Branch History Guided Instruction Prefetching

Viji Srinivasan, Edward S. Davidson, Gary S. Tyson
ACAL, EECS Department
University of Michigan
Ann Arbor, MI 48105
{sviji, davidson, tyson}@eecs.umich.edu

Mark J. Charney,* Thomas R. Puzak
IBM T J Watson Research Center
Yorktown Heights
New York, NY 10598
trpuzak@us.ibm.com

Abstract

Instruction cache misses stall the fetch stage of the processor pipeline and hence affect instruction supply to the processor. Instruction prefetching has been proposed as a mechanism to reduce instruction cache (I-cache) misses. However, a prefetch is effective only if accurate and initiated sufficiently early to cover the miss penalty. This paper presents a new hardware-based instruction prefetching mechanism, Branch History Guided Prefetching (BHGP), to improve the timeliness of instruction prefetches. BHGP correlates the execution of a branch instruction with I-cache misses and uses branch instructions to trigger prefetches of instructions that occur $(N - 1)$ *branches later in the program execution, for a given* $N > 1$. *Evaluations on commercial applications, windows-NT applications, and some CPU2000 applications show an average reduction of 66% in miss rate over all applications. BHGP improved the IPC by 12 to 14% for the CPU2000 applications studied; on average 80% of the BHGP prefetches arrived in cache before their next use, even on a 4-wide issue machine with a 15 cycle L2 access penalty.*

1. Introduction

As processor speeds have increased over the past few decades, the gap between memory access latency and the processor cycle time has steadily increased. Consequently, the performance penalty of an I-cache miss has increased. In addition, current superscalar processors fetch and issue multiple instructions per cycle which compounds the miss penalty (measured in terms of lost instruction headway). Reducing I-cache misses is therefore critical for restoring a balance between the fetch and the execution stages of these wide issue processors.

*Currently at Transmeta, Santa Clara, CA 95054, mcharney@transmeta.com

The common solution for reducing I-cache misses is to increase the cache size. However, the size of the I-cache is limited by timing and area considerations, and may also be reaching a point of diminishing returns. Prefetching would be a more viable alternative, if an effective predictor of instruction miss addresses were available.

Although many instruction prefetching techniques achieve good miss coverage with sufficient accuracy, most existing techniques generally fail to issue prefetches early enough to cover the access latency of the next level of hierarchy; hence many prefetched lines do not arrive in cache before they are referenced resulting in a "delayed hit" that suffers some portion of the full miss penalty. With the current trends toward wider issue processors and an increasing gap between the CPU cycle time and the memory access latency, to avoid a fetch bottleneck it is critical to issue prefetches earlier. In this paper we present an instruction prefetching technique where our results show that over 80% of the prefetched lines arrived in cache before they were referenced, even on a 4-wide issue machine with a 15 cycle L2 access penalty.

Prior hardware-based instruction prefetching techniques can be broadly classified into sequential and non-sequential techniques. The sequential techniques [7, 14] prefetch one or more physically contiguous (sequential) cache lines from the memory and are easy to implement. Although they achieve good miss coverage, we show that the prefetches are not issued early enough to cover the access latency of the L2 cache. Moreover, these techniques do not attempt to cover miss penalties associated with *taken* branches. The non-sequential techniques [4, 12, 15, 17] are closely tied to branch prediction. The objective of these techniques is to predict the addresses of instructions that will be executed after 1 or more branches, at least one of which is a *taken* branch. To predict prefetch addresses past 2 or more branches, these techniques rely on having a branch predictor that predicts the outcome of multiple branches in one cycle. Such predictors are complex to implement and have lower prediction accuracy [15].

In this paper we propose a new hardware-based non-sequential prefetching technique, “Branch History Guided Prefetching” (BHGP), that uses branches as trigger points to initiate prefetches to “candidate blocks.” The prefetch candidate blocks contain instructions that resulted in I-cache misses $(N - 1)$ branches after some previous execution of the triggering branch. Unlike prior non-sequential techniques, BHGP selects its prefetch candidate block without needing a branch predictor to predict the outcome of each of these N branches.

Furthermore, despite the fact that the target address of a branch instruction[1] is the beginning of a basic block which may span multiple cache lines, most of the prior techniques prefetch only the cache line containing the target address; these techniques rely on next sequential prefetching [7, 14] to prefetch the remaining lines of the basic block. However, BHGP maintains both the address and the *length* of prefetch candidate blocks, so that entire blocks can be prefetched in a timely fashion.

In our evaluations, BHGP on average eliminates 66% of the I-cache misses for some important commercial and windows-NT applications and some applications from the CPU2000 suite that have high I-cache misses. BHGP improves IPC by 12 to 14% for the CPU2000 applications studied. For the commercial and windows-NT applications over 80% of the BHGP prefetches are issued 20 or more instructions before their next use.

The rest of this paper is organized as follows: Section 2 presents the Branch History Guided Prefetching technique (BHGP). Section 3 describes the details of the simulation environment and the benchmarks used for this study. Section 4 presents the idealized performance using branches as prefetch triggers and varies the lookahead, N, to show the effect on miss coverage. Results using BHGP are presented in section 5. Related instruction prefetching work is presented in section 6, and conclusions in section 7.

2. Branch History Guided Prefetching

Branch History Guided Prefetching (BHGP) exploits a correlation between the execution of a branch instruction and later I-cache misses. This correlation is not unexpected since control flow changes caused by branches lead to many I-cache misses. BHGP identifies those branches that are followed by I-cache misses at an appropriate later time and exploits the regularity of this correlation to prefetch some “candidate block” of instructions.[2] For example, a branch instruction (Br_1) will be associated with candidate block (BB_1) if there is an I-cache miss to BB_1 exactly $(N - 1)$ branches after Br_1 is executed. When Br_1 is next executed, a prefetch for BB_1 is initiated.

Figure 1 illustrates the operation of BHGP using a high level block diagram. The prefetch hardware consists of 5 structures: Prefetch Table (PT), Branch History Queue (BHQ), and three registers: BB, L, and M. The PT is a small associative cache. Each entry of the PT contains the address of a branch instruction, the beginning address of its associated prefetch candidate block, and the length of the block (in cache lines). The BHQ is maintained as a FIFO buffer and always holds the addresses of the most recent N branches executed ($N = 5$ in Figure 1). Whenever a branch is executed by the processor it is enqueued to the tail of the BHQ and the entry at the head of the BHQ is pushed out (dequeued). The BB register holds the address of the instruction that followed the most recently executed branch (potentially the beginning address of the prefetch candidate block for the branch at head of the BHQ). For example, in Figure 1, Br_5 is the most recent branch and BB_18 is the address of the instruction executed after Br_5; BB_18 may become the beginning address of the next prefetch candidate block to be associated with the branch at the head of the BHQ, namely, Br_1 (presently BB_12 is associated with Br_1). The L register stores the number of I-cache lines referenced since the most recent branch and will be used as the length of the prefetch candidate block. The M register is a 1-bit flag which is reset to 0 whenever a branch instruction is executed and is set to 1 whenever there is an I-cache miss.

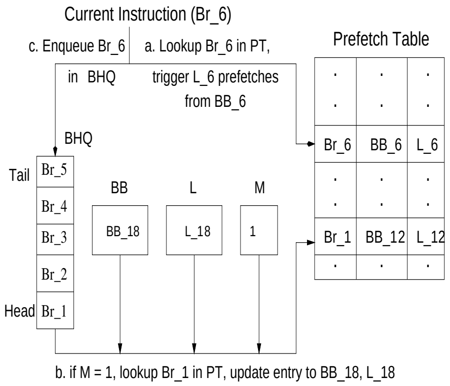


Figure 1. BHGP Hardware and Operation

To illustrate the operation of BHGP, consider the sequence <Br_1, Br_2, Br_3, Br_4, Br_5, Br_6> of six branches[3] in an execution. Each branch may be *taken* or *not taken* and the BHQ may contain multiple instances of the same branch. Figure 1 shows the state of BHQ some

[1] The phrase “target address of a branch instruction” is used informally to refer to the branch target address if the branch is *taken* or the fall-through address if the branch is *not taken*.

[2] A prefetch candidate block includes all instructions executed after a branch up to and including the next branch instruction. Candidate blocks are thus extended basic blocks and may not be disjoint.

[3] In BHGP, the term “branch” refers to conditional branches, jumps, function calls, and function returns.

time after Br_5 is executed, but before Br_6; suppose that the current state of PT is as shown. BB shows that BB_18 is the block that was entered after Br_5, the L register is counting up the number of cache lines in BB_18, and $M = 1$ indicates that there was an I-cache miss when referencing some instruction in BB_18. Eventually branch Br_6 (the final instruction of BB_18) becomes the current instruction and the following three events occur: a) The PT is searched for a Br_6 entry. If there is a match, a prefetch is initiated for the entire candidate block (BB_6 of length L_6 lines in Figure 1). b) If $M = 1$, there was an I-cache miss after Br_5 and the corresponding prefetch candidate block in the BB register (BB_18) and its length in the L register (L_18) need to be associated with the branch at the head of the BHQ, Br_1 in Figure 1. If a PT entry for Br_1 already exists, it is updated to BB_18 and L_18. Otherwise, a new entry is created with this information and replaces some other PT entry. c) Br_6 is enqueued to the tail of the BHQ and Br_1 is pushed out of the BHQ. In addition, the L register is set to 1, M is reset to 0, and the BB register is updated with the address of the first instruction after Br_6. It is important to note that if no cache miss occurs after Br_5 is executed, then M is still 0 when Br_6 occurs, and the PT entry for Br_1, if any, is left unchanged. Each PT entry thus associates a branch with its *most recently missed (MRM)* candidate block, i.e., the block that experienced a miss most recently while that branch resided at the BHQ head.

In general, for 2-way branches, there are 2^N possible prefetch candidate blocks that may occur $(N-1)$ branches after the current branch, but only a very few are responsible for most I-cache misses. We will show in section 4 that BHGP can in fact capture a majority of the miss coverage by storing only one candidate block for each branch, namely its MRM candidate block, as described above.

To limit the number of useless prefetches, BHGP prefetches the MRM candidate block only if it is not already present in the cache and its "confirmation bit" is set to 1. Each line in the next level of memory hierarchy (L2 cache) has a *confirmation bit* that is used to track whether the line was referenced while it resided in L1 after it was last prefetched. The confirmation bit of a line is initialized to 1 and is reset to 0 whenever the line is prefetched and replaced in L1 without being used; it is set to 1 again only when the line experiences a demand miss. Prefetch requests are squashed if the line's confirmation bit is 0 in the L2 directory. Most prior prefetching techniques also use these two strategies to reduce the number of useless prefetches.

In this paper, we compare the performance of BHGP with the traditional NSP technique [14] and the mBTB technique [15]. These two techniques are reviewed below; the discussion of other prior work in instruction prefetching is deferred to section 6.

Next Sequential Prefetching(NSP) [14] is a simple prefetching technique in which a prefetch to the next sequential line is triggered whenever there is a cache access. To control the number of useless prefetches, a variant called tagged prefetching [7] is used. In this technique a tag bit is associated with every L1 cache line and set when the line is prefetched into cache. A prefetch to the next sequential line is triggered both on a cache miss and on a hit to a line whose tag bit is set. Once a prefetched cache line is referenced, this tag bit is reset. NSP exploits the code sequentiality found in typical programs and our results (presented in section 5) show that NSP does achieve a high miss coverage. However, the majority of NSP prefetches do not have sufficient *prefetch distance*.[4] Moreover, NSP does not attempt to prefetch the target of a *taken* branch instruction (although this may occasionally happen by coincidence).

The mBTB prefetching technique [15] is similar to BHGP in that both use branch instructions as triggers to initiate prefetches to blocks occurring several branches after the current branch. The mBTB technique uses a multilevel Branch Target Buffer (mBTB) to prefetch the target of a branch instruction that occurs $K-1$ branches past the current branch. Each entry in the mBTB holds a branch address and 2^K target addresses; each target address entry has a 2-bit saturating counter. A FIFO buffer is used to hold the most recent K branches executed, along with their targets. When a branch instruction is executed, the PC of the branch is used to index into the mBTB and one of its target addresses (among the 2^K targets) with the highest value for its 2-bit counter is returned as the target to be prefetched. Whenever the actual target subsequently seen in the program execution matches one of the 2^K targets, the 2-bit counter associated with that target is incremented; the 2-bit counters of all non-matching targets of that mBTB entry are decremented. Since the mBTB technique prefetches target lines, rather than entire blocks, it also employs NSP [14] to increase miss coverage. As the mBTB size grows rapidly with K, the value of K was limited to 3 in [15].

The major differences between BHGP and mBTB prefetching are: (1) BHGP updates its PT once for each candidate block execution that experiences one or more misses, whereas the mBTB is updated once per basic block execution even if all accesses to it are hits. (2) the mBTB stores 2^N candidate target addresses in each entry and uses 2-bit saturating counters and a tie-breaking rule to choose one target for prefetching, whereas the PT entry stores only the MRM candidate block. Thus the size of an mBTB entry is exponential in N, whereas the PT entry has constant size. (3) BHGP prefetches an entire candidate block in a timely fashion, whereas mBTB prefetching prefetches only the first line of the target block and relies on NSP, which

[4] Prefetch distance is the elapsed time between the initiation and the next use of a prefetch and is measured either in cycles or in number of instructions between the initiation and the next use of a prefetch.

tends not to be timely, to prefetch the rest.

3. Benchmarks and Simulation Environment

To evaluate BHGP on a wide range of current workloads, we selected those integer applications from the CPU2000 suite that exhibit higher I-cache misses (*gcc, crafty, perl,* and *vortex*), as well as windows-NT traces for Doom, Explorer, and Netscape, and traces of the commercial database applications, TPCC and TPCD.

Our experiments showed that I-cache misses are not the primary bottleneck in most of the CPU2000 suite. To understand the finite I-cache effect, we simulated a system with a perfect I-cache and compared the performance to system with finite I-caches (8KB, 16KB, and 32KB). Four benchmarks, *gcc, crafty, perl*, and *vortex*, showed a 25 to 30% performance penalty due to finite I-caches; for each of the other CPU2000 benchmarks the penalty was only 1 to 3%. Hence, we selected only these four applications from the CPU2000 suite detailed cycle level simulations using the Simplescalar toolset [2]. Table 1 presents the microarchitectural parameters used for the Simplescalar simulations.

Fetch, Decode & Issue Width	4
Inst Fetch & L/S Queue Size	16
Reservation stations	64
Functional Units	4add/2mult
Memory system ports to CPU	4
L1 I and D cache each	16KB, 2-way, 32byte
L1 cache access time(cycles)	1
Prefetch buffer	2KB, 4-way, 32byte
Prefetch buffer access time(cycles)	1
Prefetch Table (level-1)	4KB, 8-way, 8byte
L1 prefetch table access time(cycles)	1
Unified L2 cache	256KB, 2-way, 32byte
L2 cache access time(cycles)	15
Mem latency (cycles)	30
Branch Predictor	2-lev, 2K-entry

Table 1. Microarchitecture Parameters for Simplescalar Simulations of CPU2000

For the database workloads, TPCC and TPCD, we used traces collected on an RS/6000 machine by the microprocessor research group at the IBM T. J. Watson Research Center. For the windows-NT applications, we used traces collected using a PC simulator based on Bochs [16]. Bochs [1] emulates the entire machine platform so that it can support the execution of a complete operating system and applications that run on it. This approach allows access to all operating system events in addition to standard instruction, data, and branch traces. To collect the traces we chose out-of-the-box Windows NT 4.0 (Build 1381) as the operating system for the virtual PC simulator. The three NT applications we studied are: Id's Doom, Microsoft Explorer 5.0, and Netscape 4.0. Doom is a first-person type combat game; the run of Doom included recording a session of a Doom game and then replaying it on the PC simulator. Both Microsoft Explorer 5.0 and Netscape 4.0 are web browsers; our input is a set of three HTML pages: the CNN web page, an ESPN web page, and the University of Michigan's homepage.

As we did not have access to cycle-accurate timing simulators for the commercial and windows-NT applications, we developed a functional cache simulator to evaluate them. We varied the I-cache size from 32KB to 256KB, the associativity from 1 to 4, and the line size from 32 to 128 bytes. As the trends of the results remained the same, we present results for only one of the configurations, namely a 32KB, 2-way associative, 32 byte lines L1 I-cache. To eliminate cache pollution due to prefetching, we used a 2KB, 4-way associative prefetch buffer along with the I-cache.

Name	**Instructions** (in millions)	**Branches** (in millions)	**Average Basic block** *(instructions)*
tpcc	172	29	5.8
tpcd	58	10	5.9
doom	751	110	6.8
explorer	407	66	6.1
netscape	864	156	5.5
gcc	2,000	320	6.2
crafty	2,000	213	9.4
perl	2,000	346	5.8
vortex	2,000	330	6.1

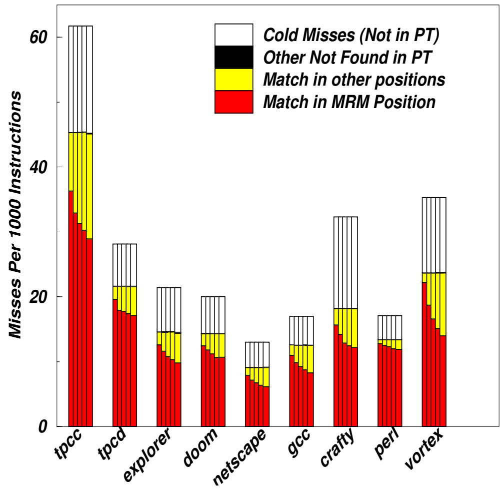
Table 2. Benchmark Characteristics

The characteristics of the benchmarks used in this study are summarized in Table 2. The first 2 billion instructions of the CPU2000 applications are used in our evaluations. Although the TPCC and TPCD application traces contain only 172 and 58 million instructions, respectively, they have the highest I-cache miss rate. Even with an I-cache of size 256KB, these relatively short traces have a high 3 to 4% miss rate.

4. Prefetch Address Selection

In this section, we discuss the effectiveness of branches as prefetch triggers for candidate blocks occurring $N - 1$ branches later, and show that BHGP can capture a majority of the potential miss coverage by prefetching only MRM candidate blocks. For this analysis, we used a conventional I-cache to count the number of misses without prefetching, a BHQ with depth N varying from 1 to 5 to study the effect of increasing the lookahead, and an infinite PT to record for *every* branch instruction an LRU-ordered stack of *all* the candidate blocks that ever resulted in an I-cache miss $(N-1)$ branches later. We limit BHQ depth to 5 because our experiments showed that 5 branch lookahead was generally sufficient to cover L2 latencies of 15 to 20 cycles.

On an I-cache miss, the block being referenced is associated with the BHQ_head_branch as a prefetch candidate block. The PT is then searched for a possible match. A match in the PT, implies that the BHQ_head_branch was already associated with this candidate block. Otherwise, this candidate block is added to the list of candidate blocks associated in the PT entry for the BHQ_head_branch (if no such entry is found, one is created). The list of candidate blocks of each PT entry is maintained as an LRU stack so that we can also count the number of matches to the top-of-stack (MRM) candidate block of the PT entry.



(In each group of bars from left to right: N = 1, 2, 3, 4, 5)

Figure 2. BHGP's Potential Miss Coverage

Each bar of Figure 2 shows the number of I-cache misses per 1000 instructions and their distribution into 4 categories. Each miss results from a reference to some candidate block (T) that occurs while some branch (Br) is the BHQ_head_branch. "Cold Misses (Not in PT)" are the I-cache misses that result from first reference to block T. "Other Not Found in PT," are I-cache misses which were not found in the PT entry for Br; however, since there must have been at least one previous miss to block T, it must be associated with at least one other branch in this infinite PT at the time that this miss occurs. "Match in MRM position" and "Match in other positions" account for the matches in the candidate block stack of the PT entry for Br. The "Cold Misses" category constitutes the vast majority of the misses not found in the accessed entry of the infinite PT; the "Other Not Found in PT" category is insignificant (fewer than 1%).

Figure 2 shows that about 68% of the misses on average are matches in some position of the candidate list of the PT entry and this percent remains essentially unchanged with increases in the lookahead, N. On average 61.2% of the misses are matches in the MRM position for $N = 1$, which degrades to 48.4% for $N = 5$. However, in order to trigger prefetches in a timely fashion for an L2 latency of 15, we accept this small degradation in coverage and use $N = 5$ for the rest of this paper. Furthermore, Figure 2 shows that we can gain most of the advantage of an infinite PT by listing only the MRM candidate in each entry, thereby justifying BHGP's choice of a low-cost PT. The achieved miss coverage may actually be higher than the "Match in MRM Position" category because although the prefetched candidate block may not be executed $(N-1)$ branches after the current branch, it may nevertheless remain in the cache until it is eventually used. In fact, our results in section 5 show that BHGP with $N = 5$, MRM candidate only, and 2000 entries, actually achieves on average a 66% reduction in the I-cache misses.

```
BB_1: 0x1200f71b4:ldwu  s3, 0(s1)
      0x1200f71b8:ldq_u zero, 0(sp)
      0x1200f71bc:ldq_u zero, 0(sp)
      0x1200f71c0:cmpeq s3, 0x36, v0
      0x1200f71c4:cmpeq s3, 0x37, t0
      0x1200f71c8:cmpeq s3, 0x71, t1
      0x1200f71cc:cmpeq s3, 0x70, t2
      0x1200f71d0:bne   v0, 0x1200f71e8 <-- Br_1 (PREFETCH TRIGGER)
BB_2: 0x1200f71d4:bne   t0, 0x1200f71e8 <-- Br_2
BB_3: 0x1200f71d8:ldq_u zero, 0(sp)
      0x1200f71dc:ldq_u zero, 0(sp)
      0x1200f71e0:bne   t1, 0x1200f71e8 <-- Br_3
BB_4: 0x1200f71e4:beq   t2, 0x1200f71f0 <-- Br_4
BB_5: 0x1200f71e8:ldq   s1, 8(s1)
      0x1200f71ec:br    zero, 0x1200f71b4 <-- Br_5
BB_6: 0x1200f71f0:ldwu  s3, 0(s1)
      0x1200f71f4:ldq   a1, 80(s6)
      0x1200f71f8:bis   zero, s1, a0
      0x1200f71fc:cmpeq s3, 0x39, s3
      0x1200f7200:beq   s3, 0x1200f7208 <-- Br_6
BB_7: 0x1200f7204:bsr   ra, canon_reg <-- Br_7
BB_8: 0x1200f7208:ldq   s0, 72(s6) <-- MRM block
      0x1200f720c:addl  s5, 0x1, s5
      0x1200f7210:cmplt s5, s4, s3
      0x1200f7214:lda   s0, 80(s0)
      0x1200f7218:ldq_u zero, 0(sp)
      0x1200f721c:ldq_u zero, 0(sp)
      0x1200f7220:stq   s0, 72(s6)
      0x1200f7224:bne   s3, 0x1200f7094 <-- Br_8
```

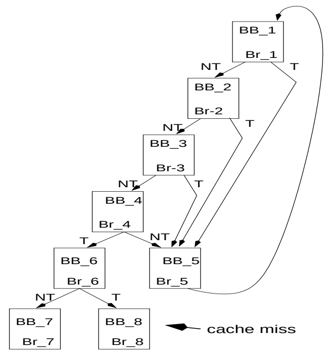


Figure 3. Code Fragment and CFG from gcc

The code fragment from *gcc* and its corresponding control flow graph, shown in Figure 3 is used to illustrate an execution pattern seen in these applications that is well-suited for retaining only the MRM candidate block in PT. There are 8 basic blocks (BB_1 through BB_8); the branch instructions at the end of these basic blocks are labeled Br_1 through Br_8, respectively. Our experiments showed that Br_1 was executed 80,284 times and the only candidate block that had an I-cache miss 4 branches after Br_1 was BB_8, which missed 71,521 times. The branch Br_6 is always *taken* and hence the candidate block BB_7 is never reached from Br_1. Although BB_1 through BB_5 can all occur 4 branches past Br_1, none of them were ever associated with Br_1 in the PT since these blocks did not miss 4 branches after Br_1. Thus Br_1 was associated only with candidate block BB_8 for the entire run of the application.

Our analysis of these applications overall showed that for N = 5, an average of 62% of the branches in the PT had only one candidate block ever associated with them, and 91% of the branches had no more than 4 candidate blocks. Although we might expect 2^5 candidate blocks for a typical branch, only a few of these actually occur and result in I-cache misses.

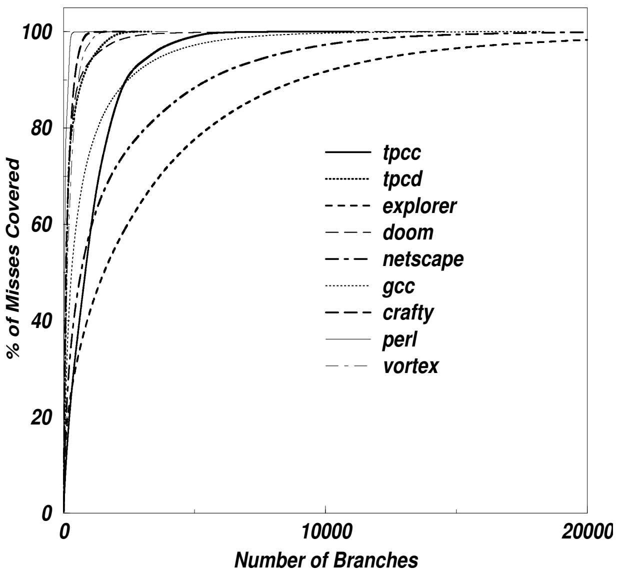


Figure 4. Branches as Prefetch Triggers

Having shown that for $N = 5$ the MRM candidate block is sufficient to cover 48.4% of the I-cache misses, we now show the number of trigger branches required to achieve this coverage. Figure 4 shows the miss coverage *vs.* the number of trigger branches required in the PT to achieve that coverage.

From Figure 4 it is clear that although the number of branches required is different for each application, about 90% of the misses are covered using at most 2000 branches for all the applications except *explorer* and *netscape*. This indicates that the PT can be implemented as a small hardware cache structure (similar to a TLB) as done in this paper.

5. Performance of BHGP

The simulation results for BHGP are presented using $N = 5$ and the standard PT where each entry lists only the MRM candidate block. For comparison, we also simulate, ORACLE, an idealized version of BHGP that employs a perfect oracle to choose the correct candidate block that will occur 4 ($= N - 1$) branches after the current branch. ORACLE uses an infinite PT, as in section 4, to squash the prefetch if the correct candidate block has never yet been associated with the current branch. We also compare BHGP performance with the traditional NSP technique [14] and the mBTB[5] technique [15]. As mentioned in section 2, the mBTB technique also uses NSP to trigger sequential prefetches. To reduce the number of useless prefetches, we maintained a confirmation bit for each L2 cache line (as explained in section 2) for both the mBTB and the NSP techniques as well for BHGP.

5.1. Misses

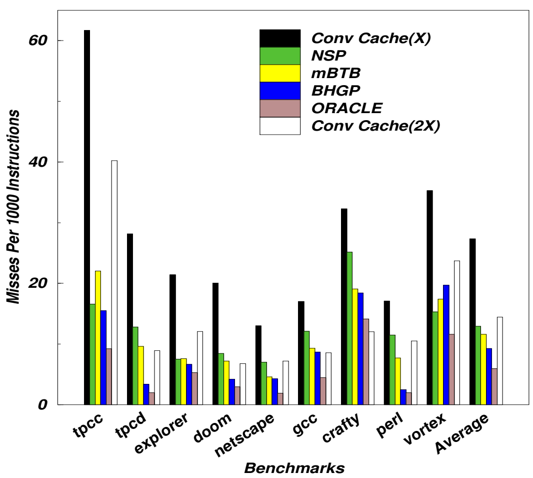


Figure 5. Instruction cache misses

Figure 5 presents the misses per 1000 instructions for all the applications. BHGP achieves an average of 66% reduction in the overall misses relative to a conventional cache of the same size (X), without prefetching. Figure 5 also shows that except for *crafty*, BHGP results in fewer misses than a conventional cache of twice the size (2X). As

[5]The mBTB in [15] used $N = 3$; to make the comparison fair we used $N = 5$ for the mBTB technique as well. Our analysis showed that mBTB with $N = 5$ always outperformed mBTB with $N = 3$.

in prior work [8, 11], we observe that NSP is quite effective in reducing I-cache misses. However, a major drawback of NSP is that a large fraction of the useful prefetches are not timely (see section 6.2). Furthermore BHGP has fewer misses than NSP for all the applications except *vortex*. Our analysis of the branches in *vortex* showed that a clear majority of them were fall-through branches, which explains NSP's high miss coverage. Figure 5 shows that for *tpcd*, *doom*, and *perl*, BHGP achieves over a 50% reduction in misses relative to NSP; for the rest of the applications (except vortex) BHGP achieves a 10 to 30% reduction in misses relative to NSP.

BHGP outperforms mBTB [15] for all the applications (except *vortex*). Although for *vortex* both NSP and mBTB have fewer misses than BHGP, we show in section 6.2 that a majority of the NSP and the mBTB prefetches are not timely while a majority of the BHGP prefetches are timely and arrive before their next reference.

ORACLE has 78% fewer misses on average than the same size conventional cache. However, keeping more than one candidate block per PT entry would require some implementable strategy to select which one to prefetch. The PT used in BHGP is economical and simple to use with entries for only a relatively small number of branches, and still achieves about 66% fewer misses than the same size conventional cache.

5.2. Prefetch Timeliness

Figure 6 shows the results of detailed cycle-level simulations that we used to evaluate the CPU2000 applications. In each group of bars, we present from left to right the number of prefetches using NSP, mBTB, and BHGP, respectively. The prefetches in each bar are divided into 3 categories according to the outcome of the next reference to the prefetched line. A prefetched line is classified as a *hit* if its next reference finds it in cache; if the line has not yet arrived in cache, it is a *delayed hit*; if it is replaced before being referenced it is classified as *useless*. Thus hits represent timely prefetches; delayed hits are those that do not have sufficient prefetch distance. Useless prefetches are those that should have been issued later or not at all.

Figure 6 shows that on average 35% of the NSP prefetches, but only 10% of the BHGP prefetches result in delayed hits. Although both mBTB and NSP achieve better miss reduction for *vortex* (see Figure 5), a high fraction of their useful prefetches result in delayed hits, as shown in Figure 6. Although BHGP has the lowest miss reduction for *vortex*, it delivers its prefetched lines in a far more timely fashion, resulting in the fewest delayed hits and the most hits among these three techniques.

Figure 6 also shows that on average only 2% of the NSP prefetches are useless. By contrast, 10% of the BHGP

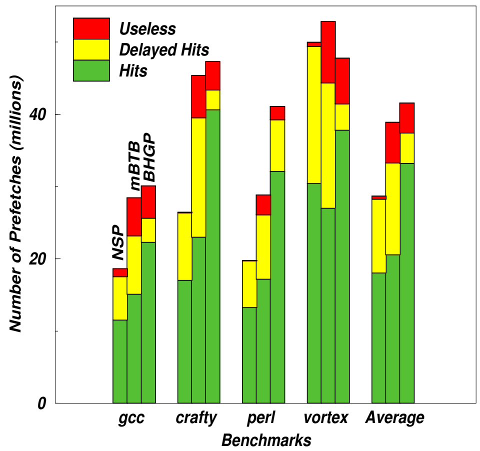


Figure 6. Prefetch Timeliness for CPU2000

prefetches and 14% of the mBTB prefetches are useless. BHGP and mBTB are more aggressive and issue more prefetches than NSP in order to increase miss coverage; however, the cost of this benefit is their higher fraction of useless prefetches.

Name	IPC				
	No Pref	NSP	mBTB	BHGP	Perfect
gcc	1.202	1.313	1.326	1.378	1.532
crafty	1.196	1.339	1.365	1.456	1.811
perl	1.350	1.463	1.522	1.624	1.683
vortex	1.107	1.328	1.288	1.336	1.673

Table 3. IPC for CPU2000 Applications

Table 3 shows the overall improvement in the IPC. To quantify the performance loss due to a finite I-cache, we also simulated a system with perfect I-cache. On average BHGP achieves 86% of the perfect cache IPC, and improves the IPC by 19% over a conventional cache, 10% over NSP, and 5.3% over mBTB. Although mBTB prefetching performs better than NSP for three of the four applications, it requires keeping 2^N targets for N branch look ahead. For *vortex*, although mBTB has the highest miss coverage, it has the most useless prefetches and a high fraction of its useful prefetches were delayed hits, causing mBTB to have a lower IPC than NSP alone.

We now consider the prefetch timeliness of the commercial and the windows-NT applications. As we did not have access to cycle accurate timing simulators for these applications, we use "prefetch distance" to measure the timeliness of the prefetches. *Prefetch distance* is the elapsed time between the initiation and the next use of a useful prefetch; it can be measured in cycles (using a cycle level simulator) or can be approximated by counting the number of instructions (using a functional simulator) between a prefetch trigger and the next (hit or delayed hit) use of the prefetched line. For the commercial and windows-NT applications, prefetch distance is assessed only in terms of instructions. Figure 7 shows the prefetch distance for the windows-NT and the commercial applications. The curves shown are cumulative distributions where the point (x, y) indicates that y percent of all useful prefetches have a prefetch distance of x or more instructions. For comparison we also show the prefetch distance averaged over all the CPU2000 applications.

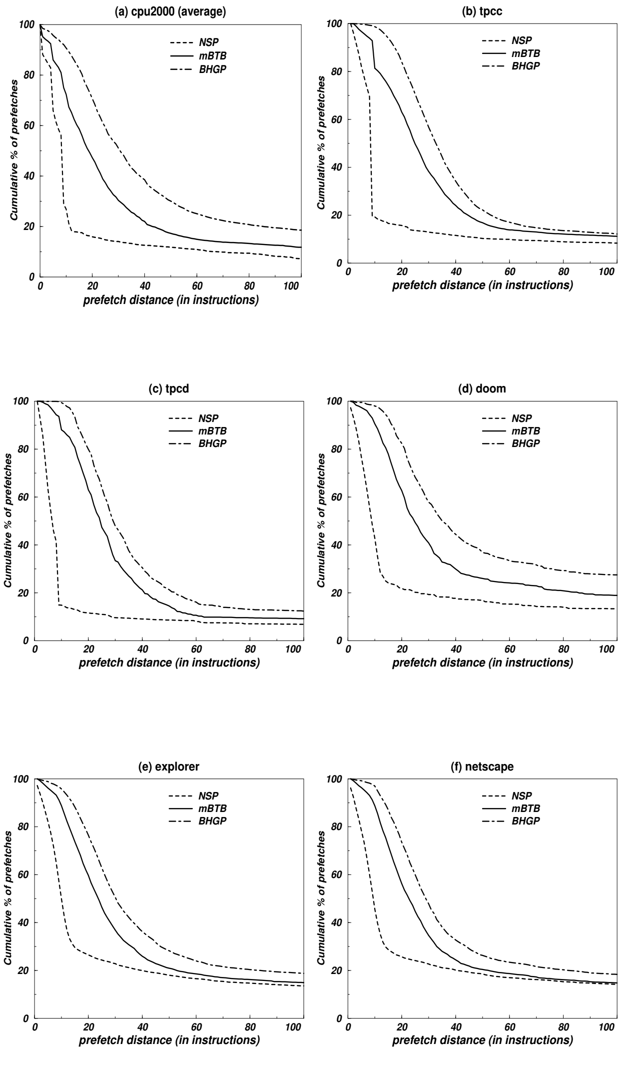


Figure 7. Prefetch Distance

Figures 7 (b) and (c) show the prefetch distance for the TPCC and TPCD applications. On average over 80% of the BHGP prefetches, but only 14% of the NSP prefetches, and 63% of the mBTB prefetches have a prefetch distance of 20 or more instructions. NSP has a sharp knee; it is relatively flat until reaching a prefetch distance of about 8 instructions. As these applications use an instruction size of 4-bytes, a 32 byte cache line holds 8 instructions. For sequential code, a prefetch distance of 8 instructions implies prefetching the next sequential line soon after the first access to the previous cache line. Although mBTB significantly improves the prefetch timeliness relative to NSP, BHGP offers significant further improvement, particularly around the critical 20 instruction distance.

Figures 7 (d), (e), and (f), show the prefetch distance for *doom*, *explorer*, and *netscape*. On average 80% of the BHGP prefetches, but only 24% of the NSP prefetches and 60% of the mBTB prefetches have a prefetch distance of 20 or more instructions. Figure 5 showed that for *explorer* NSP performed nearly as well as BHGP in reducing misses, but Figure 7(e) shows that most of the useful prefetches of NSP are not timely.

6. Related Work

As mentioned in section 1, prior work in instruction prefetching can be broadly classified into sequential and non-sequential techniques. Sequential prefetching techniques [7, 14] (discussed in section 2) prefetch one or more physically contiguous cache lines from the memory. Prefetching more than one sequential line tends to increase the prefetch distance, but also increases the chances of cache pollution due to useless prefetches. Therefore, sequential prefetching can fail when a program contains a lot of calls to small subroutines or a large number of *taken* branches. A hardware algorithm is proposed in [6] to dynamically vary the number of sequential lines prefetched. Although the technique improves the timeliness of the prefetches, it performs well only for applications with regular access patterns.

The goal of non-sequential prefetching techniques is to overcome the limitation of NSP by predicting the addresses of instructions executed after one or more branches, given that some of them may be *taken* branches. The mBTB technique [15] was discussed in section 2; others are briefly described below.

One of the earliest non-sequential prefetching techniques is "Lookahead Data Prefetching" [5]. This technique is used for prefetching data (not instructions) and requires a support unit to trigger prefetches. The prefetching architecture uses an additional program counter called the *Look-Ahead PC* (LA-PC), which runs ahead of the instruction fetch engine. The LA-PC is used to index into a reference prediction table to predict data addresses to be prefetched. The LA-PC is incremented and maintained in the same fashion as the PC with the help of its own dynamic branch predictor. Since the LA-PC runs ahead of the regular PC, this technique increases the prefetch distance.

Branch Prediction Based Prefetching [4] applies the lookahead data prefetching technique [5] to instruction prefetching. It uses a separate branch predictor to trigger prefetches. The prefetch unit runs ahead, using a LA-PC, and prefetches instructions based on its branch predictions which it logs. The execution unit checks the logs as the branches are executed and resets the LA-PC whenever a misprediction is found.

Fetch Directed Instruction Prefetching(FDP) [13] is recent work that also uses a decoupled branch predictor to trigger instruction prefetches. FDP follows the predicted stream and enqueues prefetches from the Fetch Target Queue (FTQ). The FTQ is filled using a run-ahead branch predictor. The prefetched cache blocks are placed in a separate prefetch buffer and are moved to the I-cache only when referenced. The prefetch buffer and the FTQ are both flushed on a branch misprediction.

These three techniques [5, 4, 13] each use some form of run-ahead engine to predict branch outcomes and determine prefetch targets. Branch frequency and mispredictions thus limit the distance between the run-ahead engine and the regular processor. BHGP, however, uses history rather than a run-ahead engine, and does not employ a branch predictor to determine prefetch addresses.

Target Prefetching [8] implements a line target prediction table and, like BHGP, does not depend on a branch predictor. Conceptually each entry in the table consists of a pair: (current line address, target line address). When the PC changes to a new cache line, the table is searched for a possible match. If a match is found, the target line address is prefetched. As the line size increases, the probability of finding more than one branch instruction per cache line also increases. This in turn leads to the need to store multiple target addresses in a table entry. In addition, if one of the first few instructions accessed from a cache line is a *taken* branch, the technique fails to hide the full penalty of a miss to the target address. Flow-block Prefetching [19] generalizes [8] to predict the target of a fixed length sequence of instructions called a *flow block*. A flow-block prediction table, similar to the line target prediction table in [8], is used to determine the successor flow block which may span multiple cache lines.

Cooperative Prefetching [11] relies on software rather than hardware to trigger non-sequential prefetches. It generates target addresses early for procedure returns by using a special prefetch instruction which implicitly uses the top of the return address stack as its argument. The targets of other indirect jumps are prefetched using an indirect-target table which records the past targets of indirect jumps. One more prefetch instruction is added to the ISA which conceptually stores the PC of an indirect jump instruction; this PC is used to index into the indirect-target table and the corresponding target address is prefetched. Cooperative Prefetching also employs aggressive next-N-line prefetching. Software prefetching can improve the prefetch distance substantially due to the ability to exploit global program knowledge that may be available to the compiler. However, the prefetch instructions will increase code size and may consume useful instruction fetch bandwidth. In addition, software prefetching techniques require recompilation and are not suited to running legacy code.

Wrong-path Prefetching [12] combines NSP along with prefetching the target of the current branch instruction. Therefore, for every branch instruction executed *both* the fall-through and the target address are prefetched. Since the prefetch is triggered only after the branch instruction is fetched and decoded, the prefetch distance for the target address is very small. However, if the branch instruction is *not taken* when the taken target prefetch is initiated, but is *taken* in a subsequent execution, the prefetched target address may have arrived in cache by then and would thus be a hit provided it had not been replaced in the interim.

Correlation Based Prefetching [3] pairs two temporally related events, where the first event of a pair is the prefetch trigger and the second event is the data address to be prefetched. The first event can be an instruction address, a data address, a cache line address, etc. The pairing of events is accomplished using a horizon-and-skip model. The skip parameter indicates the number of most recent events to skip over so as to increase the prefetch distance. Selecting a skip distance is similar in spirit to selecting N in BHGP. Correlation Based Prefetching has, however, been applied to prefetch data (not instructions). Similarly, Load/Store (ld/st) Cache-Based Prefetching [9] correlates the instruction addresses of the ld/st instructions with the data addresses accessed by them and uses this information to trigger data prefetches. Updating the ld/st cache only on a miss in the data cache is similar to BHGP updating the PT only on an I-cache miss.

The Branch Address Cache(BAC) [17] predicts multiple branch target addresses every cycle by extending 2-level branch predictor [18] so as to predict multiple branches per cycle; it uses the BAC to determine the starting addresses of the target basic blocks past K branches. The technique does not use the BAC to issue prefetches to the L1 I-cache, but focuses on simultaneously fetching several non-consecutive basic blocks from the cache to the processor. BAC performance is limited by the ability to predict K branches in a single cycle and by the available I-cache bandwidth. This technique motivated us to design BHGP to prefetch entire prefetch candidate blocks rather than only their initial lines.

7. Conclusions

To improve the timeliness of prefetches we have proposed and evaluated a hardware-based instruction prefetching technique called Branch History Guided Prefetching. In this technique we use branch instructions to trigger prefetches to prefetch candidate blocks occurring $(N-1)$ branches later. The results in this paper have justified the following observations:

- I-cache miss coverage does not drop much with increased lookahead (N) and thus prefetch distance can be improved.
- Even as N increases, it is not necessary to keep track of the exponentially growing number of possible prefetch candidate blocks. We need focus only on those candidate blocks that cause I-cache misses, and we showed that the very simple selection strategy of associating only one block, the MRM candidate block, with each branch instruction reduces misses substantially and suffers only modest degradation in coverage as N is increased to provide reasonable prefetch distance.
- Non-sequential instruction prefetching techniques need not be coupled with branch predictors that predict multiple branches in one cycle, and may actually be both more effective and more economical, even when they are decoupled.

Accordingly we have proposed and evaluated a non-sequential instruction prefetching technique, BHGP, which dynamically associates each branch instruction with its MRM candidate block that occurs $(N-1)$ branches later. BHGP causes each branch execution to trigger a prefetch of its MRM candidate block. BHGP is a hardware-based technique that can run legacy binaries without recompilation. Our results over a wide range of applications (CPU2000, commercial and windows-NT applications) showed that BHGP achieved an average 66% reduction in miss rate, and more importantly over 80% of its useful prefetches arrived in cache before their next use.

8. Acknowledgements

This work has been funded by gifts from IBM and Intel and by the NSF career award MIP9734023.

References

[1] Bochs Software Company, *http://www.bochs.com*

[2] D. Burger, T. Austin, "The SimpleScalar Tool Set, Version 2.0," *Technical Report TR 1342, University of Wisconsin,* June 1997.

[3] M. J. Charney, "Correlation-Based Hardware Prefetching," *Ph.D. Dissertation, Cornell University,* August 1995.

[4] I-C. K. Chen, C-C Lee, T. N. Mudge, "Instruction Prefetching Using Branch Prediction Information," *Proceedings of the Int'l Conference on Computer Design,* October 1997, pp. 593-601.

[5] T-F. Chen, J. Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors," *IEEE Transactions on Computers,* Vol. 44, No. 5, May 1995.

[6] F. Dahlgren, M. Dubois, P. Stenstrom, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 6, No. 7, July 1995, pp. 733-746.

[7] J.D. Gindele, "Buffer Block Prefetching," *IBM Tech. Disclosure Bull.* 20, 2, July 1977, pp. 696-697.

[8] W-C Hsu, J. E. Smith, "A Performance Study of Instruction Cache Prefetching Methods," *IEEE Transactions on Computers,* Vol. 47, No. 5, May 1998, pp. 497-508.

[9] P. Ibanez, V. Vinals, J. Briz, M. Garzaran, "Characterization and Improvement of Load/Store Cache-based Prefetching," *Proceedings of the 12th Int'l Conference on Supercomputing,* July 1998, pp. 369-376.

[10] Y. Liu, D. Kaeli, "Branch-Directed and Stride-Based Data Cache Prefetching," *Proceedings of the Int'l Conference on Computer Design,* 1996, pp. 225-229.

[11] C-K Luk, T. C. Mowry, "Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors," *Proceedings of the 31st Int'l Symposium on Microarchitecture,* December 1998, pp. 182-193.

[12] J. Pierce, T. N. Mudge, "Wrong-path Prefetching," *Proceedings of the 29th Int'l Symposium on Microarchitecture,* December 1996, pp. 165-175.

[13] G. Reinman, B. Calder, T. Austin "Fetch Directed Instruction Prefetching," *Proceedings of the 32nd Annual Int'l Symposium on Microarchitecture,* November 1999, pp. 16-27.

[14] A. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer,* No. 11, Dec. 1978, pp. 7-21.

[15] A. V. Veidenbaum, "Non-sequential Instruction Cache Prefetching for Multiple-Issue Processors," *Int'l Journal of High-Speed Computing,* Vol. 10, No. 1, pp. 115-140, 1999.

[16] S. Vlaovic, E. S. Davidson, G. S. Tyson, "Improving BTB Performance in the presence of DLLs," *To appear in the 33rd Int'l Symposium on Microarchitecture,* December 2000.

[17] T-Y Yeh, D. T. Marr, Y. N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," *Proceedings of the 7th Int'l Conference on Supercomputing,* July 1993, pp. 67-76.

[18] T-Y Yeh, "Two-level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors," *Ph.D. Dissertation, University of Michigan,* 1993.

[19] H. C. Young, E. J. Shekita, "An Intelligent I-Cache Prefetch Mechanism," *Proceedings of the Int'l Conference on Computer Design,* 1993, pp. 44-49.