Effective Stream-Based and Execution-Based Data Prefetching

Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou & Santosh G. Abraham
Scalable Systems Group
Sun Microsystems Inc., Sunnyvale, CA
{sorin.iacobovici,lawrence.spracklen,sudarshan.kadambi,yuan.chou,santosh.abraham}@sun.com

ABSTRACT

With processor speeds continuing to outpace the memory subsystem, cache missing memory operations continue to become increasingly important to application performance. In response to this continuing trend, most modern processors now support hardware (HW) prefetchers, which act to reduce the missing loads observed by an application.

This paper analyzes the behavior of cache-missing loads in SPEC CPU2000 and highlights the inability of unit and single non-unit stride prefetchers to correctly prefetch for some commonly occurring streams. In response to this analysis, a novel multi-stride prefetcher, that supports streams with up to four distinct strides, is proposed. Performance analysis for SPEC CPU2000 illustrates that the proposed multi-stride prefetcher can outperform current stride prefetchers on several benchmarks; most notably on `mcf`, `lucas` and `facerec`, where it achieves an additional performance gain of up to 57%. Performance of the strided HW prefetchers is also contrasted with another recently proposed prefetch scheme, runahead execution (RAE), and the synergy between the schemes is investigated.

Categories and Subject Descriptors

B.8 [**Performance and Reliability**]: Miscellaneous

General Terms

Performance

Keywords

hardware prefetcher, stream prefetching, multiple strides

1. INTRODUCTION

With the rapid improvements in processor clock speed, a main memory access resulting from a cache miss may take anywhere from 100 to almost 1000 processor clock cycles. Memory prefetching schemes attempt to initiate the memory access for a cache line, before the processor issues a load to that cache line. If the prefetch is initiated sufficiently in advance of the load, the entire latency of the memory access is covered and the load does not result in any stall cycles. Even if the prefetch does not cover the entire memory access latency, the prefetch is still beneficial because it reduces the cycles that the processor is stalled waiting for the load to complete. In addition to overlapping computation cycles with memory access cycles, prefetching may also overlap multiple memory accesses when there are multiple prefetches in close temporal proximity.

General-purpose processors have instruction set support for software prefetching. The processor moves the cache line addressed by a software prefetch to a specified level of the cache. For regular applications that are dominated by loops traversing regular data structures such as arrays, optimizing compilers are effective at generating binaries with software prefetching that reduce the cache stall cycles significantly [1, 2].

Hardware prefetchers observe the behavior of a running application and initiate prefetching on repetitive patterns of cache misses. In contrast to software prefetching, hardware prefetching does not require support from an optimizing compiler or profiling support. Furthermore, hardware prefetching can automatically adapt to the dynamic behavior of the application, such as varying data sets, or the hardware, such as systems with various cache sizes. Also, the hardware prefetches are generated without the overhead of additional address-generation and prefetch instructions.

However, hardware prefetching is limited to learning and prefetching for a limited set of cache-miss patterns that are implementable in hardware. Contemporary processors, including AMD Opteron [3], Intel Xeon [4], IBM Power4 [5] and Sun UltraSPARC III [6], provide support for hardware prefetching of strided streams. For instance, if accesses to consecutive cache line addresses miss, the hardware prefetches for the third cache line and continues to prefetch successive cache lines as the prefetched cache lines are accessed by the processor. A unit stride prefetcher prefetches a contiguous sequence of cache lines. A non-unit stride prefetcher may prefetch cache lines with a stride larger than one, say every second cache line. The upcoming Fujitsu SPARC64 is an example of a processor that supports non-unit stride prefetching [7].

The focus of this work is to leverage the advantages of stride prefetchers. Stride prefetchers do not require much state because each entry can prefetch an entire stream that may span several tens or hundreds of cache lines. Since the number of streams that are active simultaneously is fairly

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'04, June 26–July 1, 2004, Saint-Malo, France.
Copyright 2004 ACM 1-58113-839-3/04/0006 ...$5.00.

small, a small structure is sufficient to prefetch for a substantial fraction of the cache misses. The miss addresses delivered to the L2 cache are the only inputs required by the stride prefetcher. Thus, the stride prefetcher does not require any modifications to the processor core and does not even need to occupy valuable space close to the processor pipeline. Current stride prefetchers are effective on many regular scientific and engineering applications that operate on large array structures. Since software prefetching is also quite effective on applications with these characteristics, there is considerable overlap in the domains where these two schemes are effective.

In this paper, we evaluate extending the scope of current stride prefetchers to multi-stride prefetchers. In contrast to current prefetchers that prefetch for a stream with a single, albeit non-unit, stride, a multi-stride prefetcher prefetches for a stream composed of multiple non-unit strides. For instance, a multi-stride stream may generate the sequence of cache line miss addresses L, L+1, L+2, L+4, L+5, L+6, L+8 and so on. This multi-stride stream has two stride components; the unit stride occurs twice followed by a single occurrence of a stride of two after which the entire pattern repeats.

Using a cache simulator, we demonstrate that such multi-stride streams are fairly common in the SPEC benchmark suite and typically contain two to four stride components. We show that multi-stride streams generally have a large stream length and that the increased learning overhead associated with multi-stride streams do not greatly inhibit effective prefetching. We list the most common multi-stride streams in the SPEC suite. We devise the Recurrent Prefetch Table (RPT) multi-stride prefetcher that prefetches for the large majority of multi-stride streams with up to four stride components. This multi-stride prefetcher can be implemented in hardware at a cost that is close to a single-stride prefetcher while also learning multi-strides in an efficient fashion.

We evaluate the performance of our RPT multi-stride prefetcher on the entire SPEC CPU suite and show that we attain significant performance gains on three SPEC benchmarks compared to current unit stride and non-unit stride prefetchers. Furthermore, we illustrate that the RPT prefetcher can be combined with an execution-driven prefetcher to deliver additional performance gains. We present and compare other metrics of the RPT prefetcher, including, prefetch coverage, prefetch efficiency and prefetch timeliness. Finally, we conduct a sensitivity analysis to demonstrate that the RPT prefetcher is effective over a range of cache configurations and memory latencies.

2. RELATED WORK

Hardware prefetching is an active area of research and several aggressive schemes have been proposed to mitigate the increasing impact of cache misses. A discussion of stride-based prefetching is followed by an overview of other prefetching mechanisms.

2.1 Stride-based prefetchers

A stride prefetcher is usually interposed between the L1 and L2 caches, monitoring the L1 cache miss addresses that are delivered to the L2 cache. Stride prefetchers partition the L1 cache misses into distinct streams, identify the stride associated with certain streams of cache miss addresses, and prefetch for misses further ahead in the stream. Stride-based prefetching is usually effective for structured workloads, such as many high performance and technical computing (HPTC) applications [8]. Next-sequential prefetching is the simplest form of stride prefetching; whenever line L is referenced, this scheme prefetches line L+1 [9, 10]. A unit-stride prefetcher observes misses for sequential cache lines and prefetches additional sequential cache lines. A unit-stride prefetcher may be limited to only ascending streams or may handle both ascending and descending streams. For instance, after observing misses for L and L-1, a general unit-stride prefetcher issues prefetches for L-2, L-3 and so on. More sophisticated stride prefetchers handle streams with non-unit strides, e.g. observing misses L, L+4, L+8, a non-unit stride prefetcher issues a prefetch for L+12 [11]. Techniques such as 2-delta stride prediction prevent single unrelated misses from causing the stride predictor to deviate from the correct stride [12]. Our work extends stride prefetching to multi-strided streams, which are composed of multiple stride components.

Typically, applications access multiple streams concurrently and the L1 cache miss addresses are composed of an interleaving of these streams. Stride prefetchers rely on an effective mechanism to decipher and disambiguate streams. If the stride prefetcher has access to the PC (program counter) of the cache missing operations, the prefetcher attributes misses to specific instructions and tracks streams on a per PC basis [13, 14, 15]. However, current implementations of processors do not propagate the PC responsible for a miss beyond the L1 cache. As a result, several schemes have been proposed to perform stream disambiguation without requiring access to PC information. Minimum delta prediction associates a miss with a stream or prior miss that is closest [11]. Memory partitioning partitions the physical memory address space into regions and attributes all misses falling within a single region to a single stream [11]. The RPT scheme presented in this paper uses the memory partitioning approach.

The prefetch-ahead distance is the number of stream elements by which the prefetcher runs ahead of data consumption by the processor. The prefetch-ahead distance is usually the number of prefetches issued when a prefetcher learns a stream. The prefetch-ahead distance is governed by several implementation-dependent factors, including memory latency, the size of the prefetch/load miss buffer, the available bandwidth and cache sizes. While the training period during which the stride prefetcher learns the behavior of the stream can be as short as one or two misses, stride prefetchers temporarily delay prefetching and/or limit the prefetch ahead distance until it becomes more certain that a new stream has been identified [16, 17]. Extending the training period has minimal performance impact, but significantly reduces the number of ineffective prefetches, reducing the bandwidth consumed [14]. Our experimental results on the RPT scheme are based on a fixed prefetch-ahead distance of eight.

A stride prefetcher issues prefetches only on encountering a missing load or additionally on detecting the consumption of a previously prefetched cache line by the processor. In the first case, the processor incurs additional misses in the steady state even after the stream is learned. In the second case, the L1 cache marks or "tags" prefetched cache lines. The L1 cache notifies the prefetcher when the processor is-

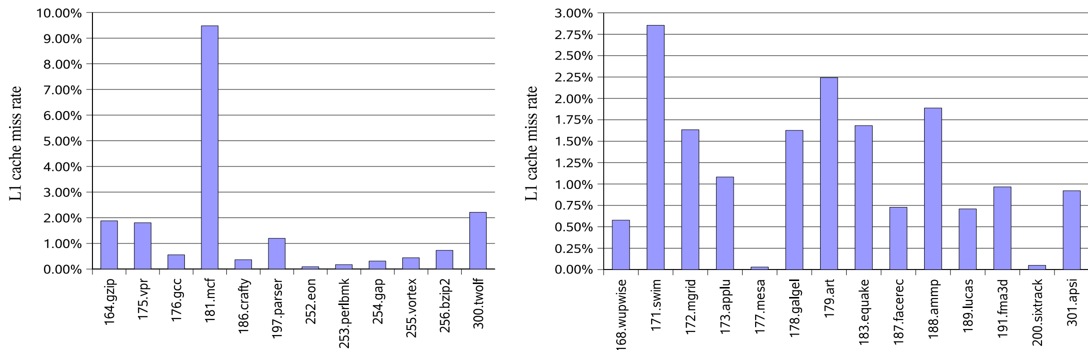


Figure 1: Data cache miss rates of SPEC CPU2000 benchmarks

sues the first load to a tagged cache line and the prefetcher initiates additional prefetches. With "tagging", apart from the misses incurred while the behavior of the stream is being learned, no further misses are necessary. Furthermore, since the prefetcher is informed as the prefetched data is consumed, additional prefetches can be issued in a more timely fashion [18]. Tagging also helps to reduce the burstiness of prefetching and ensures that prefetches are consistently ahead of data consumption by the prefetch distance, thereby improving the timeliness of prefetches. The RPT scheme described in this paper employs tagging.

Stride prefetchers either prefetch directly into a processor's caches or into dedicated prefetch structures which are queried in parallel with the processor's caches. Originally proposed by Jouppi, stream buffers are a commonly discussed form of dedicated prefetch buffers, with each stream buffer holding the prefetched data for a single stream [10]. In the original implementation, only the head of each buffer was compared with the memory address of the requests, but later refinements added support to compare requests against all the elements contained in the stream buffers [16]. Prefetching into dedicated prefetch buffers ensures that incorrect prefetches do not pollute the processor's caches [9]. Besides the additional area required for stream buffers, a major drawback of prefetch buffers is the design and verification cost associated with maintaining cache coherency between the stream buffers and the regular caches. Accordingly, the evaluations presented in this paper are based on directly prefetching data into the L1 cache.

2.2 Other prefetchers

A number of prefetching schemes with varying degrees of hardware support have been proposed. Context-based prediction schemes are capable of handling more unstructured miss behavior, essentially functioning by remembering sequences of past events and prefetching if a sequence is observed to begin repeating [9]. Consequently, these schemes require that the pattern of misses be observed at least once before prefetching can be performed [19]. The events monitored to derive the prefetch predictions vary greatly and can range from the memory addresses of the individual missing memory operations [20] to the eviction and replacement of cache lines [21] or cache tags [22]. Typically, each entry predicts and prefetches exactly one miss. Therefore, the number of entries in a context-based predictor has to be significantly larger than the number of cache lines in the cache before such prefetchers can be effective. The state associated with context-based predictors are substantial relative to the stride predictors.

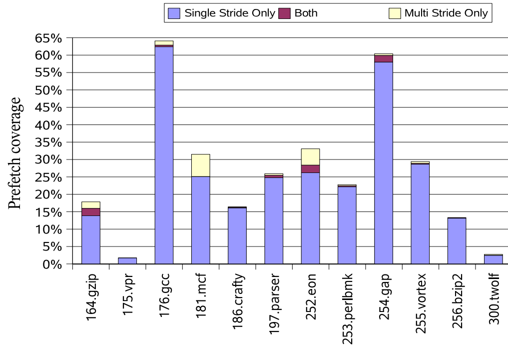
Execution driven schemes function by speculatively executing ahead of the main thread of execution in an attempt to prefetch cache missing memory accesses [23, 24]. Recently, "runahead execution" (RAE) was proposed [25, 26]. When normal execution is stalled due to, for instance, a memory operation that misses in the second-level cache, the processor transitions to runahead mode, fetches the instructions that follow the blocking load, speculatively executes the instructions that are independent of the load and converts subsequent missing loads to prefetches. Thus, each blocking load triggers the issuance of a number of prefetches that are guided by the execution of the instruction stream. In a variation of this scheme, a blocking load triggers the execution of a distilled code segment generated by a run-time optimizer with the objective of prefetching for subsequent misses [27, 28].

3. ABSTRACT STREAM ANALYSIS

In this section, the streaming behavior of load misses in the SPEC CPU2000 benchmark suite is analyzed. We investigate the extent to which the misses in a stream follow a pattern with multiple strides rather than a simple pattern with a single fixed stride. The results from this section were used to guide and justify the design of our RPT prefetcher.

3.1 Stream analysis tool

The stream analysis tool (SAT) is built on top of a trace-driven cache simulator. In order to quickly explore the design space of stride-based prefetchers, SAT is not timing-aware and processes one instruction from the trace at a time in a non-pipelined fashion. The SAT is composed of two independent stream analyzers and every load instruction in the trace that misses the L1 data cache is sent to both analyzers. The single-strided analyzer detects and tracks single-strided streams and the multi-strided analyzer detects and tracks multi-strided streams. Each analyzer reports independently whether a particular missing cache line is covered and belongs to a stream of a certain type. A particular missing cache line address may be reported as covered by either/none/both the single-strided analyzer and the multi-strided analyzer. Since a multi-stride stream consists of shorter segments of single-strided streams, segments of the



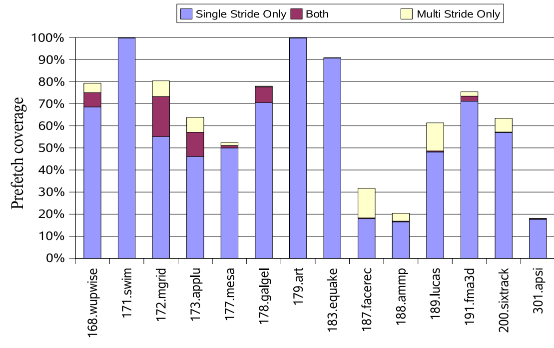
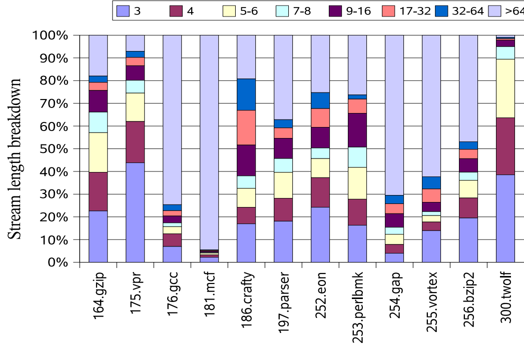


Figure 2: Coverage of single-stride and multi-stride stream analyzers



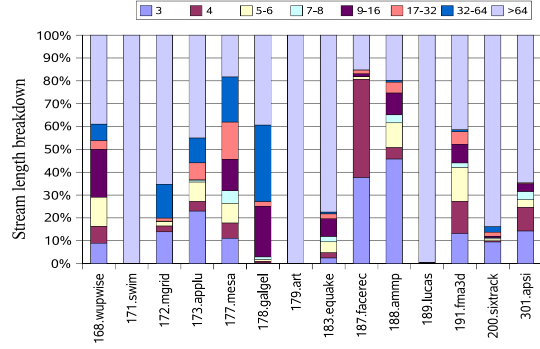


Figure 3: Stream lengths of single-strided and multi-strided streams

multi-strided stream are covered by both types of streams, if the single-strided stream segment is sufficiently long.

The operation of the two stream analyzers is very similar. Both analyzers contain two tables, one for learning and another for predicting. When the analyzer learns a stream pattern, it transfers the stream to the predictor table. On each missing line address, the analyzer first searches the predictor table. A match indicates that the line address is covered and prefetchable by that analyzer. Otherwise, the analyzer looks for a match in the learning table. On a match, it updates the learning table entry, and if the pattern is learned, the entry is transferred to the predictor table. When there is no exact or close match in either table, the analyzer allocates a new entry in the learning table for this miss address. In both analyzers, the sizes of the learning and predictor table are each 64 entries and these tables are managed using an LRU replacement policy.

In the single-stride analyzer, a pattern is learned as soon as the same stride has been seen twice. For example, if the following stream of misses are seen: L, L+2, L+4, L+6, L+8, and so on, the pattern is learned when the miss L+4 is observed and the misses starting from L+6 are considered covered and prefetchable. In the multi-stride analyzer, a pattern is learned as soon as the same pattern is seen twice. For example, consider the multi-stride pattern 1[3] 2[2], which consists of two stride components, a stride of one occurring thrice and a stride of two occurring twice. This multi-stride pattern generates the following stream of misses: L, L+1, L+2, L+3, L+5, L+7, L+8, L+9, L+10, L+12, L+14, L+15 and so on. The pattern can be learned when the miss L+15 is seen and misses starting from L+15 are considered covered and prefetchable. The multi-stride analyzer employs a greedy learning algorithm. Specifically, it searches for the shortest recurring pattern. The multi-stride analyzer recognizes multi-stride streams with up to 16 stride components.

3.2 Miss rates and coverage

The trace snapping methodology and the cache hierarchy parameters for the experiments conducted using the stream analyzer are described in Section 5. Figure 1 shows the data cache load miss rates of the SPEC CPU2000 benchmarks (the left half of the benchmarks –from 164.gzip to 300.twolf– are the integer benchmarks, while the right half are the floating-point benchmarks). The `181.mcf` benchmark has by far the highest L1 miss rate of over 9%. In general, the integer benchmarks have lower miss rates than the floating-point benchmarks, although `181.mcf`, `164.gzip`, `175.vpr`, `197.parser` and `300.twolf` are notable exceptions.

Figure 2 shows the coverage of L1 cache misses by either or both the single-strided and multi-strided streams. The lowest bar segment for each benchmark indicates the percentage of cache misses covered by the single-strided stream only, the middle bar segment indicates the percentage of cache misses that are covered both by the single- and multi-strided streams and the top bar segment indicates the percentage of cache misses covered only by the multi-strided stream. Thus, the overall bar indicates the percentage of cache misses covered by either the single- or multi-strided streams. The results show that stream coverage is not very

Stride pattern	Fractional coverage	Cumulative coverage
-2[1]-1[1]	0.343	0.343
1[2]2[1]	0.087	0.430
-16[1]17[1]	0.059	0.489
-1[1]3[1]	0.050	0.539
1[73]2[1]**	0.032	0.571
2[1]5[1]	0.029	0.600
-16[2]33[1]	0.029	0.629
-1[1]2[1]1[1]3[1]	0.027	0.656
-5[1]1[2]	0.024	0.679
1[2]3[1]	0.023	0.702
-1[1]2[1]1[179]7[1]1[181]7[1]**	0.023	0.725
1[1]19[1]	0.019	0.744
1[83]8[1]	0.014	0.758
1[9]9[1]**	0.014	0.772
1[3]2[1]1[3]2[3]*	0.010	0.782
-2[1]4[1]1[1]*	0.009	0.790
-1[1]2[2]	0.007	0.798
1[75]16[1]**	0.007	0.804
1[2]16[1]	0.006	0.811
1[4]2[1]	0.006	0.817

* - not detected by RPT
** - detected as single stride stream by RPT

Table 1: Most frequently occurring multi-strided stream patterns

effective on the integer benchmarks with high load miss rates, but is highly effective for many of the floating-point benchmarks with high load miss rates. The results also show that for some benchmarks, multi-stride streams increase overall coverage significantly over single-stream coverage, e.g. `187.facerec` and `189.lucas`. For `181.mcf`, even though the increase is not as substantial, the increase still has a potentially significant impact on performance because of the high load miss rate.

3.3 Stream lengths

Figure 3 shows the distribution of the lengths of the streams detected by the single-stride analyzer and the multi-stride analyzer. This distribution is weighted by the length of the stream. Therefore, a stream of length l00 that is seen twice contributes more to the distribution than a stream of length 10 that is seen 15 times. For simplicity, the stream lengths recorded by the single-stride analyzer and the stream lengths recorded by the multi-stride analyzer are aggregated together for this figure. The results show that the majority of the benchmarks with a significant load miss rate and good strided stream coverage have long stream lengths. Notable exceptions to this are `197.parser` and `187.facerec`. The prevalence of long streams suggest that the longer learning required by multi-stride stream predictors is not a severe performance impediment.

3.4 Stream Patterns

Table 1 shows the 20 most frequently occurring multi-strided stream patterns. Each match of a load miss to a multi-strided stream predictor table entry with that stream pattern counts as one occurrence. Across the 26 SPEC CPU2000 benchmarks, the multi-strided stream analyzer detected 20,370 unique multi-strided stream patterns. The first column in the table shows the stream patterns. Each pattern is represented by a sequence of s[n] where s is the stride and n is the number of occurrences of that stride. The second column in the table shows the number of occurrences of that pattern as a fraction of the total number of all matches to the multi-strided stream predictor. The third column is simply the accumulation of the second column up to this row and shows that these top 20 patterns account for 82% of all the matches to the multi-strided stream predictor. Thus, a few multi-strided patterns account for a vast majority of all dynamic multi-strided stream accesses. Furthermore, these multi-strided patterns are in general not overly complex and consist of a fairly small number of stride components. We use the insights gained through our analysis of streams to develop and justify the RPT multi-stride prefetcher, described in the next section.

4. RPT - A MULTI-STRIDE PREFETCHER

The stream behavior described in the previous section demonstrates that there is significant headroom for improving current single-stride prefetchers. Multi-stride streams with four or fewer stride components account for a vast majority of the commonly occurring streams in SPEC CPU2000.

The Recurrent Prefetch Table (RPT) prefetcher is a two-state, four-stride, prefetching scheme, which can support streams with up to two "steady-state" strides and two transitional strides, as illustrated in Figure 4. While its complexity is comparable to that of a two state scheme, the support for the two transitional strides allows the proposed RPT prefetcher to correctly capture the behavior of many four-stride streams, such as the stream illustrated in Figure 5. Overall, the RPT prefetcher covers 18 of the 20 multi-stride streams illustrated in Table 1 (the streams marked with a single asterisk are the only streams that cannot be covered).

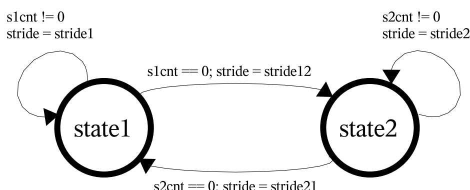


Figure 4: The state transition diagram for the RPT prefetcher

Figure 4 shows the state diagram of multi-stride streams covered by the RPT prefetcher. The multi-stride stream remains in *state1*, with stride *stride1*, for *s1cnt* iterations, before transitioning, with stride *stride12*, to *state2*. The stream then remains in *state2*, with stride *stride2*, for *s2cnt* iterations, before transitioning, with stride *stride21*, back to *state1* and so on.

While the proposed RPT prefetcher is enhanced to handle multi-stride streams, such as the stream illustrated in Figure 5, the RPT prefetcher still covers single-stride streams. These simpler streams account for the majority of streams and retaining good single stride performance is imperative to good prefetcher performance.

We now describe the proposed RPT prefetcher in more detail. As described previously, the physical memory address space is partitioned into regions and all misses in a region are assigned to the same stream. The high-order bits of the physical address that identify the region are referred to henceforth as the "tag bits". The remaining low-order bits of the cache line address, referred to as the index, identify the cache line's position within a region. All missing

cache line addresses with the same tag are part of the same stream.

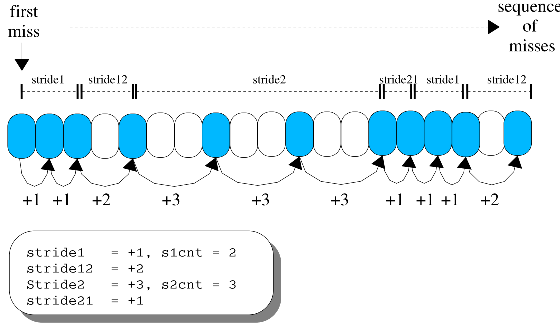


Figure 5: A multi-stride stream, as perceived by the RPT prefetcher

The proposed RPT prefetcher consists of two tables, the stream table and the index table, which are illustrated in Figure 6. The stream table is a fully associative structure which contains the tags associated with all of the active streams, while the index table contains the state of the prefetcher for each of the streams contained in the stream table. The stream state includes: the mode of the prefetcher (*learning* or *trained*), the total states for the stream (*numstates*), the current state of the stream (*current_state*), the stride associated with each of the states (*stride1* and *stride2*), the transitional strides observed when moving between the two states (*stride12* and *stride21*), an indication of how long to stay in one state before transitioning to the other state (*s1cnt* and *s2cnt*), the index of the last observed miss (*last_index*) and an indication of how long the stream has spent in its current state (*count*).

The lifetime of a stream entry, from its allocation in the stream table to its eviction from the stream table is now outlined.

Allocation: for every missing load, the tag bits of the memory address are used to search the stream table. If the tag fails to match any of the valid tags currently held in the stream table, the miss is not considered to belong to any of the known streams and a new entry is allocated in the stream table. The corresponding entry in the index table is also initialized and the *last_index* field is updated with the index of the missing load.

Training: subsequent misses with the same tag are considered to belong to the same stream and cause the training of the prefetcher to commence. The prefetcher must, after observing only a minimal number of misses, determine both the number and size of the strides associated with the stream. The stream's initial stride is computed by subtracting the *last_index* field, contained in the index table, from the index of the miss. The *s1cnt* count, which is a count of how long to remain in *state1*, is also incremented, and the *last_index* field is updated to account for this new miss.

If subsequent misses in the same stream continue to display an identical stride, RPT rapidly deduces that this appears to be a single stride stream. This occurs when the *s1cnt* count exceeds the single-stride confidence threshold and causes the entry to transition from a learning state to a single-stride trained state, in which it is ready to start issuing prefetches for the stream.

This simple training example covers streams with a single stride. If, during training, a miss deviates from the expected stride, the prefetcher recognizes that a multi-stride stream may have been detected and transitions to its second state, *state2*. The *state12* transitional stride is set using the observed stride of the miss, while the subsequent miss sets *stride2*, the stride associated with *state2*.

The next miss with a stride deviating from *stride2* is assumed to represent a transition back to the first state and the observed stride is used to set the transitional stride *stride21*.

Subsequent misses are expected to display stride *stride1* for *s1cnt* iterations, then *stride12* and so on. For a multi-stride stream to transition from a learning state to a trained state, the prefetcher must correctly predict the stride for a predetermined number of misses. This is tracked by a recurrence counter, *recnt*, which is incremented after each correct prediction. Once *recnt* passes the multi-stride confidence threshold, the entry transitions from a learning state to a multi-stride trained state, in which it is ready to start issuing prefetches for the stream.

Failure of the observed strides to match with the predicted strides causes training to fail - the behavior of the stream cannot currently be modeled by the prefetcher and the entry remains in a training mode, monitoring misses until a recognizable pattern is observed.

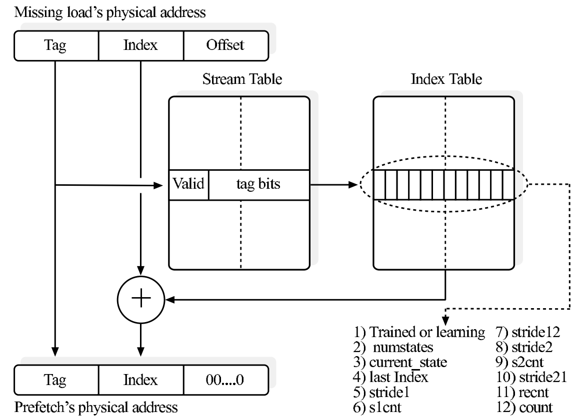


Figure 6: An overview of the RPT multi-stride prefetcher

Prefetching: once the behavior of a stream has been learned by the prefetcher, if the subsequent miss is compliant (compliant iff the observed stride matches the predicted stride for the state), the prefetcher issues a number of prefetches for the stream. The stride of the issued prefetches is governed by the state of the prefetcher and the stride associated with that state. When the sequence of issued prefetches spans multiple states, prefetches with different strides may be issued, enabling the prefetcher to follow the predicted behavior of the stream.

As RPT supports tagging, the prefetcher is informed when previously issued prefetches are consumed by the processor. Following each consumption notification, the stride of the latest access in the stream is checked against the predicted stride and, if the stride has been correctly predicted, the *last_index* field is updated to reflect the index of the access

and a prefetch is issued for the next unprefetched element in the stream.

Eviction: the stream table is managed using a least recently used (LRU) replacement scheme.

5. METHODOLOGY

Evaluation of the proposed multi-stride prefetcher has focused on determining its effectiveness on the SPEC CPU2000 suite. The benchmarks were compiled with a recent Sun compiler using the same aggressive optimization flags that Sun used for reporting their SPEC CPU2000 base results. To capture the behavior of each benchmark accurately, approximately 50 snaps were simulated for each benchmark. In addition to a trace of 6 million instructions and their associated register and memory values, each snap is annotated with cache warming information from the preceding 300 million instructions. Of the 6 million trace instructions in each snap, the first 5 million are used by the processor simulator for warming the branch predictors and the HW prefetchers, while the last 1 million are used for collecting performance statistics.

While SPEC CPU2000 is built using aggressive compiler flags that trigger the insertion of software (SW) prefetches, the majority of ISV (Independent Software Vendor) applications are not compiled with such aggressive flags, curtailing the compiler's ability to insert prefetches. It is in conjunction with these unoptimized applications, where SW prefetching support is essentially absent, that autonomous HW prefetchers can prove most beneficial. We therefore present the performance gains achieved by the HW prefetchers, both with and without SW prefetch support (to provide data for the without SW prefetching scenario, the SW prefetches in the snaps are dropped i.e. not executed). By presenting the performance gains achieved for both extremes of SW prefetching support, we better illustrate the benefit of the HW prefetchers to "real-world" applications. Unless otherwise mentioned, the results presented are without SW prefetch support.

The processor simulator used is an in-house cycle-accurate timing simulator that models an out-of-order processor with a 64 entry reorder buffer. The L1 instruction and data caches are each 32KB (4-way set associative, 64-byte line size, 4 cycle latency) and the unified L2 cache is 1MB (8-way set associative, 64-byte line size, 25 cycle latency). The modeled memory latency is 400 cycles (5GB/sec bandwidth), while the branch predictor is a 64K entry gshare predictor. While the simulator is trace-driven, it is augmented with an instruction set emulator. Using the register and memory values from the snaps, the emulator allows the simulator to accurately model control and data speculation, thereby enabling the simulator to achieve the accuracy of execution-driven simulation.

The chosen performance metric is instructions per cycle (IPC) and we present the percentage performance gain relative to the baseline system. Results are also presented for prefetch coverage, efficiency and timeliness. Prefetch coverage provides a measure of the effectiveness of the prefetcher at reducing the number of cache misses observed by an application and is computed by comparing the total number of useful prefetches (useful prefetches are defined to be prefetches that reduce the latency of a cache missing memory operation) issued by the prefetcher, *useful_prefetches*, to the total number of misses originally incurred by the application, *total_misses*, as illustrated in equation 1. Prefetch efficiency provides a measure of the accuracy of the prefetcher, comparing the number of useful prefetches issued by the prefetcher to the total number of prefetches issued,

total_prefetches, as is illustrated in equation 2. Prefetch timeliness provides a measure of how effective the useful prefetches were at reducing the latency of the targeted memory operations and is computed by determining the separation, in cycles, of the prefetch and the target memory operation, *cycles*, as is illustrated in equation 3.

$$coverage = \frac{useful_prefetches}{total_misses} \times 100 \quad (1)$$

$$efficiency = \frac{useful_prefetches}{total_prefetches} \times 100 \quad (2)$$

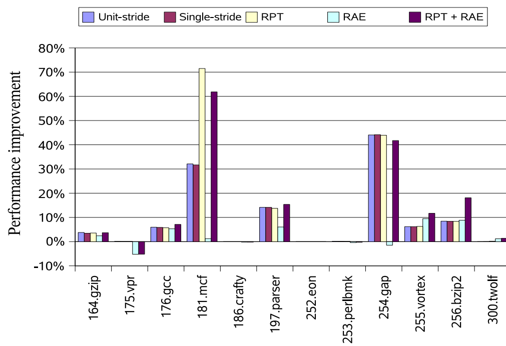
$$timeliness = \frac{\sum_{useful_prefetches} min(miss_latency, cycles)}{\sum_{useful_prefetches} miss_latency} \quad (3)$$

To analyze the performance benefits of the proposed multi-stride prefetcher, its performance is compared to the chosen baseline architecture with no HW prefetch support and the performance observed when the baseline architecture is augmented with i) a unit stride prefetcher and ii) a single non-unit stride prefetcher. It was decided to compare the proposed RPT prefetcher with these two schemes, because they:

- **Represent the implemented state of the art:** IBM's POWER4 processor [5], Intel's Pentium 4 processor [4], AMD's Opteron processor [3], and Sun's UltraSPARC-III processor [6] all support unit stride prefetchers, while Fujitsu's upcoming SPARC64 VI processor supports a single non-unit stride prefetcher [7].
- **Don't require PC information:** as previously discussed, supplying the HW prefetcher with PC information is costly, forcing a focus on schemes that can function without this information (in contrast to many recently proposed schemes [29, 22, 21, 19, 30, 31]).
- **Low-cost:** many of the proposed schemes, especially context-based prefetchers, are expensive to implement, due to the amount of state information they require to perform effectively. Consequently, they are not well suited to aggressive multi-core processors, the current industry trend, where area is at a premium and multiple instantiations of the HW prefetchers may be required to support prefetching into each core's L1 cache.

The unit and single stride prefetchers investigated are implemented in an identical fashion to the proposed multi-stride prefetcher, using the region-wise stream partitioning, prefetching directly into the L1 data cache and leveraging a tagged prefetched scheme. Additionally, no separate allocation filter is provided, the 16-entry stream table (with LRU replacement) being correctly sized to fulfill this requirement - data is subsequently provided to support this claim.

The trigger events for the HW prefetchers are loads which miss in the L1 data cache and the consumption of previously issued prefetches. For the first compliant miss, eight prefetches are issued, with the strides of the prefetches mirroring the predicted stride(s) of the stream. Prefetches are



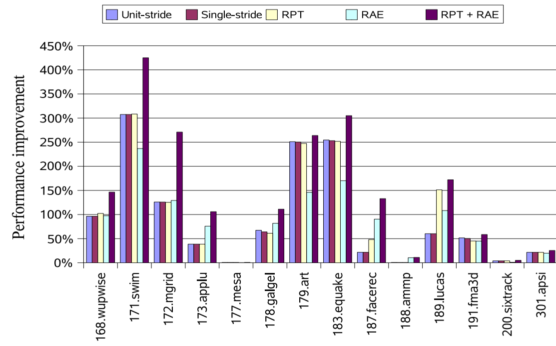
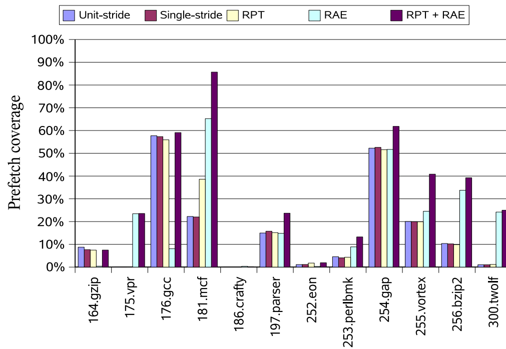


Figure 7: Performance gains achieved by the HW prefetching schemes for SPEC CPU2000 (compared to a system without HW prefetch support)



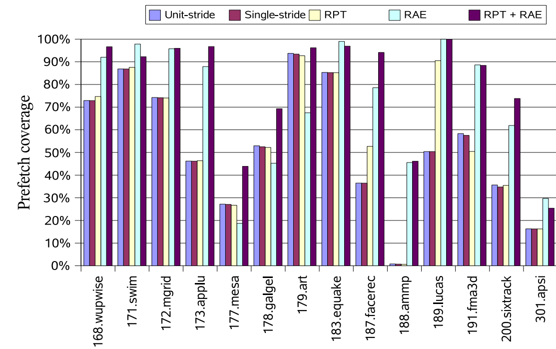


Figure 8: Prefetch coverage of the HW prefetching schemes for SPEC CPU2000

also issued in response to the application's utilization of previously prefetched data, allowing the prefetchers to maintain a constant prefetch-ahead distance. For the proposed multi-stride prefetcher, four correctly predicted miss sequences are required to cause the entry to transition to a multi-stride trained state, while four consecutive misses with identical strides are required for the prefetcher to transition to a single-stride trained state. For the unit stride and single non-unit stride prefetchers, training is completed after three compliant misses, as proposed in [16].

The tag bits, which are used to identify streams, are defined as bits 13 and upward of the physical address of the memory operations, resulting in 8KB memory partitioning; chosen to match the default 8KB page size in Solaris.

The performance gains observed using the stride prefetchers are also contrasted with the performance gains obtained using execution driven prefetching techniques, specifically runahead execution (RAE). Additionally, the performance benefits of employing both stride prediction and RAE in unison is investigated. RAE is triggered on an L2 cache miss and is terminated when data is returned.

6. SIMULATION RESULTS

Figure 7 compares, for SPEC CPU2000, the performance gains achieved by each of the HW prefetching schemes, compared to a baseline without HW prefetching support. From Figure 7, it is apparent that HW prefetching provides a significant performance benefit for the majority of the SPECfp benchmarks and a number of the SPECint benchmarks, most notably `gap` and `mcf`. It is also apparent that the single stride prefetcher appears to provide no noticeable performance advantage over the unit stride prefetcher – as noted in [14].

The proposed RPT multi-stride prefetcher is noticeably more effective for `mcf` in SPECint and `facerec` and `lucas` in SPECfp, providing an additional performance gain of up to 57% over the unit and single stride prefetchers. In the few benchmarks where RPT doesn't achieve quite as much performance gain as the single and unit stride prefetchers, this can be attributed to the additional miss RPT requires to identify a single stride stream. For applications with no multi-stride streams and very short single stride streams, this slight increase leads to the slight decrease in the performance gains observed.

Additionally, for the majority of SPECfp benchmarks and several SPECint benchmarks, there is a noticeable performance advantage to supporting both strided and execution driven HW prefetching schemes, with RPT and RAE together exhibiting a performance gain up to 66% larger than either scheme in isolation.

Figure 8 analyzes the prefetch coverage achieved by the HW prefetch schemes, illustrating significant coverage, especially for SPECfp. The coverage achieved by the RPT prefetcher is significantly higher than achieved by the unit and single non-unit stride prefetchers for `mcf`, `facerec`, and `lucas`, and leads to the significant additional performance gains achieved by RPT for these benchmarks. The syner-

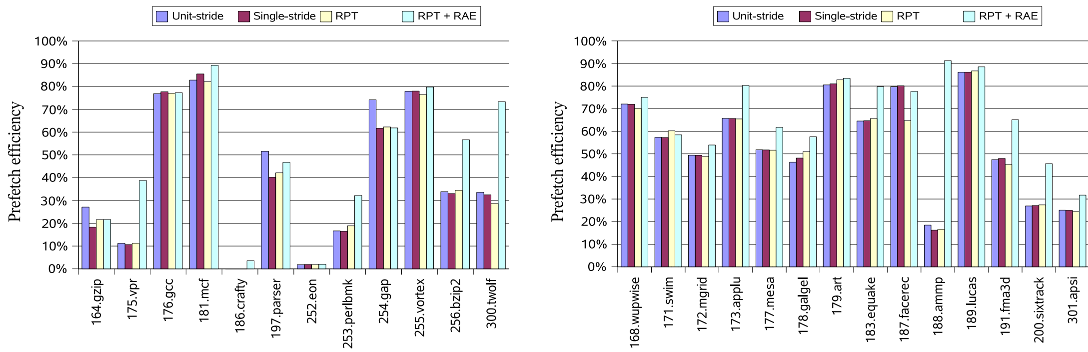


Figure 9: Prefetch efficiency of the HW prefetching schemes for SPEC CPU2000

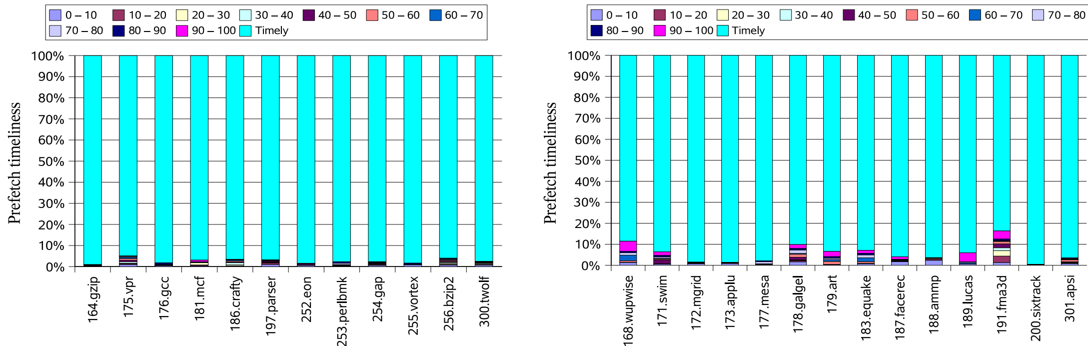


Figure 10: Prefetch timeliness of the RPT prefetcher (% of miss latency eliminated) for SPEC CPU2000

gistic behavior of the RAE and strided prefetchers is also illustrated by the increased coverage observed when both RPT and RAE are employed together.

Figure 9 presents the prefetching efficiency associated with each of the chosen HW prefetch schemes, illustrating that the vast majority of the prefetches issued are useful, especially for SPECfp. Indeed, for several applications, as many as 8 out of every 10 prefetches issued are effective. In all of the applications, the efficiency of the RPT prefetcher is similar to the efficiency of the unit and single stride prefetchers.

Figure 10 analyzes the timeliness of the prefetches issued by the proposed RPT prefetcher, and it is apparent that, for all of the benchmarks, the vast majority of prefetches (in some benchmarks, up to 95%) are completely timely, completely eliminating associated load miss stalls.

To illustrate that the size of the stream table used by the strided prefetchers is not a performance limiter, Figure 11 illustrates the performance of the RPT prefetcher for a variety of different sizes of the stream table. While it is illustrated that an overly small stream table can limit the performance benefits of the prefetcher, it is apparent that the performance of the 16-entry stream table used throughout this paper provides similar performance as a 10K entry stream table.

Figure 12 analyzes the performance benefits of the HW prefetchers when software prefetching support is provided. In contrast to the performance gains illustrated in Figure 7, the magnitude of the performance gains achieved by the HW prefetchers is reduced by SW prefetching support, but the additional performance gains achieved by supporting RPT remain significant.

Figure 13 and 14 illustrate, for the SPEC CPU2000 benchmarks for which the largest RPT performance gains were observed, the variation of the performance gains observed as the size of the L2 cache and the latency of main memory are varied (no SW prefetching support). Figure 13 illustrates that, while the benefit of the HW prefetchers is diminished as the cache size is increased (which reduces cache miss rates), significant additional performance benefit is still derived from supporting RPT. Figure 14 illustrates that, as the memory latency is increased, the performance benefits associated with the HW prefetchers is increased. This is especially true for RAE; the increasing duration of the stalls associated with the L2 cache misses, allows the speculative execution to progress further down the predicted path and uncover additional load misses.

7. CONCLUSION

Cache missing memory operations represent a performance bottleneck on modern high performance processors. Many current processors support HW stream prefetchers that attempt to reduce the misses observed by applications. Prior strided prefetchers were limited to trying to predict the behavior of a stream using a single stride.

Our analysis illustrates that multi-stride streams occur in many applications, as illustrated by the SPEC CPU2000 benchmarks and that supporting a multi-stride prefetcher

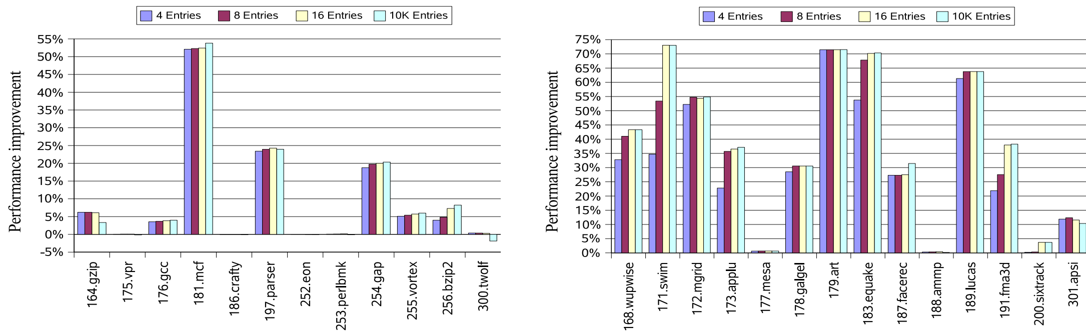


Figure 11: Performance gains achieved by the RPT prefetcher for SPEC CPU2000 with different sizes of stream table

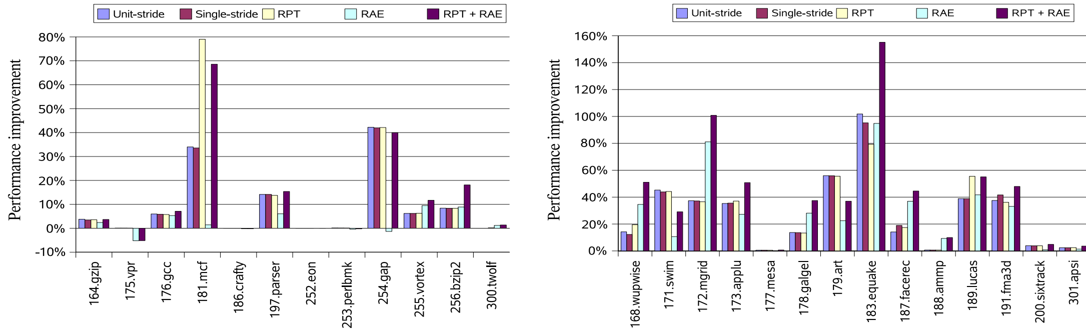


Figure 12: Performance gains achieved by the HW prefetching schemes for SPEC CPU2000; SW prefetching supported

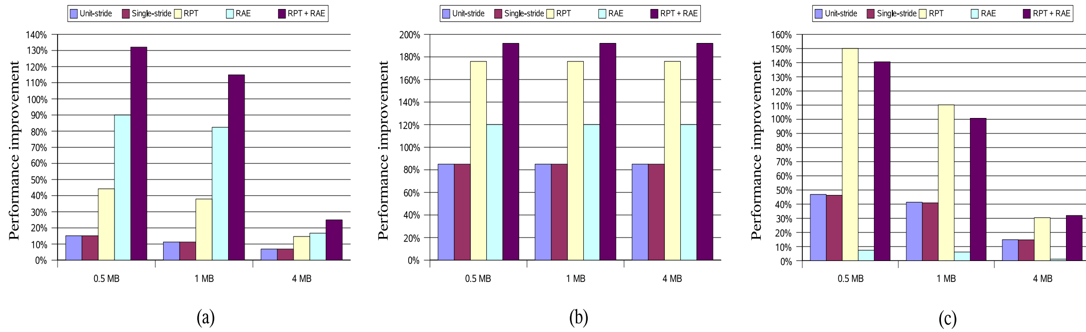


Figure 13: Sensitivity of performance to cache size for (a) `facerec`, (b) `lucas` and (c) `mcf`

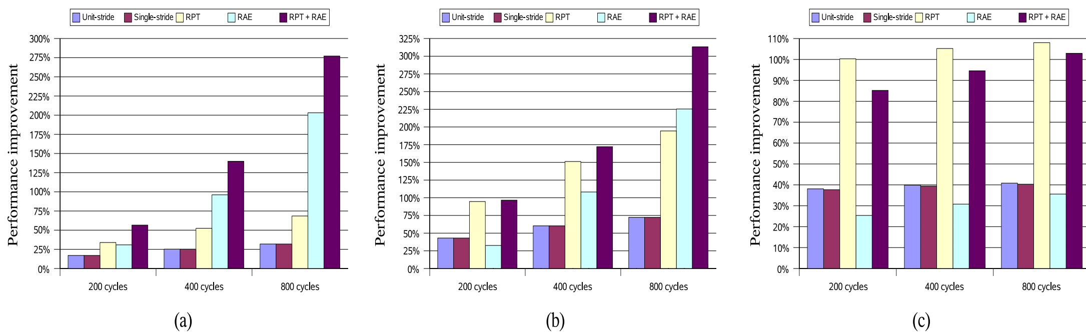


Figure 14: Sensitivity of performance to memory latency for (a) `facerec`, (b) `lucas` and (c) `mcf`

can achieve significant additional performance gains over a single stride prefetcher for those applications.

In response to this analysis, we proposed the RPT multi-stride stream prefetcher, capable of supporting up to four distinct strides. On several SPEC CPU2000 benchmarks, notably `mcf`, `facerec` and `lucas`, RPT's prefetching for these multi-stride streams results in significant performance improvements.

Finally, we conclude that it is beneficial to support run-ahead execution as well as RPT, with significant additional performance gains being achieved over that achievable by each scheme in isolation.

8. REFERENCES

[1] M. S. Lam, T. Mowry, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 62–73.

[2] P. Tirumalai et al., "Processor aware anticipatory prefetching in loops" in *Proceedings of 10th International Symposium on High Performance Computer Architecture*, 2004, pp. 106–118.

[3] Advanced Micro Devices, *Software Optimization Guide for AMD Athlon 64 and AMD Opteron Processors*, 2003.

[4] Intel Corporation, *Intel Pentium 4 and Intel Xeon Processor Optimization*, 2001.

[5] S. Fields, H. Le, J. M. Tendler, S. Dodson, and B. Sinharoy, "POWER4 system microarchitecture," *Technical white paper*, 2001.

[6] Sun Microsystems, *UltraSPARC IIIcu user's manual*, 2004.

[7] T. Maruyama, "SPARC64 VI: Fujitsu's next generation processor," in *Proceedings of the Microprocessor Forum*, 2003.

[8] Y. Jegou and O. Temem, "Speculative prefetching," in *Proceedings of the International Conference on Supercomputing*, 1992, pp. 1–11.

[9] M. J. Charney and T. R. Puzak, "Prefetching and memory system behavior of the SPEC95 benchmark suite," *IBM Journal of Research and Development*, vol. 41, pp. 265–286, 1997.

[10] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers," in *Proceedings of the 17th International Symposium on Computer Architecture*, 1990, pp. 364–373.

[11] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of the 21th International Symposium on Computer Architecture*, 1994, pp. 24–33.

[12] R. J. Eickemeyer and S. Vassiliadis, "A load instruction unit for pipelined processors," *IBM Journal of Research and Development*, Vol. 37, pp. 547–564, 1993.

[13] C. Zhang and S. A. McKee, "Hardware-only stream prefetching and dynamic access ordering," in *Proceedings of the 14th International Conference on Supercomputing*, 2000, pp. 167–175.

[14] N. Jouppi, K. Farkas, P. Chow, and Z. Vranesic, "Memory-system design considerations for dynamically-scheduled processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 133–143.

[15] J. W. C. Fu, J. H. Patel, and B. L. Janssens "Stride directed prefetching in scalar processors," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1992, pp. 102–110.

[16] N. D. Jouppi, K. Farkas, and P. Chow, "How useful are nonblocking loads, stream buffers, and speculative execution in multiple issue processors?," in *Proceedings of the 1st International Symposium on High Performance Computer Architecture*, 1995, pp. 78–89.

[17] M. Dubois, F. Dahlgren, and P. Stenstrom, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, pp. 733–764, 1995.

[18] V. Srinivasan, E. S. Davidson, and G. S. Tyson, "A prefetch taxonomy," *IEEE Transactions on Computers*, vol. 53, pp. 126–140, 2004.

[19] S. Sair, T. Sherwood, and B. Calder, "Predictor-directed stream buffers," in *International Symposium on Microarchitecture*, 2000, pp. 42–53.

[20] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, 1999.

[21] A. Lai et al., "Dead-block prediction and dead-block correlating prefetchers," in *the 28th Annual International Symposium on Computer Architecture*, 2001, pp. 52–62.

[22] Z. Hu, M. Martonosi, and S. Kaxiras, "TCP: Tag correlating prefetchers," in *9th International Symposium on High-Performance Computer Architecture*, 2003, pp. 317–327.

[23] J. L. Baer and T-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 176–186.

[24] T-F. Chen and J-L. Baer, "Effective hardware-based data prefetching for high performance processors," in *IEEE Transactions on Computers*, 1995, pp. 609–623.

[25] J. Dundas and T. N. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *International Conference on Supercomputing*, 1997, pp. 68–75.

[26] O. Mutlu et al., "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *9th International Symposium on High-Performance Computer Architecture*, 2003, pp. 129–141.

[27] D. Kim and D. Yeung, "Design and evaluation of compiler algorithms for preexecution," in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[28] H. Wang, G. Hoflehner, D. Lavery, S. Liao, P. Wang, and J. Shen, "Post-pass binary adaptation for software-based speculative precomputation," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002, pp. 117–128.

[29] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 115–126.

[30] K. Nesbit and J. Smith, "Neighborhood prefetching on multiprocessors using instruction history," in *Proceedings of 10th International Conference on High Performance Computer Architecture*, 2004, pp. 96–106.

[31] D. M. Koppelman, "Neighborhood prefetching on multiprocessors using instruction history," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2000, pp. 123–133.