Execution History Guided Instruction Prefetching

Yi Zhang Steve Haga Rajeev Barua

Department of Electrical & Computer Engineering

University of Maryland

College Park, MD 20742, U.S.A

{zhangyi, stevhaga, barua}@eng.umd.edu

Abstract

The increasing gap in performance between processors and main memory has made effective instructions prefetching techniques more important than ever. A major deficiency of existing prefetching methods is that most of them require an extra port to I-cache. A recent study by [19] shows that this factor alone explains why most modern microprocessors do not use such I-cache hardware-based I-cache prefetch schemes. The contribution of this paper is two-fold. First we present a method that does not require an extra port to I-cache. Second, the performance improvement for our method is greater than the best competing method BHGP [23] even disregarding the improvement from not having an extra port.

The three key features of our method that prevent the above deficiencies are as follows. First, too-late prefetching is prevented by correlating misses to dynamically preceding instructions. For example, if the I-cache miss latency is 12 cycles, then the instruction that was fetched 12 cycles prior to the miss is used as the prefetch trigger. Second, the miss history table is kept to a reasonable size by grouping contiguous cache misses together and associated them with one preceding instruction, and therefore, one table entry. Third, the extra I-cache port avoided through efficient prefetch filtering methods. Experiments show that for our benchmarks, chosen for their poor I-cache performance, an average improvement of 9.2% in runtime is achieved versus the BHGP methods [23], while the hardware cost is also reduced. The improvement will be greater if the runtime impact of avoiding an extra port is considered.

1 Introduction

Instruction cache misses are a significant source of performance loss in modern processors. I-cache performance is especially a problem in integer and database codes [1, 5], as these types of programs tend to have less instruction locality. A very rough calculation reveals that the performance cost of an I-cache miss can be large, even if misses are uncommon. Consider a machine with a 12-cycle I-cache miss latency, running a program which exhibits an average of 1 instruction per cycle when not idle due to I-cache misses, i.e. IPC = 1. Even if the miss rate is only 2%, the percentage of cycles lost due to I-cache misses is roughly 2%*12/1 = 24%, which is not a small number.

Two current trends are likely to make the performance penalty from I-cache misses even more in the future than it is today. First, the semi-conductor road-map [20] predicts a rapidly increasing performance gap between processors and memory in the future: processor speeds are increasing at 60% a year, while memory speeds are growing at only 7% a year [6]. In this scenario, the I-cache miss latency is likely to grow. Second, since more aggressive multi-issue machines such as EPIC architectures [9] and proposed new superscalars [7, 14] have the potential to perform an increased level of computation per cycle, every idle cycle due to an I-cache miss will delay a greater portion of computation than on an older machine. Thus higher ILP makes the I-cache miss penalty larger in terms of lost instruction issue opportunities. Combined, these two trends indicate that methods to hide memory latency, which are already important, will become increasingly more useful.

Since the following terms are helpful in our discussion, we will now introduce the definitions of useful, useless and late prefetches, as they are given in the literature. When the predictor indicates that an I-cache miss will occur in the immediate future, a *prefetch* to the cache line is requested. If the prediction is accurate and timely, the cache line will be soon be accessed, resulting in a *useful* prefetch. If the access occurs before the prefetch completes, however, the prefetch is termed *late*, and the I-cache latency is only partially hidden. If the prediction is incorrect and the cache line is not accessed before it is evicted, then the prefetch is called *useless*.

Current methods for instruction prefetching suffer from deficiencies that have hindered their adoption. The 1998 survey and evaluation paper by Tse and Smith [24] presents a well-argued case for why current methods have failed. First, in some prefetching methods [22, 15], the prefetches are triggered too late to hide all the cache miss latency. Second, some methods [11] use a huge table to store the miss history information. Finally, and most importantly, most

prefetching methods including the most recent [23, 13, 18] use *cache probing* to filter out useless prefetches. The I-cache is checked before every potential prefetch, to prevent the prefetching of lines that are already in cache. Unfortunately, cache probing requires a second port to the I-cache, so as to avoid interference with the normal instruction fetches that occur on every cycle. This second port does not come for free – it usually slows down the cache hit time and increases circuit area [19, 24]. Taken together, these factors can result in a trivial, or even negative, net gain from prefetching.

It is not surprising, therefore, that current methods for instruction prefetching are not widely used. As evidence, consider that most modern processors such as UltraSparc III [12], Itanium [10] and PowerPC [17] only prefetch sequential instructions or the predicted branch targets; no sophisticated hardware-based schemes are used.

The limitations of hardware-based instruction prefetching schemes has led some recent commercial machines such as the UltraSparc III [12],Itanium [10] and PowerPC [17] to provide compiler-directed methods. In such schemes, the hardware provides prefetch instructions which are scheduled by the compiler based on program analysis, perhaps employing profiling. Unfortunately, compiler-directed methods have their own shortcomings: first, prefetch instructions themselves constitute overhead; second, not all compilers on a machine may implement prefetching; third, old binaries are not optimized; fourth, dynamic behavior may not be predictable at compile-time; and fifth, efficient portability across different implementations of the ISA is difficult because their different hardware parameters, such as cache size and latency, can affect the prefetching strategy. Consequently, we believe, and our results demonstrate, that a good hardware-based prefetching method will have a significant positive impact.

This paper presents a hardware-based instruction prefetching method that correlates execution history with cache miss history. It improves upon existing methods by triggering prefetches early enough to not have any stalls; by avoiding the extra port in the instruction cache; and by using a reasonably-sized table. while at the same time prefetching a higher percentage of cache misses.

To achieve these gains, the method uses three innovations. First, each cache miss is correlated with the instruction that was fetched a certain number before the miss; this correlation is stored in a miss history table. Prefetches are triggered when their correlated instructions are encountered in the future. Multiple execution paths leading to a particular cache-miss may result in multiple triggering instructions for that miss. Second, contiguous cache misses are grouped together and associated with one preceding instruction. This makes it possible to use a miss history table of reasonable size. Third, efficient prefetch filtering methods are used to

reduce useless prefetches. Thus it becomes unlikely that the prefetched line will already be present in the cache, so that the instruction cache does not require a second port. For example, a confidence-counter strategy is used to retire unsuccessful correlations.

Our method is evaluated on a suite of applications for which instruction cache performance is known to be a problem. Such behavior is common with irregular or database programs. Our method is evaluated against several existing strategies, which are also implemented for comparison. We found that, on average, 71.3% of the I-cache misses are correctly prefetched by our method, and 77.6% of these are prefetched early enough to completely hide the cache miss latency. As a result, the runtimes of applications are improved by about 35%.

The rest of this paper is organized as follows. Section 2 presents further details on the execution history guided prefetching technique. Section 3 discusses some related work on instruction prefetching. Section 4 describes the details of the simulation environment and the benchmarks used for this study. Section 5 presents our experiment methodology and our experimental results. Finally in section 6 is conclusions.

2 Method

The basic method in this paper is now described. Let the *prefetch distance*, be a constant that is slightly greater than the miss latency from the L1 to the L2 I-cache, measured in cycles (typically 10-30 cycles for modern processors). The average number of instructions that will execute within this *prefetch distance* is N. When an address misses in the I-cache, then the instruction that was fetched N instructions prior to this miss, called the Nth *previous instruction* is stored in a miss history table. If the Nth *previous instruction* is encountered again, and if the cache line in question has been evicted in the meantime, a prefetch will be triggered; this eviction information is indicated by the miss history table's confidence counters. By defining the triggering instruction in terms of N (instructions) rather than directly in terms of the *prefetch distance* (cycles), we guarentee the trigger will be unique for a particular execution path, since a particular path has a fixed instruction-fetch order, but a dynamic execution pattern. A non-unique N^{th} *previous instruction* is still possible, however, if several execution paths lead to the same I-cache miss. When this happens, the different N^{th} *previous instructions* are stored independently in the miss history table. This proves to be a desirable property, as it properly handles cache lines for which there is a tendency to a miss when following one execution path, but not when following others.

The hardware for the method, shown in figure 1, consists of six components. First, a *fetch*

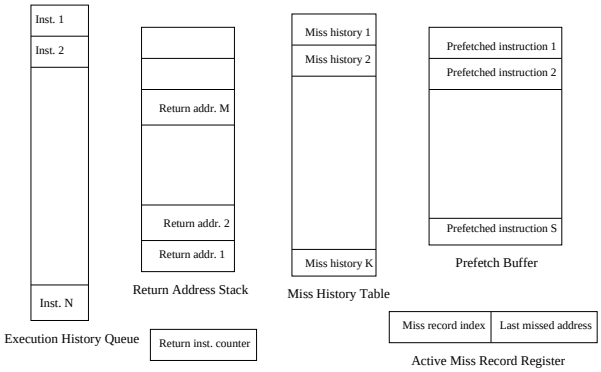

Figure 1: Hardware for our method.

Instruction address	Cache miss address	Length	Confidence counter	Valid bit

Figure 2: Format for entries in miss history table.

queue of length M stores the last M fetched instructions. M is chosen to be large enough that N instructions will be maintained in the fetch queue even when speculative instructions are evicted from this queue following a branch misprediction. Second, the *active miss record register* remembers the address and trigger of the the most recent miss. It is used to detect a $stream$, which is defined as when a sequence of I-cache misses are found to be contiguous in memory. Third, the *miss history table* stores correlations. Indexed by the low-order bits of the Nth *previous instruction*, the table's entries each have the five fields shown in figure 2: the full address of the previous instruction, the beginning address of the missing cache line (or the first address of a $stream$), the length of the stream, a confidence counter and a valid bit. Fourth, the *prefetch buffer* stores prefetched cache lines. It is used to avoid pollution of the I-cache with potentially useless prefetches. When a cache miss happens, the prefetch buffer is checked for the missing line. Only if it is absent is the lower level of the memory hierarchy accessed. The fifth and sixth items in the hardware are the return address stack and a return instruction counter; their role will be discussed later in section 2.1.

We now describe a technique for avoiding useless prefetches, *cache eviction indication*, which is derived from [18]. To understand this method, consider that a prefetch of data that is already in cache will be a useless prefetch. Most existing methods avoid this problem by directly checking the I-cache for the presence of the prefetch candidate. This method, called *cache probing*, is effective, but it uses a second I-cache port, with its consequent disadvantages. The insight we use is that there is another way to keep track of whether a line is in cache: *if the cache line has been ejected since its last fetch or prefetch, then it is not in cache, and should be*

prefetched, else it is assumed to be in cache. We term this method *cache eviction indication*. If the line is never fetched then this test returns a false in-cache result, but no correlating method, including this one, aims to avoid first-time misses anyway.

The hardware for cache eviction indication is the confidence counter of figure 2, along with associated control logic. The confidence counter is a binary counter associated with each miss history table entry, consisting of a small number of bits, usually 2-4. High values imply that the prefetch should be initiated; values below a certain threshold T imply that the prefetch should not be done as it is likely to be useless.

The *cache eviction indication* implementation requires actions at three events. *First*, when a cache line miss is encountered, a miss history table entry is created with the confidence counter set to some value R less than the threshold T. This way subsequent executions of the dynamic trigger do not prefetch, as the line is likely in cache, thus avoiding useless prefetches. Further, on a miss, if the line is present in the prefetch buffer, the miss history table entry that had caused that prefetch is found, and its confidence counter is set to be R again. Doing so avoids future useless prefetches to lines that hit in prefetch buffer. To keep track of what miss history table entry to invalidate, however, requires extra bits in the prefetch buffer: for each cache line, the index of the miss history table entry that is created when the cache line is brought into prefetch buffer is stored in the prefetch buffer entry. Analogous bits are also required in the I-cache as described below. *Second*, when a line is evicted from I-cache, the counter for the entry associated with that line is set to a high value above the threshold. This ensures that the subsequent execution after the miss will trigger a useful prefetch. To keep track of what the associated confidence counter entry is requires bits in the I-cache that store, for each cache line, the miss history table entry that was created when that line was brought into cache. These bits are analogous to the corresponding bits in the prefetch buffer. Thus upon I-cache eviction of the line, these extra bits are used to find the entry for which the confidence counter is set to maximum.

The *third* event for which cache eviction indication acts is instruction fetch. For every instruction fetch, the miss history table is accessed with the instruction address as the index. If a hit occurs, and the confidence counter for the entry is greater than the threshold, the confidence counter is decremented, and the prefetched cache lines are brought into the prefetch buffer. If the value after decrement falls to below the threshold, the miss history table entry is invalidated. This decrementing strategy ensures that entries are used only for a limited number of times – this is useful in case the original correlation was made on a different path from prefetch trigger instruction to missed line. In such a case, it is important to retire the entry after a few times

as otherwise would trigger useless prefetches forever. This is not a concern, however, if the same path from trigger to missed line is followed subsequently, as hits in the prefetch buffer in subsequent times will reset the miss history table entry as well.

The above described method of cache eviction, though complex to describe, is easy to implement. The only additional hardware is a confidence counter for each miss history table entry and extra bits to store a miss history table index for each line in prefetch buffer and I-cache, besides associated control logic. Results show that the method is as effective as cache probing in avoiding useless prefetches, without the use of a second I-cache port, and achieves a high degree of prefetch coverage.

A remaining detail is how multiple contiguous cache lines that miss in I-cache are combined into a single entry in the miss history table – this optimization reduces the size of the table required. To implement this optimization, the *active miss record register* shown in figure 1 stores the address of the previous cache miss and the index of the entry. On every cache miss, the address of the missed line is checked to see if it just follows the previous missed cache line. If it does, and the length of the previous cache miss stream is less than four cache lines, then the length of the previous cache miss stream is increased by one. Otherwise, a new miss history entry is inserted into the miss history table, and the register is set to be the current missed address. In this way, up to four contiguous misses can be combined.

2.1 Extension for return instructions

The base scheme described above may generate many harmful useless prefetches in the special case of when a procedure return instruction is executed between a correlated trigger instruction and the missing line. The reason for this special case is that a procedure can be called from several call sites, and for each, the instructions following the return are different. In this scenario, a cache miss among one return path will trigger prefetches among all, as in the base scheme, the future path is not predicted.

Fortunately, we can avoid useless prefetches spanning returns by predicting the target of the next return using a *return address stack* and prefetching only if the predicted target equals the target for the stored miss history table entry. The return address stack is not a new idea; it has been used before for predicting control flow [21]. It is maintained as a stack as follows: the return address is pushed onto it for each procedure call, and popped for each return. The stack could overflow, but the probability of overflow can be made low by increasing the size of the stack. In this way, the top of the stack always predicts the target of the next return instruction

Instruction address	Return address	Offset of missed cache	Length	Confidence counter	Valid bit	Type bit

Figure 3: Alternative format for entries in miss history table.

with a high accuracy. In our simulation, the depth of the stack is 16.

Integrating the return address stack into our prefetching scheme requires the following two modifications to the hardware. First, a signed counter is maintained for the number of return instructions in the fetch queue, minus the number of calls. The counter is incremented for every return, and decremented for a call. When the counter value is a positive number greater than zero then there are a net difference of return instructions that can be predicted. Second, an second alternative format shown in figure 3 for miss history table entries is introduced, and is used when prefetches span returns. This alternative format and the original format shown in figure 2 are distinguished by the type bit shown; the type bit is also appended to figure 2. In this alternate format, the space for the cache miss address is used for the return address encountered when the entry was created. In addition a new field stores the offset of the missing line from that return address.

The method's operation for return-associated cache misses is very similar to the case with normal cache misses. The only difference is that when an instruction is fetched, for a prefetch to be triggered, not only must an entry for the instruction be present in the miss history table, but the return address in that entry must match the current predicted return address at the top of the return address stack. When a match is found, the prefetch address is calculated as the sum of the return address and the offset in the table entry, and a prefetch initiated.

3 Related work

Instruction prefetching methods investigated have been of two types: hardware based and compiler driven. Hardware-based prefetching methods do not require any software support, while compiler driven prefetching methods rely on the compiler to specify when and what to prefetch.

Hardware-based prefetching methods have some advantages. In compiler driven approaches such as cooperative prefetching [13], explicit prefetch instructions are inserted into the executable by the compiler to prefetch the targets. Not only is compiler support needed to insert prefetch instructions, but also the prefetch instructions consume fetch and dispatch slots of the

processor at run-time. With hardware based prefetching, there is no such problem. No change is needed for the executables. This approach is compatible with legacy programs, and programs compiled elsewhere.

Hardware based cache prefetching algorithms can be divided into two types: correlated prefetching and non-correlated prefetching. We will briefly discuss some of these techniques. Correlated prefetching techniques include Markov prefetching [11] and branch-history-guided instruction prefetching(BHGP) [23]. Non-correlated prefetching techniques include fetch directed instruction prefetching(FDP) [18], next-n-line prefetching [22], and wrong-path prefetching [15].

Correlated prefetching correlates the previous cache misses with other events, such as old misses in Markov prefetching [11] or branch instructions in branch-history-guided prefetching (BHGP) [23]. Usually the correlations are stored in a dedicated table. Markov prefetching correlates consecutive miss addresses. These correlations are stored in a miss-address prediction table which is indexed using the current miss address, and which can return multiple predicted addresses. In branch-history-guided instruction prefetching, cache misses are correlated with the execution of branch instructions, and the correlations are stored in a prefetch table which is indexed using the address of the branch instructions. Later when the same events happen again, the prefetches are triggered.

Compared with the most recent related method BHGP, the major features of our method are: (1) any instruction could be a trigger, instead of limiting the triggers to be branch instructions in BHGP. In this way, the prefetched cache lines by the instructions in each basic block could be in different basic blocks in our method. (2) contiguous missed cache lines are grouped together, whereas in BHGP, the basic blocks are used as the minimum units of prefetching. Basic blocks could be big, and not all cache lines in it may be missed. In our method the number of useless prefetches are thus limited; (3) confidence counter is used in the MHT, not in the L2 cache as in BHGP. This not only limits useless prefetches, but also invalidates useless miss history records and keep the useful ones. This helps solving the contention in the MHT; (4) the triggers associated with return instructions are treated specially. This prevents the triggers in a callee function from trying to trigger a missed cache line in one caller when it is called by another.

Non-correlated prefetching does not utilize the miss history. The hardware predicts which instructions will be executed in the near future, and prefetches them – the predictions are made according to different events in different algorithms. In fetch directed instruction prefetching (FDP) [18], there is a decoupled branch predictor which runs 2 to 10 fetch blocks ahead of the instruction cache fetch. This predictor finds out the instructions that may be used in the near

future and triggers prefetches. In next-n-line prefetching [22], the access of the current cache line by the processor causes several successive cache lines to be prefetched. Another algorithm, wrong-path prefetching [15], combines next-n-line prefetching with always prefetching the taken target of control transfers that are executed by the processor.

With regards to timeliness, correlated prefetching methods generally tend to do better than non-correlated methods. Non-correlated prefetches are usually triggered when a dynamically nearby cache line is encountered, or missed in the cache. Correlated prefetching algorithms request the prefetch when the related event happens, which could be far enough to hide the cache miss latency. Although the performance study of instruction cache prefetching methods in [8] found that non-correlated prefetching methods could hide approximately 90% of the miss delay for prefetched lines, their simulated platform is a traditional supercomputer CRAY Y_MP. The measured CPI for the benchmarks is more than 2, and some of them even have a CPI of more than 4. For many of today's processors the CPI is no more than one for large classes of programs, including our simulated programs. As the miss latency increases in terms of cycles and superscalars and EPIC architectures are used, however, the timeliness of the non-correlated prefetching methods may not be good enough. As a result, correlated approaches may be more promising. In fetch directed instruction prefetching, because the branch predictor runs ahead of the instruction cache fetch, the algorithm has the timeliness property. It, however, is very aggressive by trying to prefetch every instruction that could be useful, regardless of whether it already exists in cache. Even with a perfect branch predictor, the number of prefetch requests equals the number of dynamic instructions. Typically the cache miss rate is about 5%, and 95% of the prefetches are useless. Thus this method depends on very efficient prefetching filtering algorithms.

In prefetching algorithms there is a trade off between miss coverage and memory bandwidth consumption. A more aggressive prefetching algorithm tends to prefetch more instructions, and hence may cover more cache misses. But at the same time, more useless prefetches are likely – this could consume more memory bandwidth, which will then affect other operations in the system and harm performance. There are several methods used to limit the memory bandwidth requirement. Some of them are introduced in the following paragraphs.

One method to limit useless prefetches, cache probe filtering, is used in most prefetching algorithms [23, 13, 18]. The idea is to first check whether the line that is a candidate for prefetching is already present in the primary instruction cache before the prefetch is performed. If it already there, the prefetch request is discarded. While cache probe filtering is the most widely used method, and is very efficient in preventing useless prefetches, it is expensive to build. It re-

quires an additional port for checking the address tags of the cache lines to avoid interfering with normal cache operations. Because ideal multiporting is costly, current commercial multiporting are implemented by time division multiplexing [16], or by multiple copy replication [2], or by interleaving [25]. The drawbacks of these practical implementations includes circuit complexity, area overhead, bandwidth sacrifice, and also longer access latency [19]. Taking into account these drawbacks, [24] measured a zero or negative performance gain from using prefetching methods that use cache probe filtering.

Another method to reduce the memory bandwidth consumption of prefetching is to use a prefetch bit associated with each line in the primary instruction cache and a saturating counter or a confirmation bit for each line in the next lower level instruction cache [18, 23]. The prefetch bit remembers whether the line was prefetched but not yet used, and the saturating counter records the number of consecutive times that the line was prefetched, but not used, before it was replaced. If the saturating counter for a prefetching cache line is below a threshold T, the prefething request is responded normally, or else it is dropped. In our scheme, there is a confidence counter for each entry in the miss history table. The counter works in a similar way to prevent repeated useless prefetches.

In fetch-directed instruction prefetching [18], another method is used to limit useless prefetches. The idea is that the cache lines which already exist in the i-cache should not be prefetched. In this paper, a prefetch target buffer is used to record the cache line which should be prefetched later. In each entry of the buffer there is an $evicted$ bit. This bit is set when the cache line of the corresponding prefetch target is evicted from the instruction cache. An extra field is introduced with each cache line of the instruction cache. It works as an index to the prefetch target buffer and identifies the prefetch entry that last caused the cache line to be brought into the cache. When a cache line is evicted from the cache, the index is used to access the prefetch target buffer, and the evicted bit in the entry corresponding to the index is set, indicating that the cache line will be prefetched at the next time it is used as a branch prediction. Although [18] uses a saturating counter and an evicted bit, similar to the confidence counter we use in our MHT, they do not use this information to avoid the extra I-cache port, as we do.

4 Benchmarks and simulation environment

We perform detailed cycle-by-cycle simulations of four CPU2000 benchmarks from SPEC [4] on SimpleScalar [3]. SimpleScalar is a configurable machine model that can simulate aggressive out-of-order superscalars. Table 1 shows the microarchitectural configurations used for the

Fetch & Decode Width	8
Issue Width	4
L/S Queue Size	16
Reservation Stations	64
Integer Function Units	4 add/2 mult
Branch Predictor	2-lev, 2K-entry
Memory System Ports to CPU	4
L1 I and D Cache(each)	16KB, 2-way, 32 byte
L1 Cache Access Time(cycles)	1
Unified L2 Cache	1MB, 4-way, 64 byte
L2 Cache Access Time(cycles)	12
Memory Access Time(cycles)	30

Table 1: SimpleScalar Configuration

Name	Description	Input Data Set	Instructions executed (in Millions)	IPC	I-cache Miss Rate
crafty	A high-performance Computer Chess program	The crafty.in in the reference input set	2,000	1.17	3.6%
gcc	The GNU C compiler	The integrate.c in the reference input set	2,000	0.92	3.4%
perlbmk	The intepreter of the Perl language	The scrabbl.in in the train input set	2,000	0.83	4.1%
vortex	An object-oriented database program	The train input set	2,000	0.71	6.9%

Table 2: Benchmarks

SimpleScalar simulations in our experiments. The SimpleScalar simulator models processor in detail, including number and latency of functional units, branch prediction, branch penalties, the memory hierarchy, cache miss penalties, and so on.

Four benchmarks from CPU2000 are evaluated: crafty, gcc, perl and vortex. The benchmarks selected are those where I-cache performance is especially a problem; for many floating point and some integer codes it is not. The benchmarks include Vortex, an object-oriented database program, representative of a important class of applications. The details of these applications are described in table 2.

To evaluate the performance of our EHGP algorithm, it is implemented on SimpleScalar. To compare with other prefetching algorithms, we also implement other two methods: next-n-line

	BHGP	EHGP
Prefetch buffer	2K, 4-way, 32 byte	16-entry, fully associated, 32 byte
Prefetch buffer access time (cycles)	1	1
Prefetch table	16K, 8-way, 8 Bytes	16K, 8-way, 8 Bytes

Table 3: Hardware parameters used for BHGP and EHGP.

prefetching [22] and branch history guided prefetchings (BHGP) [23]. While both of them are purely hardware based, the former is a typical non-correlated prefetching technique, and the latter is the most recent correlated method. Next-n-line prefetching has two variants: prefetches could be triggered each time a cache line is accessed, or when a cache miss happens. Our simulation results show very little difference between them. We use the latter approach. For the branch history guided prefetching, the distance between the cache miss and the associated branch instruction is set to be 5 basic blocks as used in the [23]. Both approaches use two methods to filter useless prefetchings at the same time: cache probe and the saturating bit in L2 cache. The parameters used for branch history guided prefetching and for execution history guided prefetching are listed in the Table 3. The simulation results for branch history based prefetching are different from those in the [23], because the SimpleScalar used here is for PISA (Portable Instruction Set Architecture), while they are using a ALPHA version.

5 Results

This section presents the results of our simulation experiments. First, the performance of different prefetching methods are compared. Next, several parameters of our prefetching method are varied in order to find their best values.

Figure 4 shows the normalized runtimes for our benchmarks for five competing prefetching strategies including ours. All runtimes are normalized to the runtime without prefetching = 1.0. On the x-axis are the four benchmark programs, each with five bars for the different methods. From left to right, the first bar is for a double-sized I-cache configuration, the second is for next-n-line prefetching [22], the third is for branch history guided prefetching (BHGP) [23], the fourth is for branch history guided prefetching without the cache probing operation, and the fifth is for our method, execution history guided prefetching (EHGP). As the figure shows, all prefetching methods outperform the double-sized I-cache configuration except in perl, showing that increasing the size of the I-cache is not the best solution to poor I-cache performance.

Further, we observe that the two miss history based methods, BHGP and EHGP, outperform the Next-n-line prefetching. The reason lies in the timeliness of the triggered prefetches.

The salient result from figure 4 is that our EHGP method is able to outperform BHGP without using an extra I-cache port as BHGP does. This is an important contribution since an extra port to I-cache is likely to slow down the clock cycle time [24, 19]. Consequently since our cycle-accurate simulator does not model the cycle time itself, the actual performance for BHGP is likely to be significantly worse than EHGP. The figure also shows that BHGP critically relies on cache probing – without the second port, BHGP performs much worse. Finally note that the absolute speedup is substantial at 50%. It is also reasonable as it can be approximated as follows: for perl, the base IPC is 0.83, the I-cache miss rate is 4.1%, and the miss latency is 12 cycles, therefore with 50% misses prefetched early enough, the speedup is 4.1%*50%*12/0.83, which is 29%.

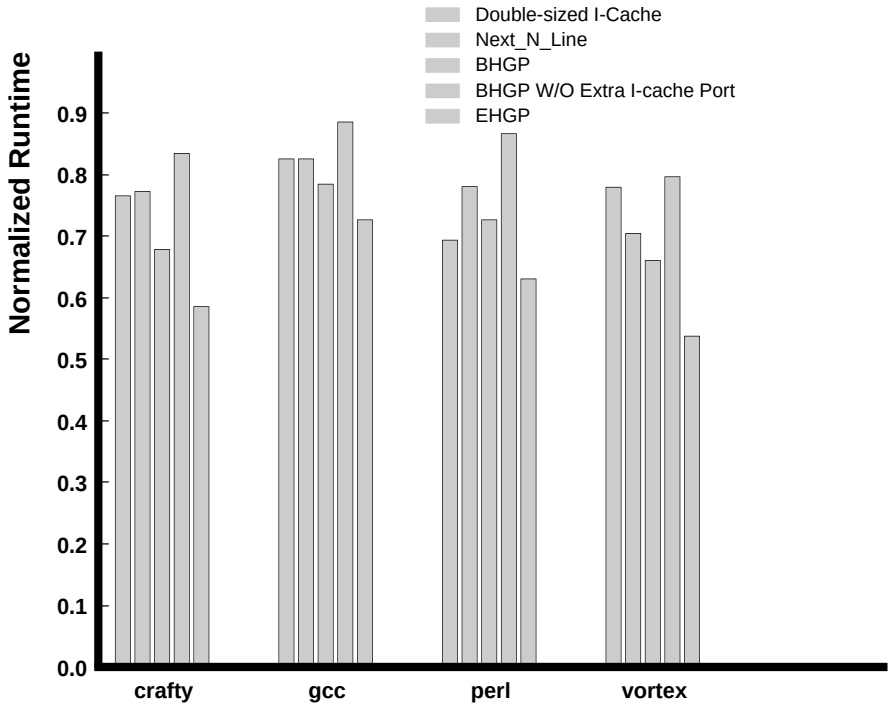

Figure 4: Normalized Runtime for Different Methods

Figure 5 shows the breakup of the prefetches into useful, late and useless prefetches as a fraction of the total I-cache misses. For each benchmark program on the x-axis there are four cases measured: next-n-line prefetching (NNL), branch history guided prefetching (BHGP), branch history guided prefetching without cache probing filtering (B(W/O)), and our method, execution history guided prefetching (EHGP). Three observations are made from the results. First, B(W/O) triggers more useless prefetches than BHGP as is expected, and also triggers fewer useful prefetches. Second, although EHGP has more useless prefetches than BHGP

stemming from its not using cache probing, it makes up the disadvantage by having more useful prefetches. Third, the actual memory bandwidth requirement for BHGP is higher than displayed in this figure. Although some of the triggered prefetch requests are ignored by the L2 cache because the saturating bit equals 0, they consume memory bandwidth anyway, and that is not measured in the figure.

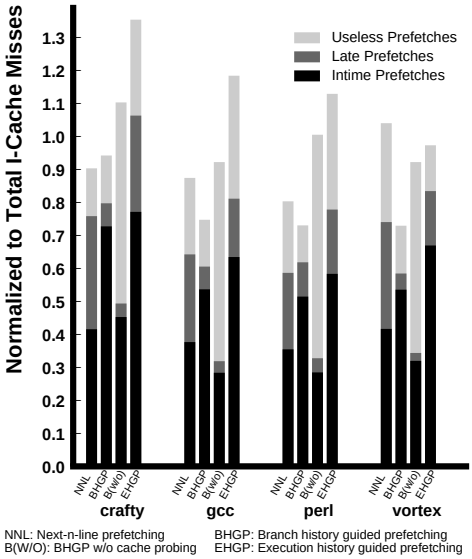

Figure 5: Breakdown of Prefetches for Different Methods.

Simulation results show that history-based prefetching methods may work well even though they do not prefetch first-time cache misses. Part of the explanation is that most cache misses are not first time misses. Our results show that only a negligible 0.003% of misses are first time misses. Non-history-based schemes can prefetch first-time misses, but they have their own problems. Next-n-line and wrong-path prefetching prefetch the cache lines too late to hide the cache miss latency. Fetch directed prefetching, though it avoids the use of an extra port, triggers too many useless prefetches. Even with a perfect branch predictor the number of prefetches may be proportional to the number of dynamic instructions.

One of the most serious problems with history based methods is that they may build up large amounts of history. Combining continuous cache misses into one history entry reduces the size of the history table required. Figure 6 shows the distribution we measured of the lengths of continuous cache misses. About half of the cache misses belong to miss streams with length larger than 4. In BHGP combining is achieved by prefetching whole basic blocks at a time instead of lines. On average basic blocks are about 6 instructions long, which is two cache lines. In contrast, custom-length combinations are generated by EHGP when needed. Moreover, it is possible that only part of the basic block caused the cache miss. Prefetching the whole block as

in BHGP may not be the best alternative.

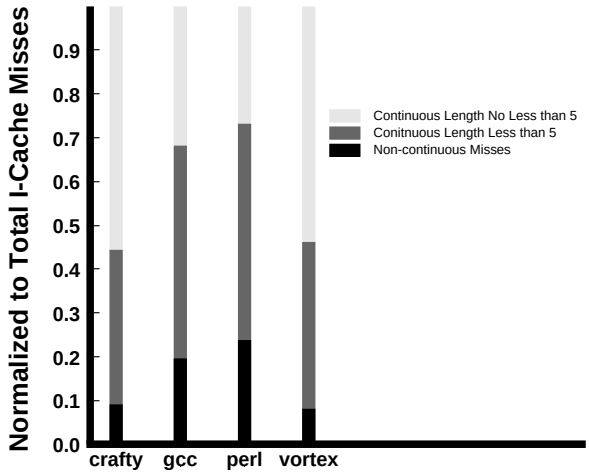

Figure 6: Distribution of Continuous Cache Misses

To evaluate the impact of combining on our results, our prefetching algorithm is evaluated with different maximum combining lengths. When the length of a cache miss stream excels the maximum value, the stream is splitted to two streams, two unrelated miss history records are maintained. The results are illustrated in figure 7. When cache misses are not combined, the performance is significantly worse than for when the maximum combination length is 4. When there is no limitation on the length, the performance is a little worse. This is because that more useless prefetches are requested.

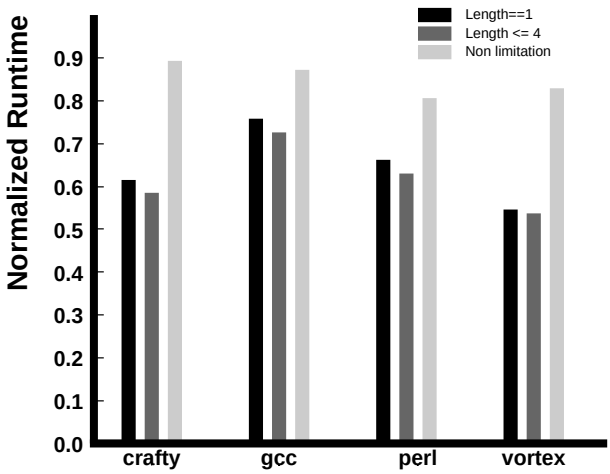

Figure 7: Impact of Length Field

Figure 8 measures the benefit of the extension for return instructions to our baseline method described in section 2.1. As expected, the cache misses are prefetched with more accuracy. As

a result, the performance with the extension is better.

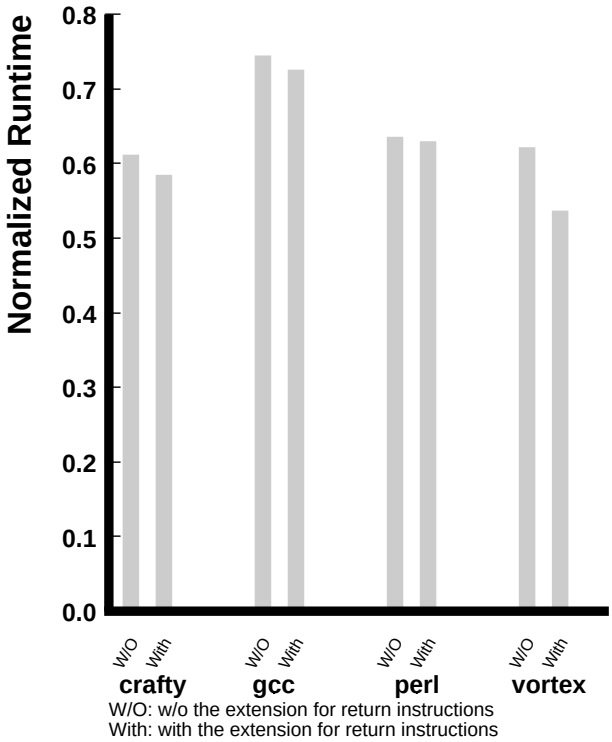

Figure 8: Impact of Return Address Stack

Figure 9 shows the measured performance of our method with different sizes for the miss history table. With a small table, less prefetches are triggered, and thus it results in less performance improvement. With a big history table, although more performance improvement is obtained, the area cost increases. There actual table size should be decided by the manufacturer based on available area and time of access.

Another interesting question is the performance advantage of our method over BHGP versus the size of MHT. We found that the average performance improvement fell to 5.7% for a 8K size MHT, down from 9.2% for 16K. In the other direction, increasing the table size to 32K kept the performance improvement relatively unchanged from the 16K case. As explained in [23], in BHGP 8K is enough for MTH, and increasing it to 16K does not improve the performance much. In our method, 16K is much better. As technology develops, this may be a desirable property. (Detailed figures not showed.)

Figure 10 shows the impact of varying N, defined above as the number of instructions between the trigger and the I-cache miss. When N is small, the accuracy of the prefetch algorithm is good, but the prefetches are triggered too late to fully hide the cache miss latency. With a larger N, the accuracy may be worse and prefetched lines may be evicted before being used.

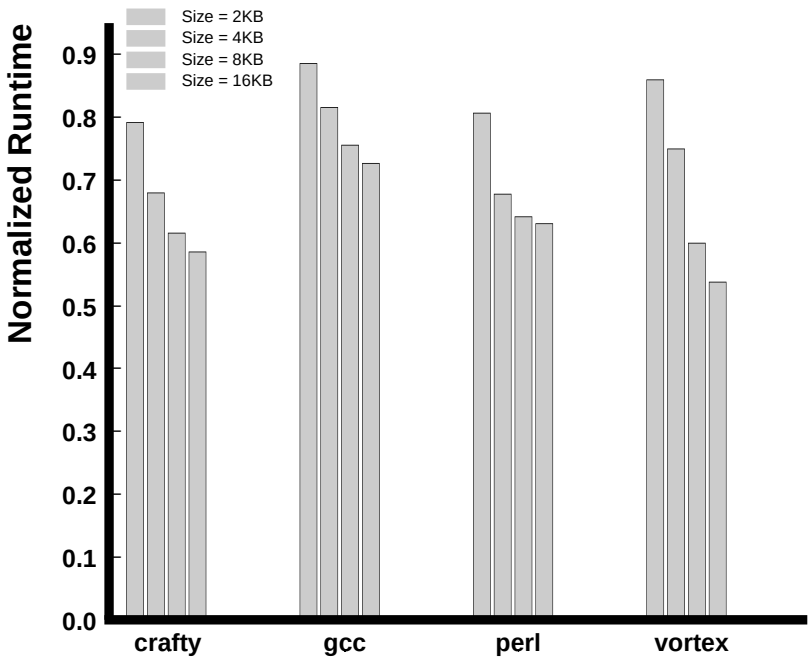

Figure 9: Impact of Miss History Table Size

The key is to choose the optimal value of N for the given architecture. For our platform, $N = 16$ gives the best performance.

6 Summary

This paper describes a hardware-based execution history guided method for instruction prefetching. It improves upon existing methods by triggering prefetches early enough, avoiding the extra port in I-cache, and yet prefetching a high percentage of cache misses with a table of reasonable size.

To achieve these gains, our method makes three innovations. First, cache misses are correlated with dynamically preceding instructions at a certain fixed dynamic distance. Prefetches are subsequently triggered by those instructions before the cache misses happen. Second, contiguous cache misses are grouped together and associated with one preceding instruction, allowing a miss history table of reasonable size. Third, efficient prefetch filtering methods are used and thus an extra I-cache port is unnecessary.

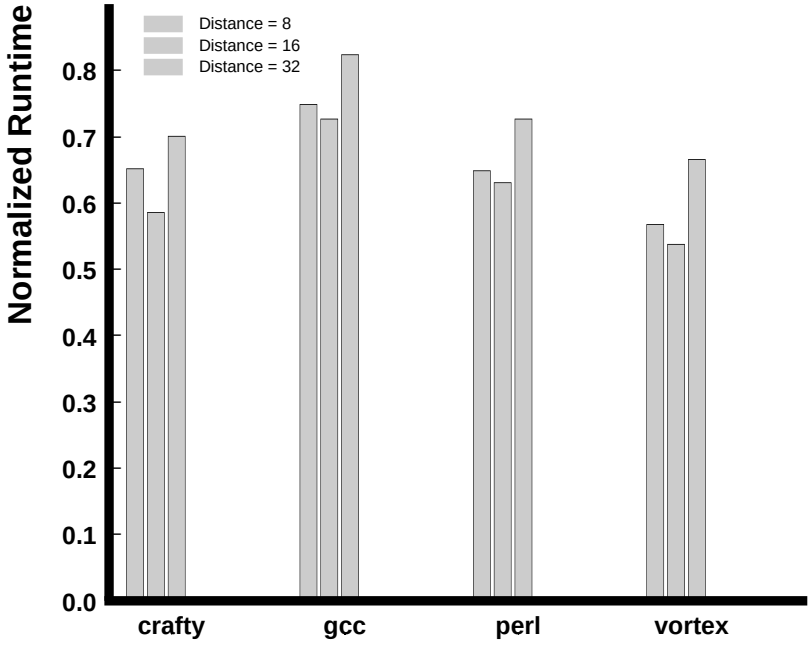

Figure 10: Impact of Trigger Distance, N

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *The VLDB Journal*, pages 266–277, 1999.

[2] *Alpha Architecture Handbook.* Digital Equipment Corporation, Maynard, MA, 1994.

[3] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report TR 1342, University of Wisconsin, Madison, WI, June 1997.

[4] S. P. E. Corporation. The SPEC benchmark suites. *http://www.spec.org/*.

[5] A. M. Grizzaffi, M. Colette, M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–155, October 1994.

[6] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, second edition, 1996.

[7] D. S. Henry, B. C. Kuszmaul, G. H. Loh, and R. Sami. Circuits for wide-window superscalar processors. In *Proc. of the 27th Int'l Symp. on Computer Architecture (ISCA)*, pages 236–247, Vancouver, British Columbia, Canada, June 2000.

[8] W.-C. Hsu and J. E. Smith. A Performance Study of Instruction Cache Prefetching Methods. *IEEE Transactions on Computers*, 47(5):497–508, May 1998.

[9] *Intel IA-64 Architecture Software Developer's Manual, Volumes I-IV*. Intel Corporation, January 2000. Also available at http://developer.intel.com.

[10] *Intel(R) Itanium(TM) Processor Hardware Developer's Manual*. Intel Corporation, August 2001.

[11] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

[12] G. Lauterbach and T. Horel. UltraSPARC-III: designing third generation 64-bit performance. *IEEE Micro*, 19(3):56–66, 1999.

[13] C.-K. Luk and T. C. Mowry. Cooperative instruction prefetching in modern processors. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 182–194, November 30-December 2 1998.

[14] Y. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark. One Billion Transistors, One Uniprocessor, One Chip. *IEEE Computer*, 30(9):51–58, Sept. 1997.

[15] J. Pierce and T. N. Mudge. Wrong-path instruction prefetching. In *International Symposium on Microarchitecture*, pages 165–175, 1996.

[16] *IBM Regains Performance Lead with Power2*. Microprocessor Report, October 1993.

[17] *PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual*. IBM Corporation, 1999.

[18] G. Reinman, B. Calder, and T. Austin. Fetch Directed Instructin Prefetching. In *Proceedings of the 32nd Annual ACM/IEEE international symposium on microarchitecture on MICRO-32*, pages 16–27, Haifa Israel, November 1999.

[19] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin. On high-bandwidth data cache design for multi-issue processors. In *Proceedings of the thirtieth annual IEEE/ACM international symposium on Microarchitecture*, pages 46–56, December 1-3 1997.

[20] International Technology Roadmap for Semiconductors, 1998 Update. *Semiconductor Industry Association*, page 4, 1998.

[21] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *International Symposium on Microarchitecture*, pages 259–271, 1998.

[22] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[23] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak. Branch History Guided Instruction Prefetching. In *Proceedings of the 7th Int'l Conference on High Performance Computer Architecture (HPCA)*, pages 291–300, Monterrey, Mexico, January 2001.

[24] J. Tse and A. J. Smith. CPU Cache Prefetching: Timing Evaluation of Hardware Implementations. *IEEE Transactions on Computers*, 47(5):509–526, May 1998.

[25] K. Yeager and et. al. Superscalar Microprocessor. *Hot Chips VII*, 1995.