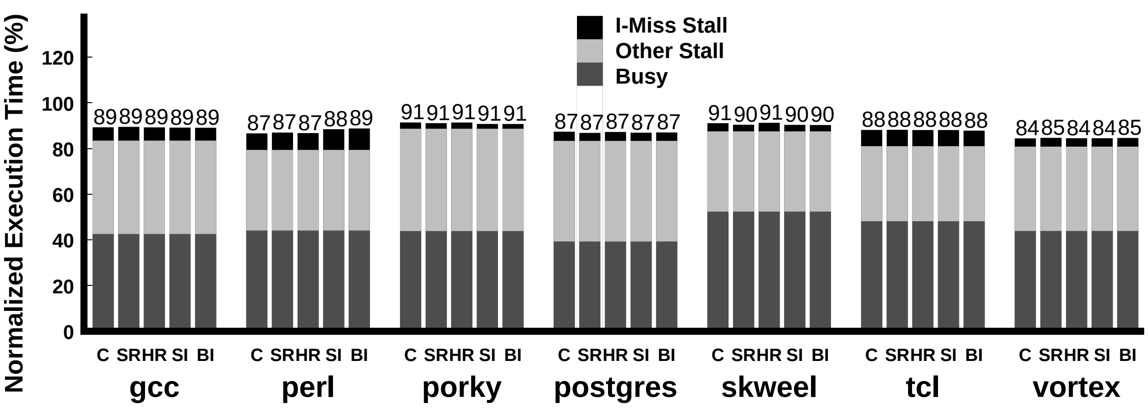

Figure 10: Impact of adding prefetches for procedure returns and indirect jumps (**C** = basic cooperative prefetching, **SR** = basic plus `pf_r` prefetches, **HR** = basic plus using hardware to prefetch the next three return addresses at each return, **SI** = basic plus `pf_i` prefetches with a smaller indirect-target table, **BI** = basic plus `pf_i` prefetches with a bigger indirect-target table.)

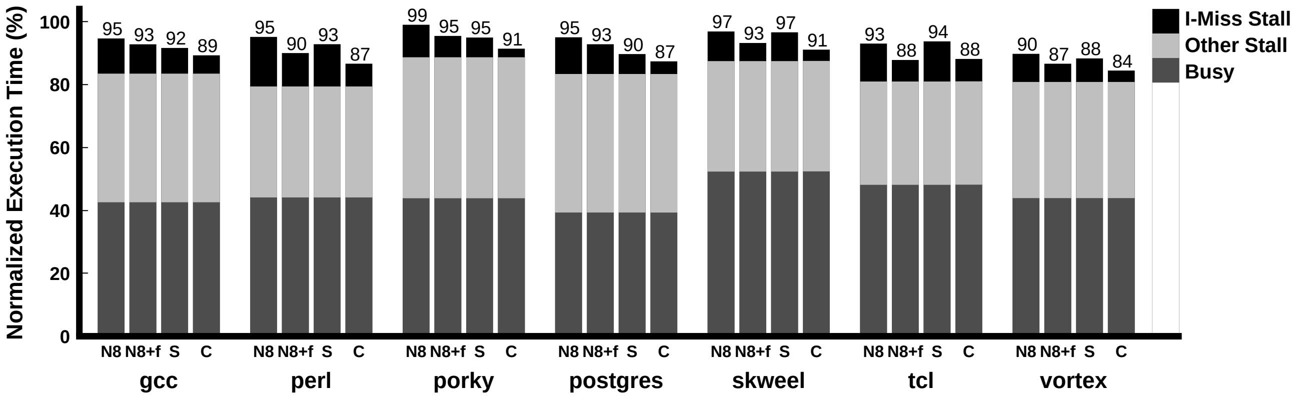

Figure 11: Performance of four different combinations of prefetch filtering and compiler-inserted prefetching (**N8** = next-8-line prefetching alone, **N8+f** = next-8-line prefetching with prefetch filtering, **S** = compiler-inserted prefetching alone without prefetch filtering, **C** = cooperative 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 11 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`, prefetching 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 non-sequential prefetches early enough) are *orthogonal*. Hence both components of our design are clearly important for performance and are complementary in nature.

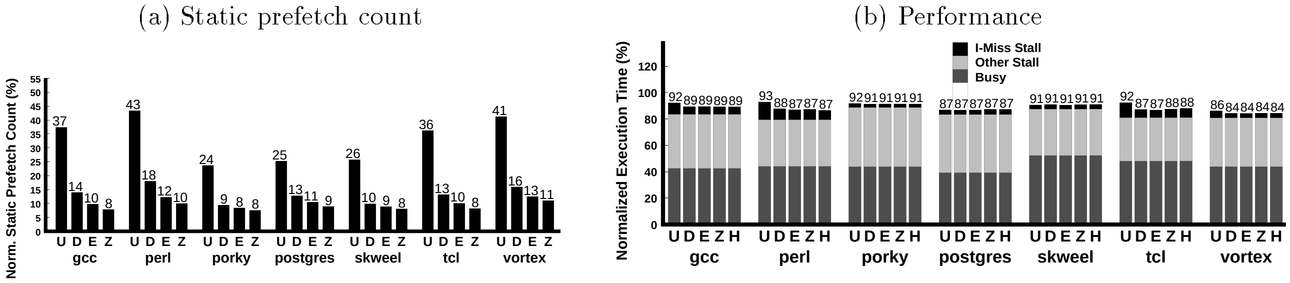

Figure 12: Impact of prefetch optimization on (a) the static prefetch count and (b) the performance of cooperative prefetching. (**U** = unoptimized, **D** = combining prefetches at dominators, **E** = case **D** plus eliminating unnecessary prefetches, **Z** = case **E** plus compressing prefetches, **H** = case **Z** plus hoisting prefetches.) The y-axis of (a) is normalized to the number of instructions in the original executable.

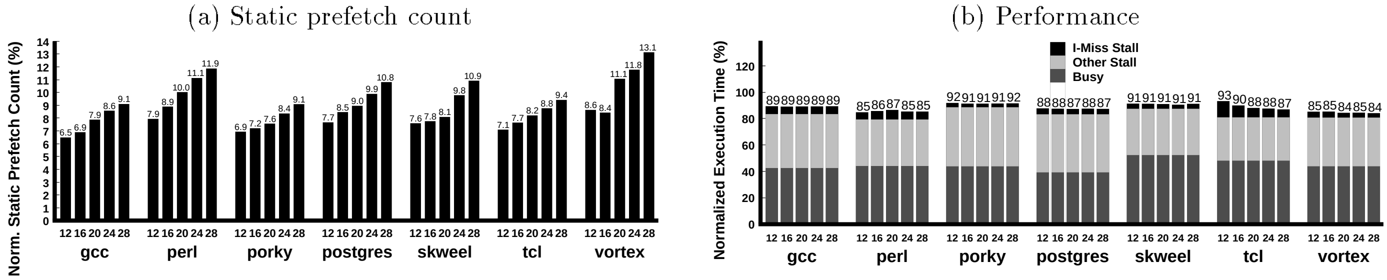

Figure 13: Impact of the prefetch-scheduling distance on (a) the static prefetch count and (b) the performance of cooperative prefetching. (xx = a prefetch-scheduling distance of xx instructions is used in the compiler scheduling; the case **20** is the default for our basic cooperative prefetching.) The y-axis of (a) is normalized to the number of instructions in the original executable.

6.4 Impact of Prefetching Optimizations

To evaluate the effectiveness of the compiler optimizations in reducing the number of prefetches, we measured their impact both on code size and performance. Figure 12(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 be bloated by over 40%. Combining prefetches at dominators (**D**) dramatically reduces the prefetch count by more than a 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 12(a).) Altogether, the prefetch optimizations limit the prefetch count to only 9% of the original code size on average.

Figure 12(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 Prefetch-Scheduling Distance

A key parameter in our prefetch scheduling compiler algorithm is the *prefetch-scheduling distance* (i.e. *SCHED_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

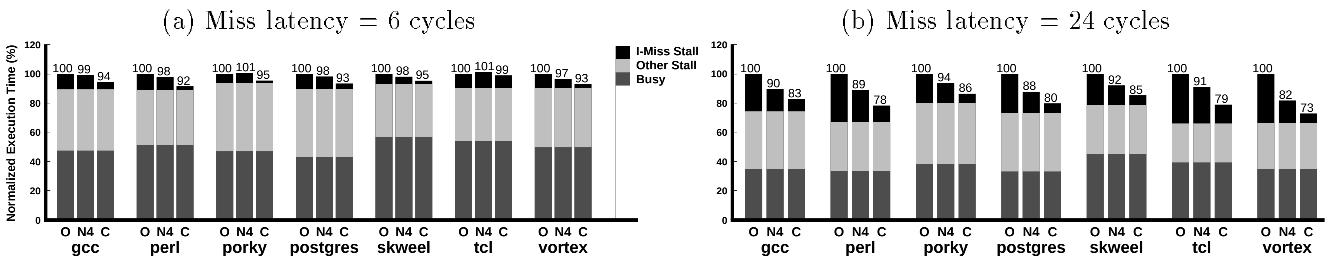

Figure 14: Impact of varying the cache miss latency. (**O** = original, **N4** = next-4-line prefetching, **C** = cooperative prefetching.)

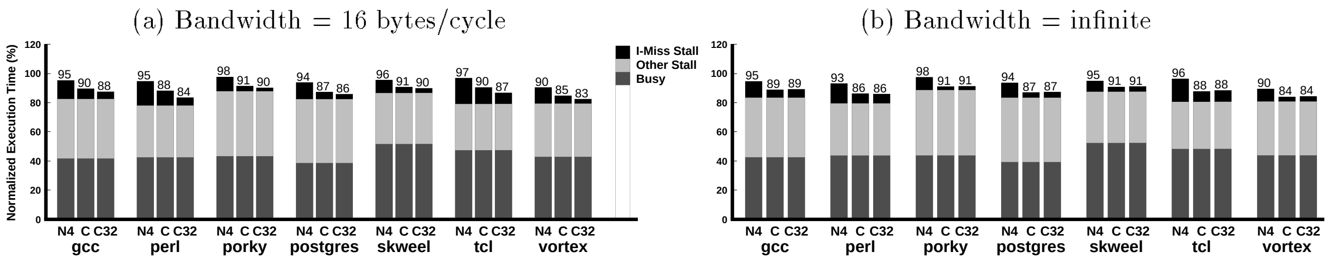

Figure 15: Impact of varying the bandwidth between the I-cache and L2 cache. (**N4** = next-4-line prefetching, **C** = cooperative prefetching, **C32** = cooperative prefetching with the original bandwidth which is 32 bytes/cycle).

latency, but setting the parameter too high can increase the code size (since more prefetches 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 prefetch-scheduling 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 prefetch-scheduling 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 13(a) and 13(b), respectively).

As we observe in Figure 13(a), increasing the prefetch-scheduling distance can result in a noticeable increase in the code size. Fortunately, even with a prefetch-scheduling 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 prefetch-scheduling distance, as we see in Figure 13(b). While `tcl` enjoys a 6% speedup as we increase this 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 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 that in our experiments so far, the primary-to-secondary miss latency has been 12 cycles (plus any delays due to contention). Figure 14 shows the performance of next-4-line and cooperative prefetching when this parameter is decreased to 6 cycles and increased to 24 cycles. (Note that the compiler's prefetch-scheduling distance was set to 12 and 28 instructions, respectively, for the 6-cycle and 24-cycle cases.) As we see in Figure 14, 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

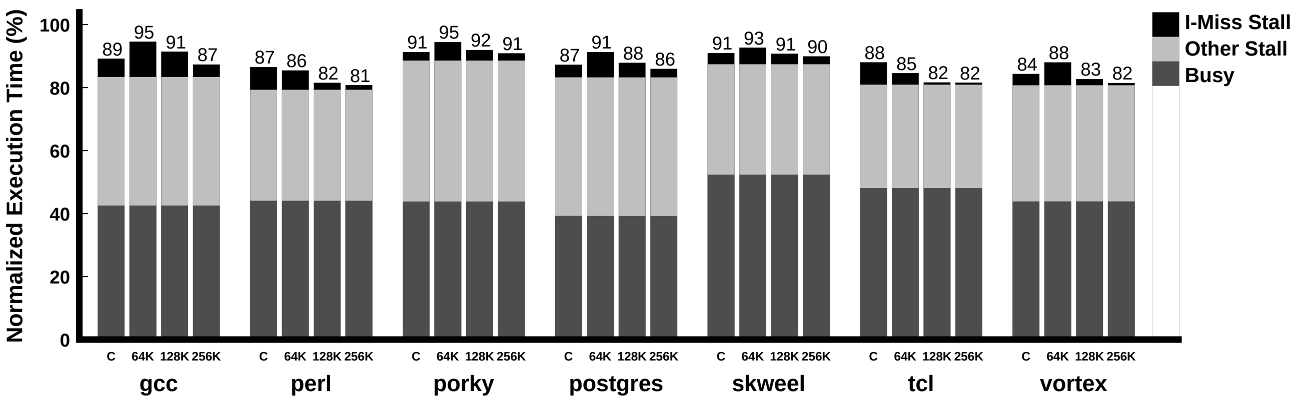

Figure 16: Performance comparison of cooperative prefetching and larger I-caches (**C** = a 32 KB I-cache with basic cooperative prefetching, xx**K** = a xx KB I-cache without prefetching.) The y-axis is normalized to the execution time of a 32 KB I-cache without prefetching.

average speedup of 24.4%, which is double the average speedup of next-4-line prefetching (12.2%).

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 15 shows the impact of decreasing this bandwidth to 16 bytes/cycle, and increasing it to unlimited bandwidth. (Note that the **C32** case—cooperative prefetching with the original bandwidth of 32 bytes/cycle—is include on the same axis simply as a point of comparison.) There are two things to note from Figure 15. First, we see in Figure 15(a) that while reducing the bandwidth does degrade the performance of cooperative prefetching somewhat—from an average speedup of 13.3% to 12.5%—the overall performance gain still remains high. Hence cooperative prefetching can achieve good performance with realistic amounts of bandwidth. (Note that this bandwidth includes servicing data cache misses as well.) Second, in Figure 15(b) we observe 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.).

6.7 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 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 8 KB for the 2-bit saturating counters added to the L2 cache. (We do not count the storage for prefetching indirect jumps because they are not used in basic cooperative prefetching.)

Figure 16 compares the performance of a 32 KB I-cache with cooperative prefetching with that of three larger I-caches, ranging from 64 KB to 256 KB, 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 32 KB to 64 KB (10.8%) despite of the substantially higher hardware cost of the larger cache. In addition, cooperative prefetching outperforms the 128 KB I-cache in three of the seven applications, and is within 2% of the performance with a 256 KB 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.

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 non-sequential 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

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] D. Bernstein, D. Cohen, A. Freund, and D. E. Maydan. Compiler techniques for data prefetching on the PowerPC. In *PACT'95*, June 1995.

[3] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA'97*, June 1997.

[4] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA'90*, pages 364–373, May 1990.

[5] D. Kaeli and P. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *ISCA'91*, pages 34–42, May 1991.

[6] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[7] A. Maynard, C. Donnelly, and B. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commerical workloads. In *ASPLOS-VI*, pages 145–156, October 1994.

[8] J. Pierce and T. Mudge. Wrong-path prefetching. In *MICRO-29*, December 1996.

[9] V. Santhanam, E. Gornish, and W.-C. Hsu. Data prefetching on the HP PA8000. In *ISCA'97*, June 1997.

[10] A. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(2):7–21, 1978.

[11] A. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, Sept. 1982.

[12] J. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Supercomputing'92*, pages 588–597, 1992.

[13] C. Xia and J. Torrellas. Instruction prefetching of system codes with layout optimized for reduced cache misses. In *ISCA'96*, pages 271–282, June 1996.

[14] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.

[15] A. Yu and J. Chen. *The Postgres95 User Manuel v1.0.* University of California at Berkeley, Sept 1996.