Execution History Guided Instruction Prefetching

Yi Zhang

Actuate Corporation

South San Francisco, CA 94080, U.S.A

Steve Haga

Department of Electrical & Computer Engineering

University of Maryland

College Park, MD 20742, U.S.A

Rajeev Barua

Department of Electrical & Computer Engineering

University of Maryland

College Park, MD 20742, U.S.A

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 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, 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 is 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. When compared to the original machine without prefetching, our method improves performance by about 35% for our benchmarks.

Keywords: Instruction prefetching, instruction cache, memory latency, hardware pre-fetching.

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* of that cache line is requested. If the prediction is accurate and timely, the cache line will 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 *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 of the cache miss latency. Second, some methods [11] use a huge table to store the miss history information. Finally and more importantly, most prefetching methods, including the most recent [23, 13, 18], use *cache probing* to filter out useless prefetches. Cache probing requires that the I-cache be checked before every potential prefetch, in order to prevent the prefetching of lines that are already in the 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 therefore not surprising that current methods for hardware-based 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 predicted branch targets; no sophisticated hardware-based schemes are used. Instead, these machines provide special prefetch instructions that are scheduled by the compiler based on program analysis, perhaps employing profiling. However, compiler-driven methods have their own shortcomings: first, prefetch instructions themselves constitute overhead; second, not all compilers may implement prefetching; third, old binaries are not optimized; fourth, dynamic program behavior may not be predictable at compile-time; and fifth, efficient portability across different implementations of the ISA is difficult because the prefetching strategy is affected the hardware parameters, such as cache size and latency. 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 prevent stalls, by avoiding an 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 of instructions 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. To achieve this, 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. Database and irregular programs typically demonstrate such behavior. Several existing strategies are also implemented for comparison. Our method correctly prefetches an average of 71.3% of I-cache misses, and 77.6% of these are prefetched early enough to completely hide the cache miss latency. As a result, the speedup for the benchmark applications is about 1.5.

The rest of this paper is organized as follows. Section 2 presents further details on the execution history guided pre-fetching technique. Section 3 discusses the related works on instruction prefetching. Section 4 describes the details of the simulation environment and the benchmarks used for this study. Section 5 presents our experimental methodology and results. Section 6 summarizes our conclusions.

2 Execution history guided instruction prefetching

We now describe our basic method. Let the *prefetch distance* be a constant that is slightly greater than the miss latency from L1 to L2 in the I-cache (typically 10-30 cycles for modern processors). Let N be the average number of instructions that execute within this *prefetch distance*. Whenever there is an I-cache miss, the address of the instruction that was fetched N instructions prior to the miss, called the N^{th} *previous instruction* is stored into a miss history table (MHT). Upon encountering this N^{th} *previous instruction* in the future, the hardware will prefetch unless the line is still in the cache, as indicated by the MHT entry's confidence counter. By defining the prefetch-triggering instruction in terms of N (instructions) rather than directly in terms of the *prefetch distance* (cycles), we guarantee that 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 MHT. This proves to be a desirable property, as it properly handles cache lines for which there is a tendency to a miss when following some execution paths, but not when following others.

The entries of the MHT contain the five fields shown in Figure 1: 1) the address of the N^{th} *previous instruction* (only the upper bits are needed since the MHT is indexed by the low-order bits), 2) the beginning address of the cache line to prefetch, 3) a confidence counter, 4) a valid bit, 5) and the length of the *stream*. A *stream* is as a sequence of I-cache misses that are contiguous in memory. Streams allow multiple prefetches from a single MHT entry.

The hardware for the method, shown in Figure 2, consists of six components. First is the *miss history table* as already described. Second, a *fetch queue* of length M stores the most recently fetched instructions. The fetch queue is used to identify the N^{th} *previous instruction*. M should therefore be large enough to guarentee that N instructions are always in the queue, even when speculative instructions are evicted from it due to a branch misprediction. Third, the *active miss record register* stores the address of the most recent miss and a pointer to the corresponding MHT entry it created. This register is used for detecting streams, as described later in this section. Fourth, the *prefetch buffer* stores the prefetched cache lines separately from the main cache. This buffer prevents the I-cache from evicting useful lines for the sake of potentially useless prefetches. When a cache miss happens, the prefetch buffer is first checked for the missing line, before accessing the lower level of memory. The fifth and sixth items in the hardware are the return address stack and a return instruction counter; their roles will be discussed later in Section 2.1.

We now describe *cache eviction indication*, a technique for reducing useless prefetches, derived from [18]. Since prefetching data that is already in the cache would simply waste memory bandwidth, most existing methods directly check the I-cache for the presence of the prefetch candidate.

This solution, called *cache probing,* requires a second I-cache port, with its consequent disadvantages. *Cache eviction indication* uses a different method: *if the cache line has been evicted since its last fetch or prefetch, then it is not in the cache, and should be prefetched,* otherwise it can be assumed to be in the cache. If the cache line has never been fetched before, then this test will yield an incorrect result. This presents no difficulty, however, as no correlating method, including ours, aims to avoid first-time misses.

The hardware for implementing *cache eviction indication* is a confidence counter as shown in Figure 1, along with the associated control logic. A confidence counter exists for each miss history table entry; it consists of a small number of bits, usually 2-4. Values above a certain threshold T indicate that the prefetch should be initiated; low values imply that the prefetch should not be requested since it is likely to be useless.

An implementation of *cache eviction indication* requires actions at three events. *First,* when a cache line miss is encountered, the miss history table is updated. One update action is to created a new entry with the confidence counter set to some value R no more than the threshold T. This prevents subsequent executions of the dynamic trigger from prefetching, as the line is likely to be in the cache. In our implementation, we set both T and R to 0. The other update action that occurs on a cache line miss is that, 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 the 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.

The *second* event that requires action under the *cache eviction indication* scheme is when a line is evicted from the I-cache. The counter field of the corresponding miss history table entry is set to a high value that is above the threshold. This ensures that a subsequent execution of the triggering instruction will cause a useful prefetch. Keeping track of the associated confidence counter entry requires extra bits in every I-cache line, to store the miss history table entry that was created when that line was brought into the cache. These bits are analogous to the corresponding bits in the prefetch buffer. Thus, upon evicting a line from the I-cache, these extra bits are used to find the entry for which the confidence counter is set to maximum.

The *third* event in the *cache eviction indication* scheme is instruction fetch. For every instruction fetch, the miss history table is accessed with the instruction address as the index. If a matching entry is found and the confidence counter for that entry is greater than the threshold, then the confidence counter is decremented, and the prefetched cache line is brought into the prefetch buffer. If the value after decrement falls below the threshold, the miss history table entry is invalidated. This decrementing strategy ensures that entries are used for only a limited number of times – which is useful in case the original correlation between the prefetch trigger instruction and the missed line was for a different execution path. In such a case, it is important to retire the entry after a few times as it would otherwise trigger useless prefetches forever. This is not a concern, however, if the same path from the trigger to the missed line is followed subsequently; later hits in the prefetch buffer will reset the miss history table entry as well.

The above algorithm for cache eviction, though complex to describe, is simple 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 the prefetch buffer and I-cache, plus the associated control logic. Results show that the method is almost 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 the 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 2 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 follows the previous missed cache line. If it does, and if the length of the previous cache miss stream is less than four cache lines, then this length 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 into one entry.

2.1 Extension for return instructions

The base scheme described above may generate many useless prefetches for the special case of when a return-from-procedure operation occurs between a prefetch triggering instruction and the I-cache miss it is correlated to. The reason for this special case is that a procedure can be called from several call sites, and the instructions following the return are different for each of them. In this scenario, a cache miss among one return path will trigger prefetches among all.

Fortunately, we can avoid useless prefetches spanning returns by predicting the target of the next return using a *return address stack* and then prefetching only if the predicted target equals the target of 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]. The stack is maintained as follows: the return address is pushed onto it for each procedure call, and popped for each return. The stack can overflow, but the probability of overflow is made low by increasing the size of the stack. Thus, the top of the stack predicts the target of the next return. In our simulation, the depth of the stack is 16.

Having extended the algorithm to handle prefetches that span returns, we now describe how to identify this situation, so as to use the extension. A given prefetch spans a return if and only if the sum of all returns in the fetch queue is greater than the sum of calls in the fetch queue *that occur prior to the last return*. For example, if the fetch queue contains a single call followed later by a single return then we know that the N^{th} previous instruction is from the same procedure as the current instruction, and we would not use the extension. We similarly do not use this extension if the number of calls is greater than the number of returns, as this represents a case where the N^{th} previous instruction is from the caller (or an ancestor of the caller); prefetching into a callee does not present the same problems as prefetching into the caller. This also explains why call instructions that occur after the last return in the fetch queue are not counted.

Integrating this extension into our prefetching scheme requires the following three hardware modifications. First, the entries of the fetch queue require both a new field for the return address on the top of the stack at the time the instruction was fetched and also two new bits to indicate whether the instruction is a call or a return. Second, to indicate when to use the extension, two counters are introduced. One is signed and maintains the number of returns in the fetch queue minus the number of calls that occur prior to the last return. The other records the number of calls after the last return instruction in the queue. Third, an alternative format for miss history table entries is introduced, as shown in Figure 3. This format is used when prefetches span returns. Comparing Figures 1 and 3 reveals that two new fields have been added to this alternative format, and the *cache miss address* field has been replaced with the *return address*.

When the extension is used, it operates very similarly to the way normal cache misses are handled. The only difference is that, for an instruction to trigger a prefetch, not only must there be a valid MHT entry for that instruction, but the return address for that entry must also match the current predicted return address, as found by examining the top of the return address stack. When a prefetch is initiated, the prefetch address is calculated as the sum of the return address and the offset field from its table entry.

3 Related work

The instruction prefetching methods that have been previously investigated in the literature are of two types: hardware based and compiler driven. Hardware-based prefetching methods do not require any software support, where as compiler-driven prefetching methods rely on the compiler to specify when and what to prefetch.

In compiler-driven approaches, such as cooperative pre-fetching [13], explicit prefetch instructions with predicted targets are inserted into the executable code by the compiler. Thus, no benefit can be obtained for legacy programs, or for programs compiled elsewhere. 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 are no such problems.

Hardware-based cache prefetching algorithms can be divided into two categories: correlated prefetching and non-correlated prefetching. Correlated prefetching techniques include Markov prefetching [11] and branch-history-guided instruction prefetching(BHGP) [23]. Non-correlated prefetching techniques include fetch-directed instruction prefetching(FDIP) [18], next-n-line prefetching [22], and wrong-path prefetching [15].

Correlated prefetching associates each cache miss with some previous event, such as a prior miss in Markov prefetching [11] or a branch instruction in branch-history-guided instruction 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, our method has four main features. First, any instruction can be a trigger, where as BHGP only uses branch instructions as triggers. When only branches are triggers, each basic block can only have one entry in the miss history table. Our finer granularity allows more prefetching opportunities. Second, contiguous missed cache lines are grouped together, whereas in BHGP, the basic blocks are used as the minimum units of prefetching. For a large basic block spanning multiple cache lines, it is possible that only some of these lines will cause misses. Thus, our method reduces the number of useless prefetches. Third, a 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 while keeping the useful ones, thus reducing contention in the MHT. Fourth, the triggers associated with return instructions are treated specially. This prevents the triggers in a callee function from prefetching a cache line that corresponds to a different caller.

We now consider the non-correlated prefetching methods proposed in the literature. Instead of using a miss history table, these methods predict which instructions will be executed in the near future, in order to prefetch them. These predictions are made according to different events in the different algorithms. In fetch directed instruction prefetching (FDIP) [18], there is a decoupled branch predictor which runs 2 to 10 fetch blocks ahead of the instruction cache fetch. This predictor identifies 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. Wrong-path prefetching [15], combines next-n-line prefetching with a mechanism to always prefetch the taken target of control transfers that are executed by the processor.

In regards to timeliness, correlated prefetching methods generally outperform non-correlated methods. A prefetch is timely if it occurs early enough to fully hide the memory latency. Correlated algorithms are more timely because they relate a cache miss to a prior event, which can be chosen so as to occur early enough to hide the cache miss latency. In contrast, non-correlated prefetches are usually triggered when a dynamically nearby cache line is encountered or misses in the cache. Thus the prefetches are often late. An exception is fetch-directed instruction prefetching, FDIP, which is a non-correlated method with the timeliness property. It is timely because the branch predictor runs ahead of the instruction cache fetch. FDIP is very aggressive, however, by trying to prefetch every instruction that could be useful, regardless of whether it already exists in the cache. Even with a perfect branch predictor, the number of prefetch requests equals the number of dynamic instructions divided by the cache block size. Typically the cache miss rate is about 5%, so 95% of these prefetches are useless. Thus FDIP depends on very efficient prefetch filtering algorithms.

The statement that correlated prefetching methods are more timely than non-correlated methods may appear to contradict [8], where it was found that non-correlated prefetching methods could hide approximately 90% of the miss delay for prefetched lines. The explanation is that the simulated platform in [8] is a traditional supercomputer CRAY Y_MP. The measured IPC for their benchmarks is less than $\frac{1}{2}$, and sometimes less than $\frac{1}{4}$. On most of today's processors, however, the IPC is above 1 for many programs – including those programs studied in our results. Moreover, as the gap between memory access time and computer cycle time grows, the timeliness of the non-correlated prefetching methods will deteriorate further. As a result, correlated approaches appear more promising.

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 can 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 consumed by useless prefetches. Some of these methods 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 is already there, the prefetch request is discarded. While cache probe filtering is the most widely proposed method, and prevents all useless prefetches, it is expensive to build, because it requires 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 multiport caches are implemented by time division multiplexing [16], by multiple copy replication [2], or by interleaving [25]. The drawbacks of these practical implementations include circuit complexity, area overhead, bandwidth sacrifice, and also longer access latency [19]. Taking into account these drawbacks, [24] measures a zero or negative performance gain from using prefetching methods that require cache probing.

Another method to reduce the memory bandwidth consumption from prefetching is to use a prefetch bit associated with each line in the primary instruction cache plus a saturating counter or a confirmation bit for each line in the next lower level of the 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 above a threshold T, the prefetch request is ignored. In our scheme, there is a confidence counter for each entry in the miss history table. The counter works in a similar way to [18, 23], preventing repeated useless prefetches.

Fetch-directed instruction prefetching [18], employs a different method for limiting useless prefetches. A prefetch target buffer records the cache line to be prefetched. 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 for each cache line of the instruction cache. This field functions 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 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. The SimpleScalar simulator models a processor in detail, including the number and latency of the functional units, branch prediction, branch penalties, the memory hierarchy, cache miss penalties, and so forth. Table 1 shows the microarchitectural configurations used for the SimpleScalar simulations in our experiments. These parameter values are reasonable for modern processors, and were chosen for ease of comparison with [23].

To compare the performance of our EHGP algorithm, with existing prefetch algorithms, we have also implemented two older methods in SimpleScalar: next-n-line prefetching [22] and branch history guided prefetching (BHGP) [23]. While both of these 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 can be triggered each time a cache line is accessed, or when a cache miss happens. Since our simulation results show very little difference between these variants, 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 in [23]. Both next-n-line and BHGP use the same two methods to filter useless prefetches: cache probing and a saturating bit in L2 cache. The parameters used for branch history guided prefetching and for execution history guided prefetching are listed in Table 2. The simulation results for branch history based prefetching are different from those in [23], because SimpleScalar runs PISA (Portable Instruction Set Architecture) code, while [23] uses a version of ALPHA. It should be noted, however, that for each benchmark, the number of branches is almost the same when compiled for PISA as for ALPHA.

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 an important class of applications. The details of these applications are described in Table 3. As in [23] the first 2 billion instructions are used.

5 Results

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

Figure 4 shows the normalized runtimes of our benchmarks for five competing prefetching strategies, including our own. All runtimes are normalized so that the runtime without prefetching equals 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 for on perl. Increasing the size of the I-cache is therefore not the best solution to poor I-cache performance. Further, we observe that the two miss history based methods, BHGP and EHGP, outperform 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 relies heavily on cache probing. Finally note that the absolute speedup 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% of misses prefetched early enough, the speedup is 4.1%*50%*12/0.83, which is 29%.

Figure 5 shows the breakup of 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 pre-fetching (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 Figure 5. First, more useless prefetches are triggered with B(W/O) than with BHGP as is expected, and also fewer useful prefetches are triggered. Second, although EHGP makes more useless pre-fetches than BHGP, stemming from its not using cache probing, it compensates for this by having more useful prefetches. Third, the actual memory bandwidth requirements for BHGP and for next-n-line are higher than displayed in Figure 5. Although some prefetch requests are ignored because the saturating bit from the L2 cache equals 0, yet these requests still consume memory bandwidth while accessing the L2 cache. This additional cost is not measured in the figure. Our method, EHGP, does not have this difficulty.

The simulation results show that history-based prefetching methods may work well even though they do not prefetch first-time cache misses. This shows that most cache misses are not first time misses. In fact, 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 will prefetch the cache lines too late to fully 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 is 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 that is required. Figure 6 shows the distribution of cache misses, as broken down by the number of continuous cache misses of which this miss is a part. About half of the cache misses belong to miss streams with lengths larger than 4. In BHGP, the issue is addressed by prefetching whole basic blocks at a time instead of individual lines. Basic blocks average about 6 instructions, which is 2 cache lines. In contrast, custom-length combinations are generated by EHGP when needed. Moreover, it is possible that only part of a basic block misses in the cache. Therefore, prefetching the entire block as with BHGP may not be the best approach.

To evaluate the impact of combining cache lines, our pre-fetching algorithm is evaluated with different maximum combination lengths. When the length of a cache miss stream exceeds the maximum value, the stream is split into two streams, and two unrelated MHT entries are maintained. Figure 7 shows the result. When cache misses are not combined, the performance is significantly worse than when the maximum combination length is 4. Also, when there is no limitation on the length, the performance degrades. This is because more useless prefetches are requested.

Figure 8 measures the benefit of extending our baseline method to handle return instructions, as 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 9 shows the measured performance of our method with different sizes for the miss history table. With a smaller table, fewer prefetches are triggered, thus resulting in less performance improvement. Since a larger MHT will consume more chip area, the actual table size should be decided by the manufacturer based on the available area and the time of access.

Another interesting question is the performance advantage of our method over BHGP versus the size of the 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], BHGP only requires 8K for the MHT, and increasing it to 16K does not significantly improve performance. In our method, 16K is a better choice. As technology develops, this may be a desirable property. (Detailed figures not shown.)

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. 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, by avoiding the extra port for the 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.

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.

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

	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 byte	16K, 8-way 8 byte

Table 2: Hardware Parameters for BHGP & EHGP

Name	Description	Input Data Set	IPC	I-cache Miss Rate
crafty	computer chess program	crafty.in reference input set	1.17	3.6%
gcc	GNU C compiler	integrate.c reference input set	0.92	3.4%
perlbmk	interpreter of the Perl language	scrabbl.in train input set	0.83	4.1%
vortex	object-oriented database	vortex.in train input set	0.71	6.9%

Table 3: Benchmarks. All benchmarks were tested for the first 2 billion instructions executed.

Instruction address	Cache miss address	Length	Confidence counter	Valid bit

Figure 1:

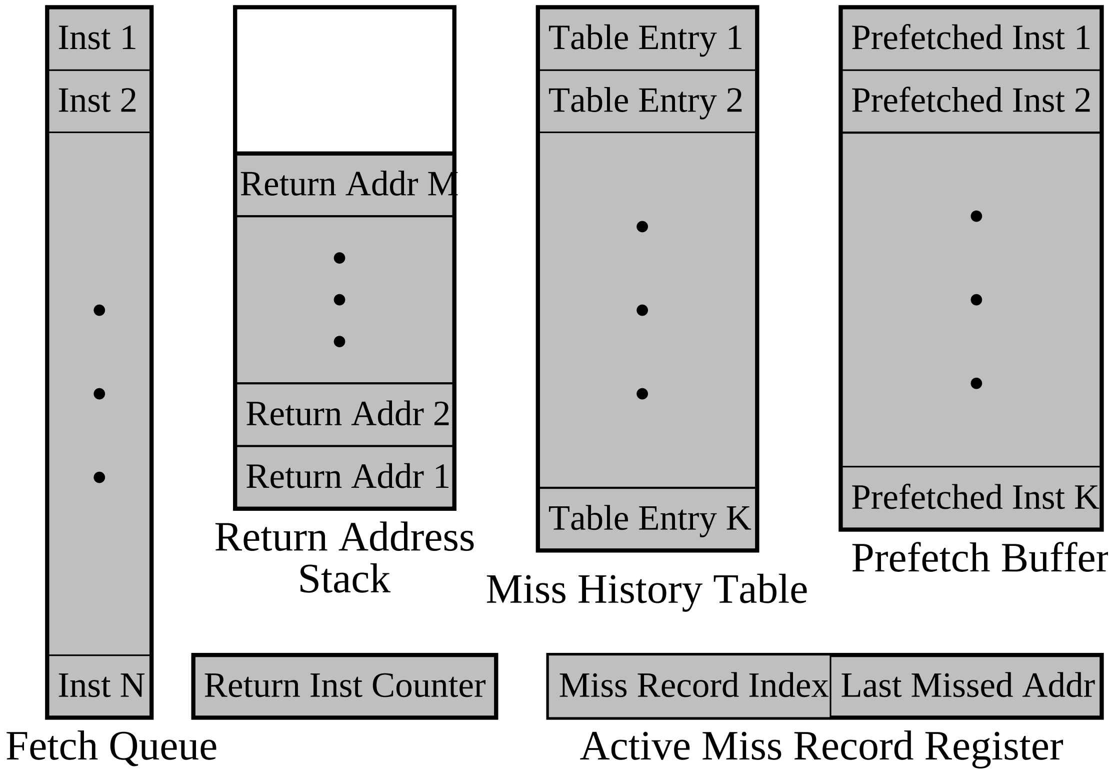
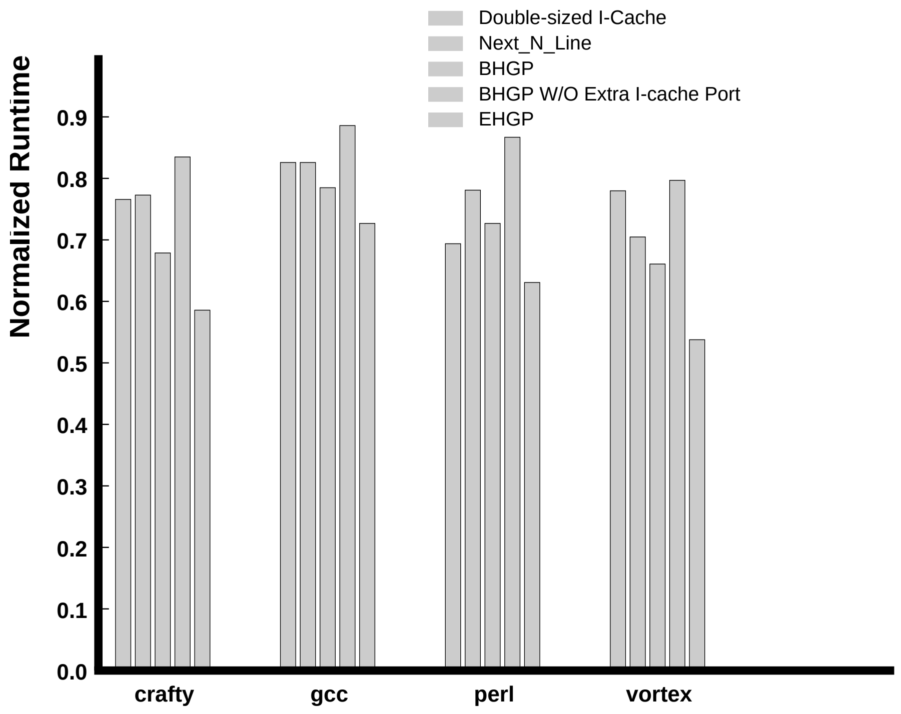

Inst 1
Inst 2
Inst N
Fetch Queue
Return Addr M
Return Addr 2
Return Addr 1
Return Address Stack
Return Inst Counter
Table Entry 1
Table Entry 2
Table Entry K
Miss History Table
Prefetched Inst 1
Prefetched Inst 2
Prefetched Inst K
Prefetch Buffer
Miss Record Index
Last Missed Addr
Active Miss Record Register

Figure 2:

Instruction address	Return address	Offset to cache miss	Length	Confidence counter	Valid bit	Type bit

Figure 3:



Double-sized I-Cache
Next_N_Line
BHGP
BHGP W/O Extra I-cache Port
EHGP
Normalized Runtime
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
crafty
gcc
perl
vortex

Figure 4:

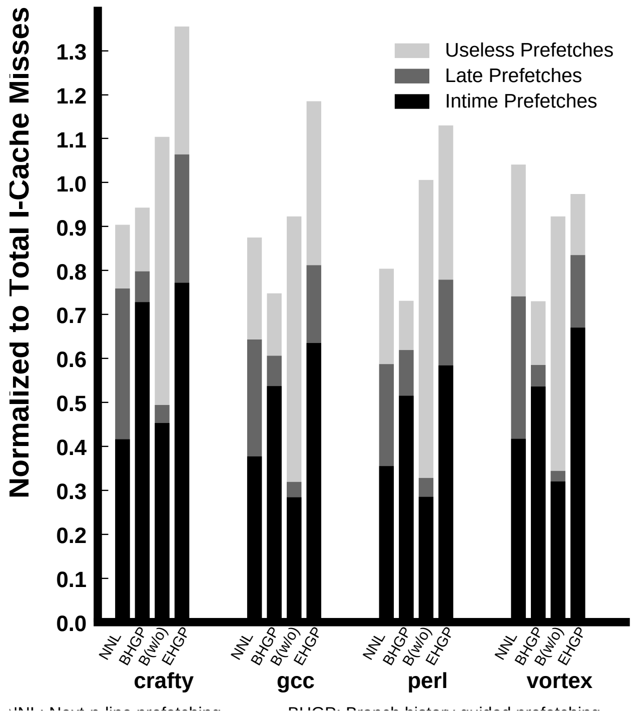

Normalized to Total I-Cache Misses
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Useless Prefetches
Late Prefetches
Intime Prefetches
NNL
BHGP
B(w/o)
EHGP
crafty
gcc
perl
vortex

Figure 5:

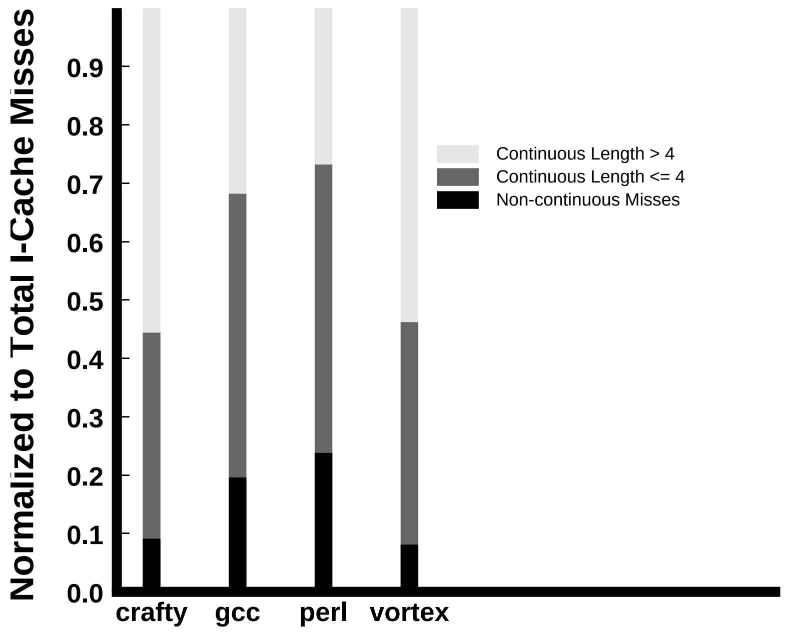

Normalized to Total I-Cache Misses
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
crafty
gcc
perl
vortex
Continuous Length > 4
Continuous Length <= 4
Non-continuous Misses

Figure 6:

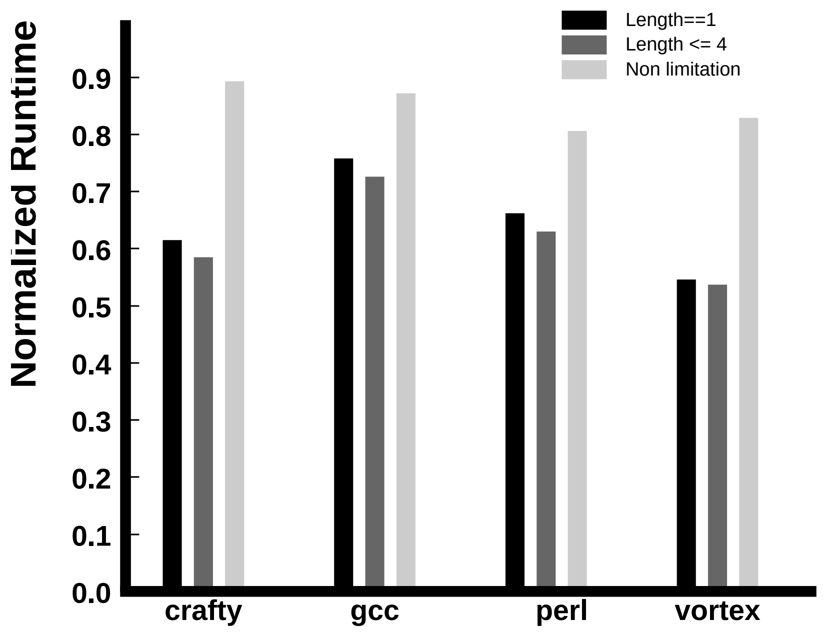

Length==1
Length <= 4
Non limitation
Normalized Runtime
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
crafty
gcc
perl
vortex

Figure 7:

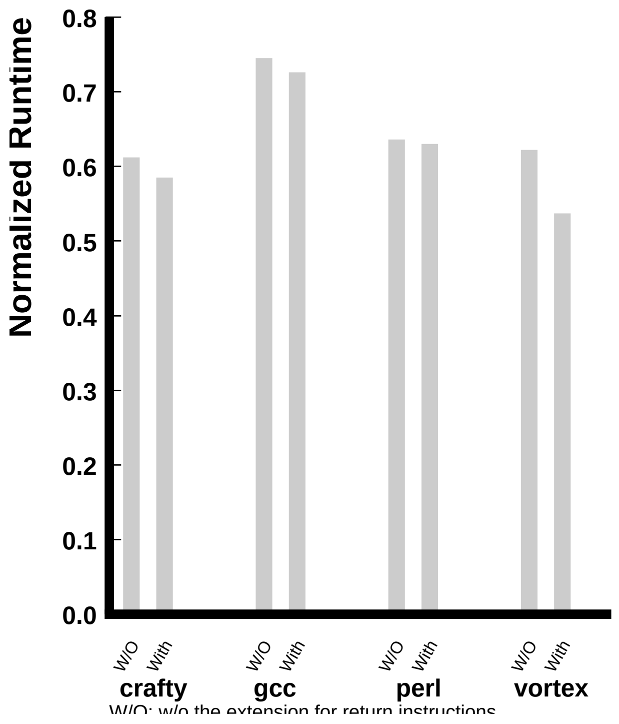
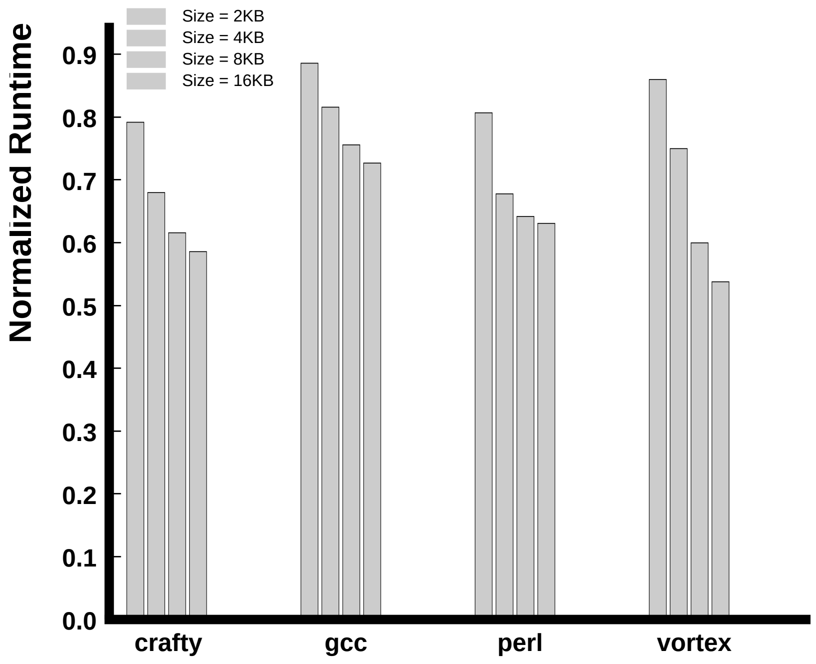

Normalized Runtime
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
W/O
With
crafty
gcc
perl
vortex
W/O: w/o the extension for return instructions

Figure 8:



Size = 2KB
Size = 4KB
Size = 8KB
Size = 16KB
Normalized Runtime
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
crafty
gcc
perl
vortex

Figure 9:

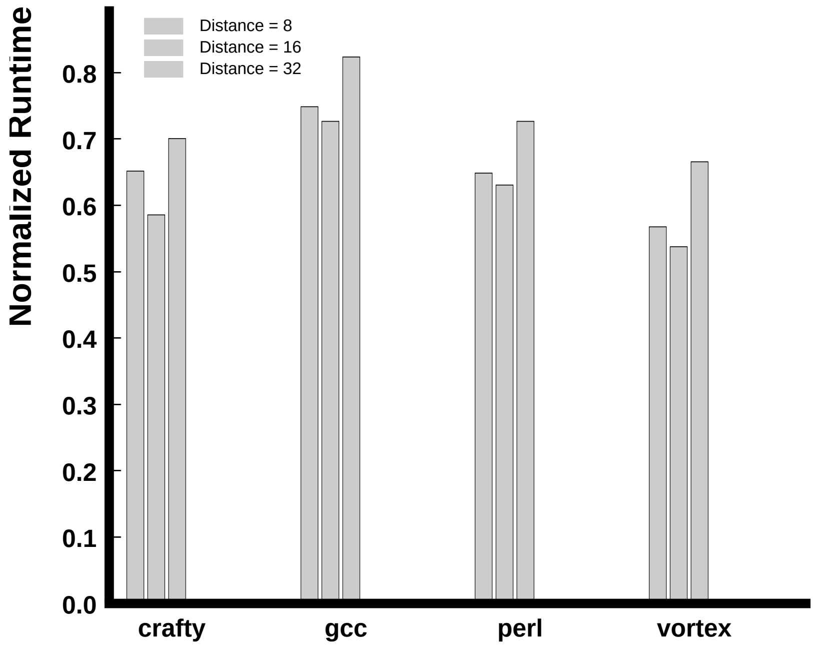

Normalized Runtime
Distance = 8
Distance = 16
Distance = 32
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
crafty
gcc
perl
vortex

Figure 10: