Spectral Prefetcher: An Effective Mechanism for L2 Cache Prefetching

SAURABH SHARMA, JESSE G. BEU, and THOMAS M. CONTE
North Carolina State University

Effective data prefetching requires accurate mechanisms to predict embedded patterns in the miss reference behavior. This paper proposes a novel prefetching mechanism, called the spectral prefetcher (SP), that accurately identifies the pattern by dynamically adjusting to its frequency. The proposed mechanism divides the memory address space into tag concentration zones (TCzones) and detects either the pattern of tags (higher order bits) or the pattern of strides (differences between consecutive tags) within each TCzone. The prefetcher dynamically determines whether the pattern of tags or strides will increase the effectiveness of prefetching and switches accordingly. To measure the performance of our scheme, we use a cycle-accurate aggressive out-of-order simulator that models bus occupancy, bus protocol, and limited bandwidth. Our experimental results show performance improvement of 1.59, on average, and at best 2.10 for the memory-intensive benchmarks we studied. Further, we show that SP outperforms the previously proposed scheme, with twice the size of SP, by 39% and a larger L2 cache, with equivalent storage area by 31%.

Categories and Subject Descriptors: B.3.2 [**Memory Structure**]: Design Style—*Cache memories*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Prefetch, L2 cache, autocorrelation, frequency, adaptive, absolute and differential domain, memory

1. INTRODUCTION

The past two decades has witnessed tremendous advances in semiconductor process technology and micro-architecture, exponentially reducing processor cycle times. Meanwhile, access times of memory have decreased at a glacial rate of 10% per year. Consequently, memory latencies measured in processor cycles are continually increasing and are on the order of hundreds of cycles. To bridge the processor-memory latency gap, computer architects have primarily relied on high-speed cache memories. Because of size constraints, however, on-chip caches are unable to keep up with the growing data requirements of applications. As a result, important classes of applications suffer from high-cache miss rates and, subsequently, performance degradation.

Authors' addresses: Saurabh Sharma, Jesse G. Beu and Thomas M. Conte, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695.

In addition to high-speed caches, many architects rely on prefetching, which is shown to be a primary technique to mask or eliminate memory latencies. Prefetching works by anticipating data misses and fetching the data before the processor requires it. While several models have been proposed for prefetching either via hardware [Charney and Reeves 1995; Chen and Baer 1992; Cooksey et al. 2002; Hu et al. 2003; Joseph and Grunwald 1999; Jouppi, 1990; Lai et al. 2001; Nesbit and Smith 2004; Nesbit et al. 2004; Palacharla and Kessler 1994; Roth et al. 1998] or software [Lipasti et al. 1995; Luk and Mowry, 1996; Mowry et al. 1992], hardware implementations are more popular, because of the availability of run-time information, which can significantly improve the effectiveness of prefetching. Many previous proposals for hardware prefetchers target specific patterns observed in the reference stream, such as strided accesses [Chen and Baer 1992; Jouppi 1990; Palacharla and Kessler 1994] and accesses to linked-data structures [Cooksey et al. 2002; Roth et al. 1998]. These classes of prefetchers are effective for specific access patterns, but have limited applicability across a wider range of application programs.

A more generic hardware approach is correlation-based prefetching (CP) [Charney and Reeves 1995; Hu et al. 2003; Joseph and Grunwald 1999; Lai et al. 2001; Nesbit and Smith 2004; Nesbit et al. 2004], which compares future memory references with past memory behavior to prefetch repetitive reference patterns. The address predictor of CP rely on the hypothesis that any given sequence (in this case, the missed address stream), with or without stride locality, will repeat itself. Unfortunately, CP suffers from several key shortcomings. First, rather than recording only the repeating patterns for prediction, these prefetchers record all the misses, both the repeating pattern as well as random noise, present in the missed address stream. Predictions based on these random elements result in lower prediction accuracy. Second, these prefetchers are usually trained with L1 cache miss streams for prefetching, which are often clustered in out-of-order engines. As a result, prefetching hardware is required to be fast so it can intercept the patterns present in the miss stream, while being large enough to record all miss instances. Moreover, a considerable fraction of the L1 miss stream results in L2 access hits, which are usually tolerated by an aggressive out-of-order processor. Finally, CP does not offer both high coverage (the fraction of demand cache misses resolved by the prefetcher) and high prediction accuracy (the fraction of the data offered by the prefetcher that was used) [Joseph and Grunwald 1999].

This paper proposes the *Spectral Prefetcher*, a novel mechanism for cache prefetching that captures frequencies within a pattern in the cache miss sequence. The spectral prefetcher, as proposed, is aimed specifically at prefetching into the L2 cache by inspecting the L2 cache miss stream. Frequencies are defined in terms of *recurring distances* (lags)—the number of miss events observed between the reappearance of the data item in the miss stream. When more than one miss is observed with the same recurring distance, the prefetching hardware assumes a pattern and begins recording for future prediction. It dynamically partitions the physical address space in a strided fashion and detects the tag pattern (higher order bits) within each partition. Two missed references are within the same partition, referred to as tag concentration zones

(TCzones), if their addresses have the same low order bits (i.e., they map to the same cache set). This approach is similar to the tag correlating prefetcher [Hu et al. 2003] that predicts the pattern of tags with the same cache index (low-order bits).

Ideally, a prefetcher should capture any pattern in the missed address stream and offer the processor all the data it needs for processing. Prior research in correlated prefetching has tried to capture either the patterns in absolute values [Charney and Reeves 1995; Hu et al. 2003; Joseph and Grunwald 1999] or the stride patterns among the absolute values [Nesbit and Smith 2004; Nesbit et al. 2004]. In contrast to prior work, SP uses an adaptive algorithm that dynamically tunes itself to capture patterns in either dimension—*absolute* (when patterns among the values are predominant) or *differential* (when stride patterns among the values are predominant).

The main contributions of this paper are:

- Establish the concept of the spectral prefetcher: a prefetching mechanism that accurately identifies the patterns by dynamically adjusting to their frequency. This scheme overcomes the limitations of correlation-based prefetching, which follows strict value locality [Lipasti et al. 1996] by recording every miss instance for prediction. In contrast, the proposed scheme records only the repeating patterns and issues timely prefetches.
- Introduce an adaptive mechanism that guides the spectral prefetcher along two dimensions—*absolute* (value locality) and *differential* (stride value locality)—where it switches dynamically between the two modes whenever either is failing to acquire the pattern within the cache miss sequence. In differential mode, the spectral prefetcher has the ability to mask compulsory misses.

Using a cycle-accurate simulation of an aggressive out-of-order superscalar processor, we show that the spectral prefetcher with 1 MB of on-chip implementation speeds up memory intensive benchmarks by 1.59, on average, and, at best, by 2.10. This outperforms the previous proposal of tag correlating prefetchers [Hu et al. 2003], when given 2 MB of storage area (twice the size of SP), by 39%. When the spectral prefetcher is compared with a larger L2 cache with approximately equal storage area (i.e., 3 MB) it outperforms the larger L2 cache by 31%.

The rest of the paper is organized as follows. Section 2 describes the framework used to examine the feasibility of the spectral prefetcher, as well as discusses the attributes of an ideal prefetcher, specifically prefetching from main memory. Section 3 shows, with the help of autocorrelation, that tags for a given TCzone exhibit locality. We also discuss the frequency of patterns in terms of recurring distance, which forms the basis of the spectral prefetcher. Section 4 details the structure and the operations of the spectral prefetcher. Section 5 presents the sensitivity analysis and identifies individual solutions to key issues. Section 6 combines the results into a single prefetching architecture and evaluates the new spectral prefetcher. Related work is discussed in Section 7 followed by the conclusion in Section 8.

Table I. Configuration of Simulated Processor

Front end	A 4-way 64KB instruction cache with 64-byte line size, 64K-entry Gshare branch predictor, and a 1024-entry return address stack. A perfect BTB is assumed for providing target addresses.
Execution core	The superscalar core is an 8-wide (fetch/dispatch/issue of 8) machine with 128-entry instruction window and 64-entry issue queue. There are 8 fully symmetric functional units. The pipeline depth is seven stages. The minimum branch miss-prediction penalty is five cycles. The processor frequency is assumed to be 4 GHz.
Caches and buses	The first-level data cache is a 4-way 32KB cache with 64-byte line size. The second-level unified cache is an 8-way 2 MB cache with 64-byte line size and 10-cycle access latency. There are two buses. A first-level bus is shared by first-level data cache and the instruction cache and runs between first-level caches and a unified second-level cache. A second-level bus is between second-level cache and memory. L1/L2 bus is 32-byte wide that operates on 2 GHz frequency with 1 bus-cycle arbitration. L2/MEM bus is 16-byte wide that operates on 1 GHz frequency. The caches are nonblocking and can resolve 32 outstanding misses (32 MSHRs).
Execution latencies	Load to use from first-level cache takes one cycle. INT ALU takes one cycle.
Memory disambiguation	Processor uses Oracle disambiguation. A 128-entry Load-Store queue is incorporated.

2. SIMULATION METHODOLOGY

Results provided in this paper were collected using a modified version of SimpleScalar [Burger and Austin 1999] simulator. The timing simulator models the MIPS R10000 processor and executes only user-level instructions. The simulator is execution-driven and moves down any speculative path until the detection of a fault or branch misprediction. The baseline architecture is an aggressive out-of-order superscalar processor; it has a large window of execution, a large branch predictor, as well as large associative caches. The simulator has an oracle load-store disambiguation policy that causes loads to be dependent only on stores that write to the same memory location. Because contention can have significant influence on performance, we have rewritten the memory interface in SimpleScalar to model cache-hierarchy, bus occupancy, bus protocol, limited bandwidth, and main memory characteristics. The random access latency and row cycle time of memory is assumed to be 200 cycles. The simulator does not model internal DRAM operations like page opening, precharges, and refreshes. We assume that the access latency on a realistic memory channel is dominated by bus accesses and data transfers, as shown by Cuppu et al. [2001]. The main processor and memory hierarchy parameters are shown in Table 1.

We evaluate our results using benchmarks from the SPEC2K suite [Henning 2000]. The programs were compiled using full compiler optimizations and for programs we skip 2 billion instructions to avoid cold start effects. We then simulate 1 billion instructions using the reference input set. Figure 1 shows

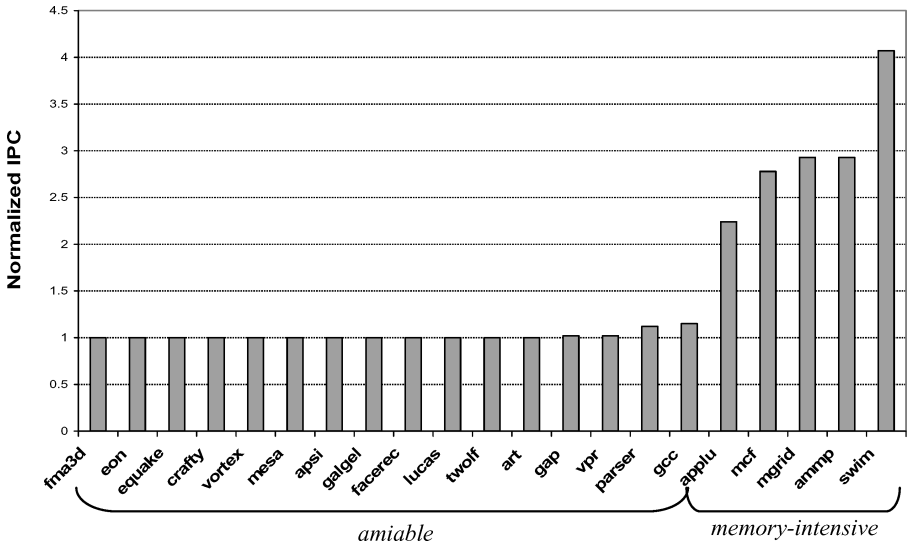


Fig. 1. Potential IPC improvement with Oracle L2 cache for SPEC2K.

the speedup results with an oracle L2 for all benchmarks. The speedup results, normalized against the baseline IPC, show the maximum performance that can be achieved by a prefetcher that is prefetching from memory for the L2 cache. We further divide the benchmarks into two groups: *amiable* and *memory-intensive*. Amiable benchmarks are those that have a small working set size which completely fits in the L2 cache, while the memory-intensive benchmarks are those with extremely large working set sizes, which do not fit in the L2 cache. The two groups are shown in Figure 1. As expected, the performance improvement of the amiable benchmarks with an oracle L2 is negligible, while memory-intensive benchmarks show a promising speedup. In Section 5, we present a sensitivity analysis using memory-intensive benchmarks (*applu, mcf, mgrid, ammp* and *swim*) to identify the key performance issues (like hardware size) for the spectral prefetcher. In Section 6, we evaluate the prefetcher using both the amiable and memory-intensive benchmarks. The prefetcher is required to boost the performance of memory-intensive benchmarks without degrading the performance of the amiable benchmarks.

2.1 Discussion

Recent prefetch research [Hu et al. 2003; Joseph and Grunwald 1999; Nesbit and Smith 2004; Nesbit et al. 2004; Palacharla and Kessler 1994] has advocated prefetching for the L2 cache because modern out-of-order processors usually tolerate L1 cache misses with relatively little performance degradation. However, most of these proposals [Hu et al. 2003; Joseph and Grunwald 1999; Palacharla and Kessler 1994] examine the L1 cache miss stream as the prediction source for prefetching, which are often clustered inside the out-of-order engines. These prefetchers are required to be highly efficient since they may need to analyze multiple references in a short span of time. Moreover, in the absence of regular strided access, the prefetcher would need to be large enough to record every

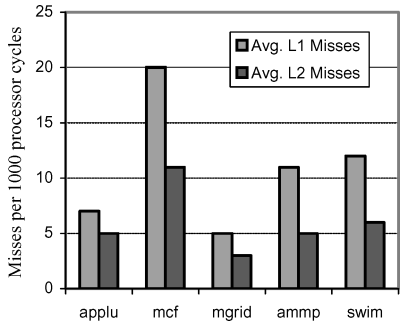


Fig. 2. Average L1 and L2 misses per 1000 cycles.

miss instance between reoccurrences to capture the repeating behavior. The prefetcher proposed in this paper uses the L2 cache miss stream as the prediction source. These L2 misses are presented to the external memory system and occur much less frequently than L1 cache misses. Figure 2 shows the average number of L1 cache and L2 cache misses measured per 1000 processor cycles for the memory-intensive benchmarks. As expected, the number of L1 cache misses is much larger than that of the L2 cache misses.

The apparent disadvantage of training with the L2 miss stream is that the cache hierarchy removes part of the reference pattern leaving underlying randomness in the miss address stream. Consequently, the patterns present in the L2 miss stream are much harder to predict than those of the L1 miss stream. This brings about the following requirements for prefetchers that inspect the L2 miss stream for predictions:

- The prefetcher must be highly accurate and offer timely data to the processor. An inaccurate prefetcher may alter the demand-fetch locality of the cache and can lead to performance loss for workloads that are sensitive to memory contention.
- The prefetcher should have high coverage while using the smallest possible size.

In Section 6, we will show that the spectral prefetcher, which dynamically captures the frequency of the repeating pattern, is highly accurate and offers better performance than other prefetching mechanisms. In order to satisfy the above requirements, SP partitions memory by the lower order bits; these partitions are referred as the TCzones. Since memory is partitioned by the lower order bits, TCzones are strided across the memory, which, in turn, forces a fixed number of TCzones. Caches also partition physical memory in a similar fashion when viewed from the perspective of sets. In our simulation environment, the L2 cache has 4096 sets with 64-byte line size that naturally divides memory into 4096 TCzones, populated by 14-bit tags (assuming 32-bit machine). To divide memory into 512 TCzones from the previous setup, the three most significant bits of the index are concatenated with the tag resulting in 17 bit tags (leaving 9 TCzone index-access bits). An example of how 512 TCzone map into a 4096 entry (set) cache is shown in Figure 3.

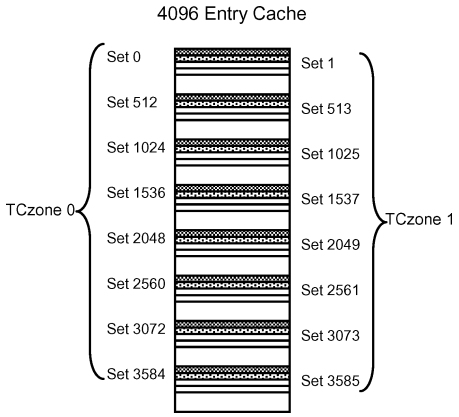


Fig. 3. An example of 512 TCzone.

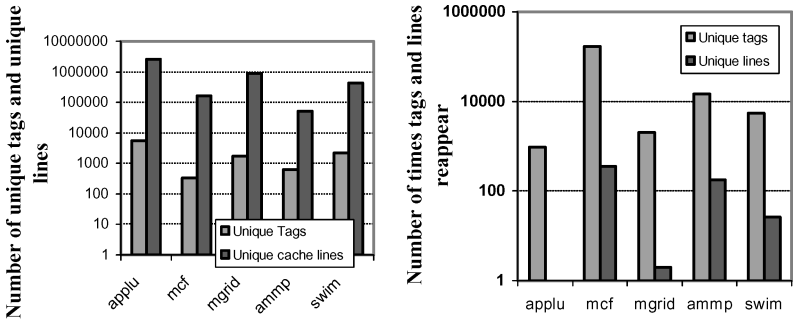


Fig. 4. Number of unique tags and unique lines (a); number of times each tag and line reappears in the miss stream (b). Results are presented in a log scale.

We studied the behavior of the tags in the L2 miss stream and the results of this experiment are shown in Figure 4. The first graph of Figure 4 shows the number of unique tags and the number of unique cache line addresses observed in the L2 miss streams of the memory intensive benchmarks. The second graph shows the number of times a tag and an address recur in the miss streams. As previously mentioned, this is assuming 17 tag bits in order to produce 512 TCzones with 9 index-access bits. The unique number of cache line addresses is much larger than that of unique tags for all the benchmarks. The fewer tags recur more frequently than the addresses, making them a solid basis for prediction. The potential benefits of predicting tags for a specific TCzone are: (1) The lower order bits do not require any tracking as they are implicitly hidden in the TCzone index-access bits and (2) predicting tags produces timely prefetches because tag predictions span pages while applications usually access the same page several times before accessing another. This means data from a prediction into another page of memory will not be required immediately. In the coming sections, we will show that tags for a given TCzone exhibit locality and then present a prefetching mechanism that exploit the locality of the tags.

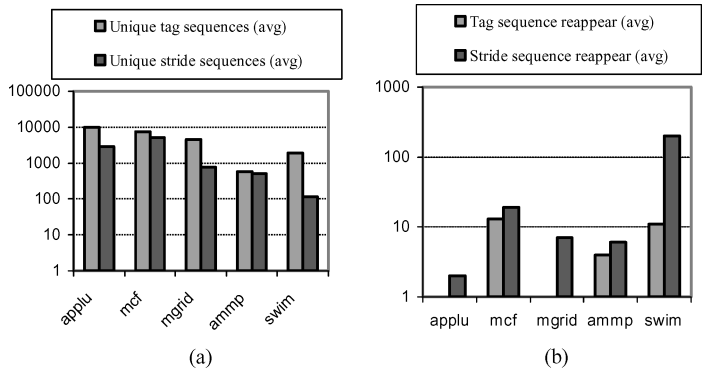


Fig. 5. (a) Average number of unique tag sequences and stride sequences (log scale); (b) average number of times every tag sequence and stride sequence reappear (log scale).

3. LOCALITY OF TAGS IN THE MISS STREAM

In the previous section, we explained how physical memory can be partitioned into TCzones and examined the recurrence behavior of their tags in the L2 miss stream. In this section, we will establish that tags recur repeatedly and follow a pattern within a TCzone, either in absolute or differential form. To illustrate this fact, we measured the repetitiveness of tag sequences in the L2 miss stream by counting the average number of times tag sequences recur within a TCzone. In these experiments, a sequence length of three were searched in the absolute domain, while a sequence length of two was searched in the differential (sequence of strides between the tags) domain for 512 TCzones. This was done for unbiased profiling, as the three consecutive tags in the absolute domain maps into two consecutive strides in the differential domain. The sequences of tags (or strides) were generated by combining the consecutive tags (or strides) together. For example, if *A, B, C, D, E*... were the miss tags arriving in the specific TCzone, tag sequences *ABC, BCD, CDE*... were generated for profiling. Figure 5 shows the results of this experiment for the memory-intensive benchmarks.

Figure 5a shows the average number of unique tag sequences and unique stride sequences observed by every TCzone. Figure 5b shows the average number of times a tag sequence and a stride sequence reappear within a TCzone. For example, the results for the *mcf* benchmark indicate that the unique number of tag sequences is slightly more than the unique number of stride sequences and tag sequences recur a little bit less than the stride sequences. This implies the tag and the stride sequences of the *mcf* benchmark constitute a pattern in the absolute and differential domain, respectively. The results for *ammp* also indicate a similar kind of behavior; recurrence is present in both the absolute and the differential domain. Unlike the above-mentioned benchmarks, *applu* and *mgrid* show recurrence only in the differential domain. Consequently, a predictor that has the ability to inspect patterns in the differential domain will be more effective for these benchmarks. Finally, the results for *swim* indicate a stronger repetitive behavior in the differential domain rather than the absolute domain.

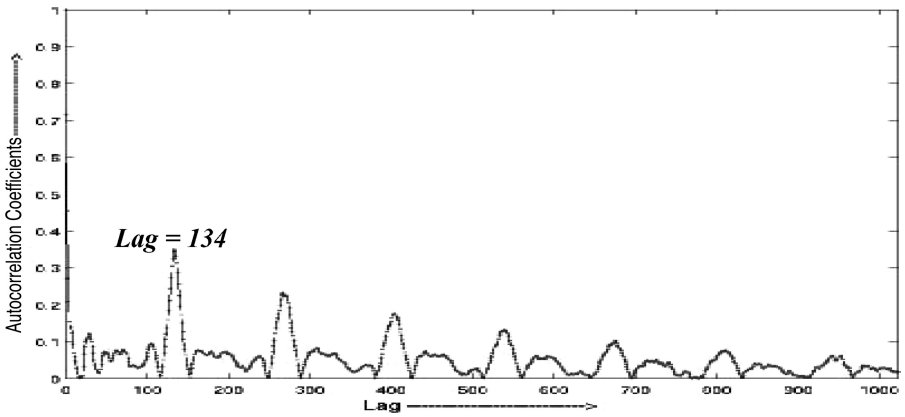


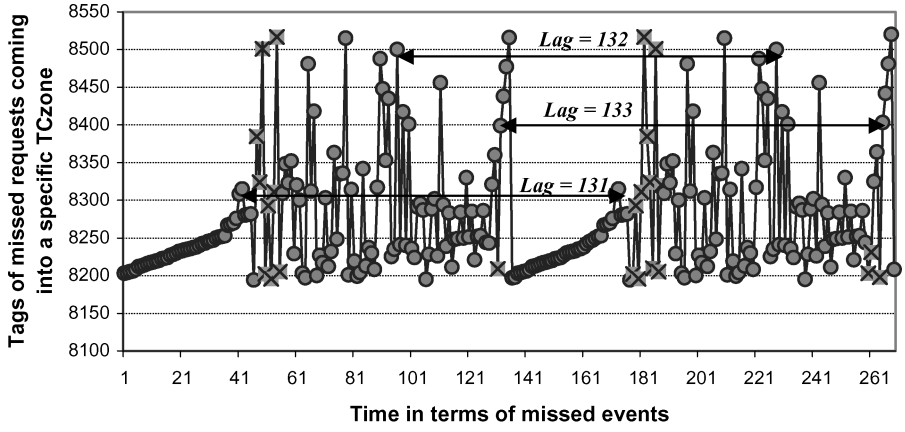
Fig. 6. An autocorrelation plot of the miss tag-sequences of the mcf benchmark arriving into a given TCzone.

Although the results shown in Figure 5 demonstrate that the tag sequences (either in absolute or differential form) exhibit recurring behavior, it does not convey the characteristics of the pattern by which these tag (or stride) sequences reappear in their respective domains. The characteristics of the patterns, such as the recurring distance of the elements present in the pattern and how frequently these distances change, are important for designing a prefetcher. As a result, in the second phase we measured the pattern characteristics with the help of a spectral method called *autocorrelation*, which measures the correlation between values of a data set and is often employed for detecting nonrandomness within the data set. If there is a repeating pattern present in the dataset, autocorrelation can provide information, such as the recurring distance of the elements within the pattern and inferences can be made, such as the likelihood of reappearance. The autocorrelation plot is generated by calculating the correlation coefficients between the values of the same data set at times i and $i + k$, where k is called the *lag*. The formula for calculating correlation coefficient r_k for a given data set Y is:

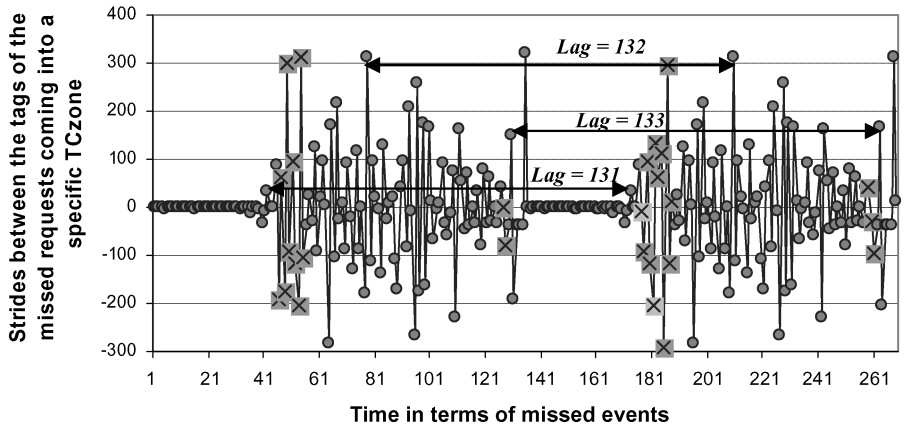
$$r_k = \frac{\sum_{i=1}^{N-k} (Y_i - \bar{Y})(Y_{i+k} - \bar{Y})}{\sum_{i=1}^{N} (Y_i - \bar{Y})^2}$$

Here N is the size and $\bar{Y}$ is the *mean* (average) of the data set. If the data set is random, the autocorrelation coefficients will be near zero for all nonzero time-lag separations. The values of coefficients lie between -1 and $+1$, exclusively. If the coefficient is close to 1 (or -1) at lag k, the data set is said to be correlated for lag k.

An autocorrelation plot for the miss tag sequences of *mcf* was generated and is shown in Figure 6. As can be seen, there is a spike in the autocorrelation plot, suggesting strong correlation in the tag sequences at a lag of *134*, followed by several smaller spikes that are the harmonics of this initial spike. There



(a)



(b)

Fig. 7. (a) Snapshot of the missed tags for a TCzone (mcf). (b) Snapshot of the strides between the missed tags for a TCzone (mcf).

are two important implications of this plot. First, the spike is not isolated; there is a ramp up to and from the lag of *134* on the autocorrelation plot, because of the random noise present in the missed address stream. Second, most of the data points, or tag sequences, are correlated at a lag of $\sim$*134*. This implies that the tags separated by $\sim$*134* miss events are related to each other by some correlating function. If the relationship is perfect, i.e., the tags are reappearing at a distance of $\sim$*134* (miss events), then it can be said that the pattern embedded in the missed tag sequences is repeating at a frequency of $\sim$*1/134*. The above-stated implications can be confirmed in Figure 7, where a snapshot of the miss tags of *mcf* is shown.

The plot in Figure 7a depicts miss tags as *dots* and *crosses*, where the dots represent the tags that repeat in the miss stream; crosses are the random tags present in the miss stream. As can be seen, most of the tags are perfectly related

A (1), **B** (2), **C** (3), **D** (4), $\mathbf{R_1}$ (5), **A**(6), **B**(7), **C**(8), **D**(9) … **(absolute)**
$\mathbf{a}'$(1), $\mathbf{b}'$(2), $\mathbf{c}'$(3), $\mathbf{r}'_1$(4), $\mathbf{r}''_1$(5), $\mathbf{a}'$(6), $\mathbf{b}'$(7), $\mathbf{c}'$(8) … **(differential)**

Fig. 8. Sample miss tag and stride series arriving into a specific TCzone.

and are reappearing by three different frequencies: *1/131*, *1/132*, and *1/133*, thus confirming the results of the autocorrelation plot. The frequency variation among the reappearing tags are due to the presence of random elements being inserted into the pattern midstream, changing the recurring distance of the elements following the random tags present in the pattern. The prefetcher can detect these frequencies by recording and observing the reappearance of the tags in a fixed storage location and use this information to filter out the random elements, only recording the repeating pattern for predictions. There are two potential benefits for predicting only the repeating patterns: the predictions of the prefetcher will be highly accurate and random elements will not evict useful data from the history tables (which are often directly accessed).

Similarly, the behavior of the missed tags in the differential domain can be observed in Figure 7b, where a snapshot of the strides between the missed tags of *mcf* is shown. The plot is chronologically aligned with the plot of Figure 7a with the strides between the tags shown. We have used the same convention (representing repeating strides with *dots* and random elements with *crosses*). As can be seen, the strides also follow a pattern and are reappearing by the same frequencies: *1/131*, *1/132*, and *1/133*. There are two important differences between the plots shown in the Figure 7a and Figure 7b. First, the number of random elements increases in the differential domain. In order to understand this phenomenon, an example is shown in Figure 8, where $A, B, C \ldots$ are the missed tags arriving into a given TCzone and $a', b', c' \ldots$ are the corresponding strides between the tags. Figure 8 shows that the example pattern is following a frequency of *1/5* in both the absolute and the differential domain and the random element R_1 in the absolute domain gets mapped into r'_1 and r''_1 in the differential domain. Thus, the increase in the number of random elements for the differential domain can be formulated as:

$$k^*(n_1 + n_2 + n_3 \ldots) + k,$$

where k is the number of bursts of random elements present in the (absolute) miss stream and $n_1, n_2, n_3 \ldots$ are the sizes of the bursts, respectively. The implication of this phenomenon is that in the presence of random elements, a prefetcher following strict value locality in the differential domain will have lower prediction accuracy and lower coverage than the prefetcher following strict value locality in the absolute domain.

Second, linearly increasing relationships in the absolute domain are converted into repeating patterns in the differential domain. For example, Figure 7a and 7b show that the linearly increasing line in the absolute domain pattern is converted into a sequence of same stride values in the differential domain. When a linearly increasing part of the original pattern in the absolute domain is converted into a pattern in the differential domain, we say a high-frequency component has been introduced. For example, the actual frequency

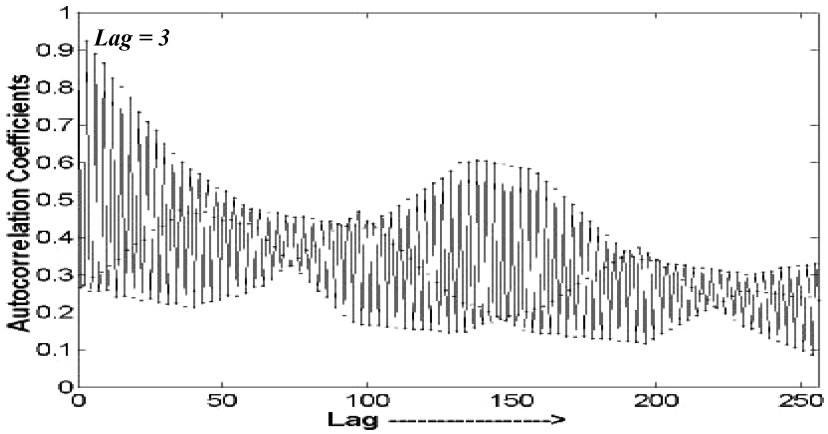


Fig. 9. Autocorrelation plot for the miss tag-sequences of the *mgrid* benchmark.

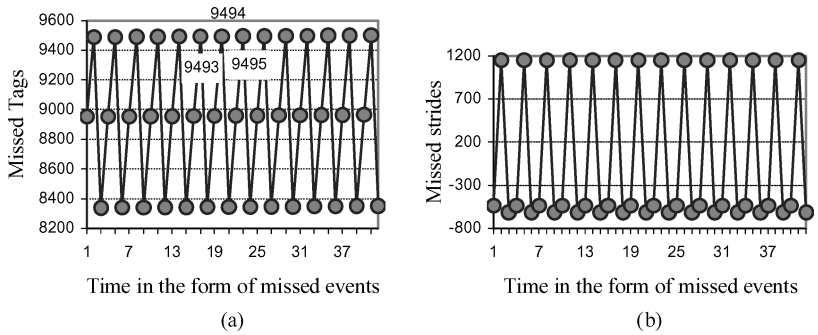


Fig. 10. (a) A snapshot of missed tags observed for a given TCzone (mgrid). (b) A snapshot of the strides between the missed tags observed for a given TCzone (mgrid).

of the patterns in Figures 7a and 7b is $\sim 1/134$, but in the differential domain a high-frequency component is introduced that reappears by a distance of ~ 1. An apparent disadvantage of the high-frequency components is that it will make the job of the prefetcher more difficult in judging the actual frequency of the pattern; only a part of the pattern will be detected if the prefetcher misjudges the high-frequency component to be the actual frequency of the pattern.

Although, the frequency of the pattern gets disturbed in the differential domain for *mcf*, there are applications such as *applu* and *mgrid*, which exhibit recurrence only in the differential domain, as shown in Figure 5. In order to evaluate the behavior of the miss tag sequences for these applications, an autocorrelation plot for the miss tag sequences of *mgrid* was generated and is shown in Figure 9. As can be seen, there is a strong correlation in the tag sequences at lag of 3. This further implies that the tags separated by a distance of 3 in the miss stream are related to each other. For example, a snapshot of the miss tags of *mgrid* exhibit the linearly increasing relationship: $A, B, C, A + 1, B + 1, C + 1, A + 2, B + 2, C + 2 \ldots$, as shown in Figure 10a. Here $A, B, C \ldots$ are the miss tags that arrive into a specific TCzone. There is no perfect relationship between

Table II. Characteristics of the Repeating Patterns for Memory-Intensive Benchmarks

Benchmarks	Patterns Observed Either in Absolute, Differential or Both the Domains	Presence of Randomness in the Access Behavior	Frequency of the Repeating Pattern in Absolute Domain	Frequency of the Repeating Pattern in Differntial Domain
applu	differential	None	None	1/3 to 1/7
mcf	absolute and differential	Yes	1/47 to 1/313	1 to 1/16, 1/47 to 1/313
mgrid	differential	None	None	1 to 1/21
ammp	absolute and differential	Yes	1/9 to 1/202	1 to 1/202
swim	absolute and differential	Minimal	1/762 to 1/1935	1 to 1/12, 1/762 to 1/1935

the miss tags in the absolute domain, while they constitute a pattern in the differential domain. As shown in Figure 10b, a snapshot of the strides between the miss tags of *mgrid*, the pattern reappears by the distance of 3 (*1/3* frequency) in the differential domain. These observations lead to an important point—the prefetcher should monitor both the absolute and differential domain and select the domain that can increase the effectiveness of prefetching.

In addition to the characteristics of the patterns, autocorrelation also provides a rough estimate of the space requirements for detecting the pattern. The frequency of a pattern is itself an answer to the size; if the frequency is $1/x$ than we need at least an x entry history table to detect the pattern. We measured the frequency of the repeating patterns, within a TCzone, for all the memory-intensive benchmarks by searching among the miss tag series, with the help of autocorrelation, in both the absolute and differential domain for 512 TCzones. The results of this experiment are shown in Table II. The second column of the table indicates that patterns were observed either in absolute, differential, or both domains, while the third column shows the presence of randomness in the missed access behavior. The fourth and fifth columns show the frequency range of the patterns observed in the absolute and differential domains, respectively. It is to be noted that the profiling results, with the help of autocorrelation, do not provide any information about the effectiveness of the prefetcher working in the absolute only, differential only, or adaptive (absolute and differential) mode. Autocorrelation only provides feedback about the characteristics of the missed address behavior of the applications. For example, in *mcf*, patterns were seen in both domains and the frequency of the pattern ranged from 1/47 to 1/313. It was further observed that in the differential domain high-frequency pattern emerged with a reappearing distance between 1 and 16. This means that a prefetcher with a history table size of 313 entries can capture a pattern present in the miss access behavior of *mcf*. These results have a great impact on the space requirement of the history table. To illustrate this issue, let us consider two extremes. At one extreme, consider if the prefetcher detects only the high-frequency components introduced in the differential domain. This will greatly reduce the space requirements as these components reappear by the distances of *1* to *21*, for all benchmarks. Unfortunately, these components are often embedded within the original pattern for benchmarks, like *mcf*, and will result in detection of only a part of the actual pattern, as previously mentioned. At the other extreme, consider if the prefetcher detects the pattern only in the

absolute domain. This will lead to a larger capacity requirement as the patterns reappear by the distances of *9* to *1935* and will favor only the applications that show recurrence in the absolute domain.

We have demonstrated that the tag sequences, within a TCzone, follow patterns either in absolute, differential, or both the domains. In this section, we also discussed the frequency of the repeating pattern in terms of the recurring distance of the tag (or stride) in the miss stream, which forms the basis of the prefetcher, explained in the next section.

4. PREFETCHER IMPLEMENTATION

In the last section we showed that the tags or the strides, within a TCzone, follow a pattern in the missed address stream of the L2 cache. It was also shown that the individual elements (tags or strides) of the pattern arrive with a finite frequency. In this section, we present a prefetcher that filters out the random noise from the miss stream and records only the repeating pattern by adjusting to its frequency—thus the name "*spectral prefetcher*" (SP). We start by comparing SP with tag-correlating prefetcher [Hu et al. 2003] (TCP) and then, with the help of an example, we describe the operations of the spectral prefetcher.

TCP was selected for comparison as it also divides memory in a strided fashion (TCzones) and predicts the pattern of tags for a given cache set. Figures 11a and b depicts the structure of TCP and SP, respectively. TCP is a two-level correlating prefetcher where the first-level table, called the tag history table (THT), contains the last k missed tags of the same cache set. These tags are combined together with the current miss tag and the cache set to access the second-level table called the pattern history table (PHT). The PHT provides the next predicted tag, which is combined with the cache set to generate the predicted address. Similarly, SP partitions the physical memory into TCzones and detects tag patterns within each zone. It allocates an *analyzer* and *correlator* for each zone, which are indexed by the index-access bits of the TCzone. The function of the analyzer is to detect the pattern either in the absolute or differential domain and to update the correlator, which provides future predictions for prefetching. TCP and SP have much in common, but differ in the following ways:

- TCP follows strict value locality by recording every miss as a potential candidate for future prediction, while SP attempts to predict only the repeating patterns present in the miss stream by tracking the arrival records of the missed tags.
- Unlike TCP, which attempts to catch pattern only in the absolute domain, SP can switch dynamically between the absolute or the differential domain whenever either is failing to acquire the pattern within the cache miss sequence.

SP is not an extension of TCP; it can be used for full cache-line address prediction when memory is not partitioned into TCzones. In this case a single, large analyzer–correlator pair will be sufficient for pattern detection and prediction.

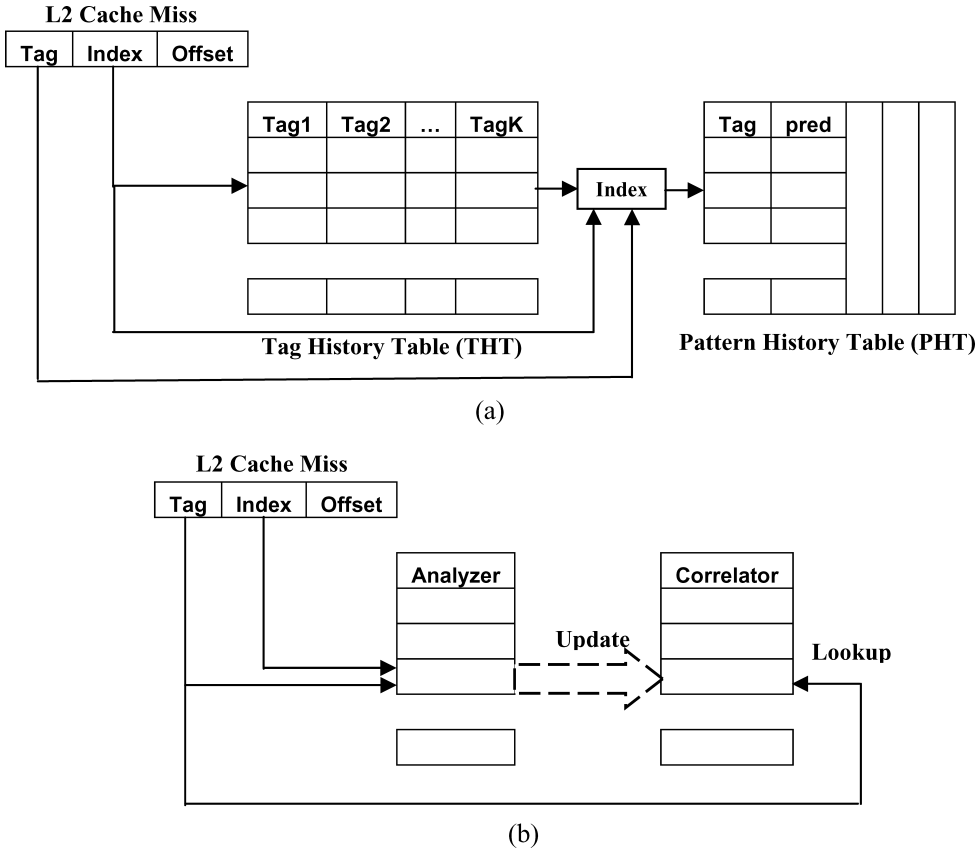


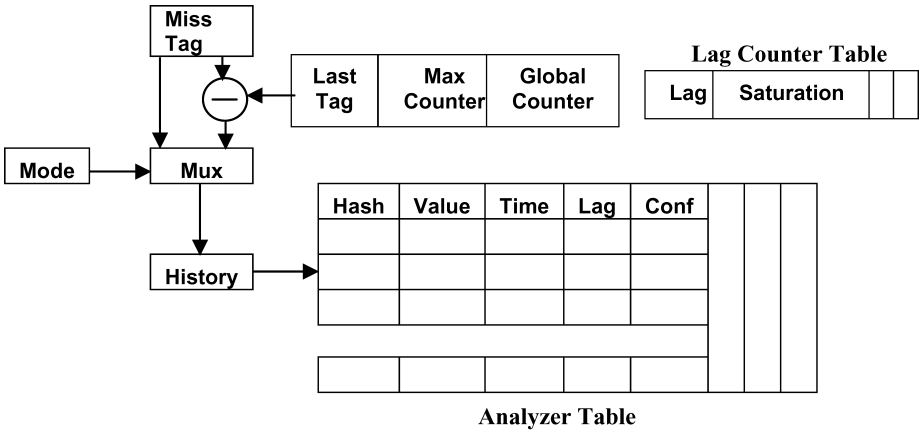
Fig. 11. (a) Structure of tag correlating prefetcher (TCP); (b) structure of spectral prefetcher (SP).

4.1 Operations of the Spectral Prefetcher

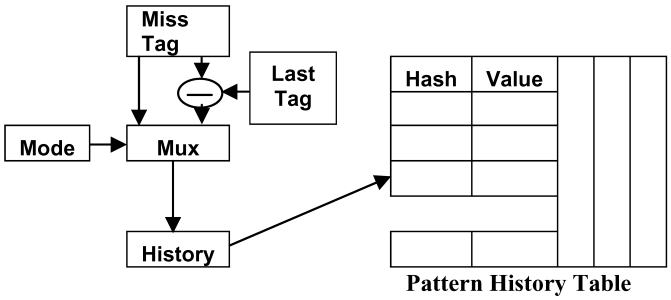
The operations of SP consist of three basic functions: *analysis*, *update,* and *lookup*. In the analysis phase, the analyzer tries to detect patterns by tracking the arrival records of the missed tags (or strides) within the TCzone, while in update phase it passes the pattern to the correlator. The lookup operation is performed by the correlator in order to predict a prefetch address based upon the knowledge of past tag sequences that arrived into a specific TCzone. Figures 12a and b depicts the structure of the analyzer and the correlator, respectively.

The components of the analyzer and the correlator are defined as:

- ***Max counter*** (MC): controls the sample size the analyzer can observe for pattern detection. This value is updated dynamically to allow transition from the analysis to the update state to take place sooner than a static, high value for MC would. This, in turn, allows the pattern to be passed to the correlator sooner.
- ***Global counter*** (GC): maintains the number of tags the analyzer has observed while detecting the pattern. When GC becomes equal to MC, thr analyzer switches from the analysis to the update phase.



(a)



(b)

Fig. 12. (a) Structure of the analyzer; (b) structure of the correlator.

- ***Mode***: A flag that tells the analyzer or the correlator whether they are in absolute or differential mode. When the analyzer does not find any pattern and reaches the update state, control logic flips this bit.
- ***History***: This entry is present in both the analyzer and correlator. It maintains the history of the last k tags or strides observed in the miss stream. It is used for indexing the analyzer table (analyzer) and the pattern history table (correlator).
- ***Last tag*** (LT): This entry is also present in both the analyzer and correlator. It maintains the last observed tag and is used in the differential mode to calculate the stride.
- ***Lag counter table*** (LCT): This is a k-entry fully-associative cache-like structure in the analyzer. The function of LCT is to maintain a record of the frequencies with which the elements of the patterns are repeating in the miss stream. Each entry of LCT has two fields: *lag* and *saturation counter*. The first field maintains a record of the recurring distance (or lag) by which the tags (or strides) are arriving in the miss stream. The second field maintains the number of tags (or strides) observed in the miss stream associated with

A (1), **B** (2), $\mathbf{R_1}$(3), **C** (4), **D**(5), **E** (6), **F** (7), $\mathbf{R_2}$ (8), **A** (9), **B** (10), **C** (11), **D**(12), **E** (13), **F** (14), **A** (15), $\mathbf{R_3}$ (16), **B** (17), **C** (18), **D** (19) …

Fig. 13. Sample miss tag series arriving into a specific TCzone.

that lag. If the saturation counter becomes equal to the threshold value, the controlling logic assumes a pattern is detected and updates MC to analyze only x more entries, where x is the *current lag*, in order to pass the pattern to the correlator sooner. For example, if the saturation counter becomes equal to threshold for lag x, the controlling logic will update the MC to analyze only $GC + x$ entries. Once the MC has been dynamically updated, it is held static until the next analysis.

- ***Analyzer table***: This table is present in the analyzer to maintain arrival records of the tag (or stride) observed in the miss stream. The index of the table is generated by the tag (absolute) or the stride sequence (differential) present in the history. The first field stores the hashed history, while the second field stores the tag (or stride). The record of the recurring distances is kept in the *lag* field, while the *time* field maintains the last time the corresponding tag (or stride) was present in the miss stream. The *conf* bit maintains whether the corresponding entry is a part of the pattern or not.
- ***Pattern history table* (PHT)**: This table is present in the correlator for storing the tag (or stride) correlation pairs that are passed by the analyzer in the update phase. The index of the PHT is formed by the tag (absolute) or the stride sequence (differential) present in the history. The first field stores the hash history as updated by the analyzer, while the second field maintains the next tag (or stride) for prediction.

Now that the components of analyzer and correlator have been described, we discuss how these components detect and predict the pattern embedded in the miss tag series. Assume that the miss tag series that arrives into a specific TCzone is shown in Figure 13. In this example, different tags are identified by different letters. As can be seen, the pattern consists of elements: *A, B, C, D, E,* and F that reappear by two different frequencies: $1/7$ and $1/8$. The random elements in the miss tag series are represented as: R_1, R_2, *and* R_3. Using this miss tag series the operation of SP are described as follows:

- **Analysis**: Assume, in the beginning, that all the components of analyzer are in the reset condition, except the max counter (MC), which is set to *255*. The analyzer is further assumed to be in absolute mode and the threshold value for the saturation counter of the LCT is *1*. The condition of the analyzer, after observing the first *3* miss tags, is shown in Figure 14a. Since there was no entry for the hash history of *AB* in the analyzer table, a new entry was allocated and the value field was updated by the miss tag R_1. The value of GC was passed to the time field, while the lag and conf bits were set to zero. The last tag (LT) and global counter (GC) were then updated to show that the immediate past tag is R_1 and the number of tags observed is 3, respectively. The state of the analyzer after miss tag *C* arrives for a second

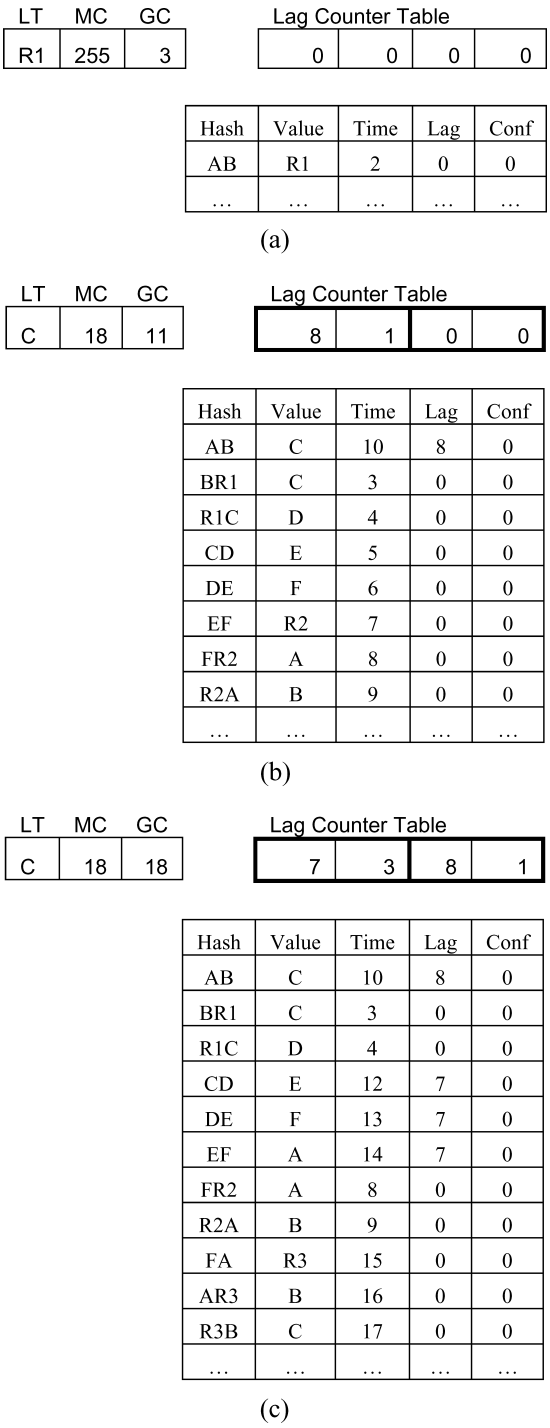


Fig. 14. An example of analysis operation of the spectral prefetcher.

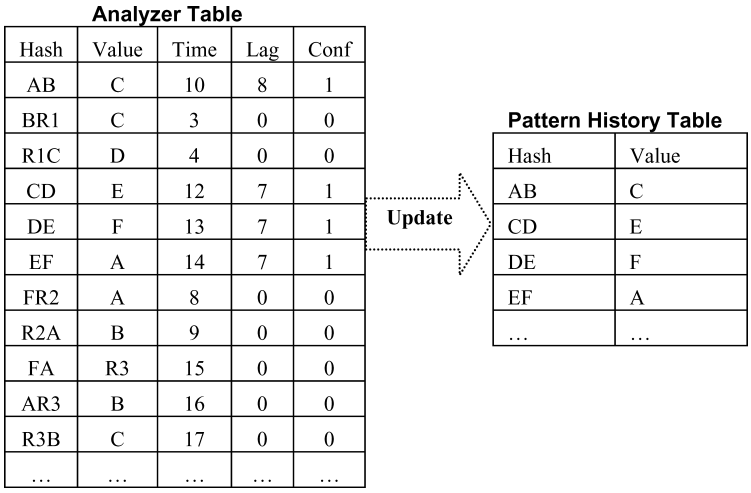


Fig. 15. An example of update operation of the spectral prefetcher.

time is shown in Figure 14b. The values of history and GC are *AB* and 11, respectively. As there was an entry present for *AB* in the analyzer table, the control logic will detect reappearance and update the value field by the miss tag *C*. The lag field of the entry will be updated by the recurring distance, which is "GC – time" or *8* (not *9* since GC the state of prior to increment is used). In addition, an entry is also allocated in the LCT to track the recurring distance observed in the miss stream. As the saturation counter in the LCT entry becomes equal to the threshold value, the controlling logic assumes that the pattern is detected by updating the MC from *255* to *18*. This reduces the sample space for analysis. The final stage of analysis is shown in Figure 14c, where GC reaches the value of *18* and becomes equal to the MC. In the mean time, the analyzer detected the reappearance of histories: *CD*, *DE*, and *EF*, which reappear at a distance of *7* in the miss stream.

- **Update**: In the update stage, the analyzer passes the pattern to the correlator. The control logic, when in the update state, searches the LCT for entries whose saturation counter is equal to the threshold value. For each entry that satisfies this condition, the logic searches the analyzer table for tags, whose lag matches that of the corresponding LCT lag (or the recurring distance), and sets the conf bit of these analyzer table entries. Finally, the entries whose *conf* bit has been set are passed to the PHT of the correlator. In addition, the *mode* of the analyzer is also passed to the correlator. This is done in order to inform the correlator that it is updated either by the pattern of tags or by the pattern of strides. At the other extreme, if no LCT entry matches the threshold value, the control logic assumes that no pattern was found and changes the *mode* from absolute to differential (or vice versa). At the end of the update stage, the components of the analyzer, such as – LT, GC, LCT, and analyzer table, are initialized (reset) and MC is set to its maximum value. This is done so the analyzer can once again start the operation of analysis. For our running example, the update stage is shown in Figure 15. Here the

analyzer has detected a pattern whose elements arrive with frequencies of *1/7* and *1/8*.

- **Lookup**: In this operation, the correlator calculates a prefetch address based upon the knowledge of the immediate past tag (or stride) sequence present in the history entry. The value field in absolute mode holds the next tag and in differential mode holds the next stride. Here, from the PHT, the entry tagged with the current history is selected and its value field used to predict the next tag. If, in differential mode, the value field is added to the missed tag to generate the next tag, as shown in Figure 12b. Finally, the next tag is combined with the index-access bits of the TCzone to form the complete cache line address and, subsequently, a prefetch to this address is issued.

An apparent disadvantage of the update phase is the latency associated with searching the analyzer table for patterns. This latency, however, is mitigated by the fact that an L2 cache miss takes hundreds of cycles to resolve, which can stall the processor, creating a window to work within. Having studied the structure and the operations of the spectral prefetcher, in the next section we present the individual solutions to the key issues that affect the performance of the spectral prefetcher.

5. SENSITIVITY ANALYSIS

In this section, we analyze in detail three aspects that affect the performance of the SP: the selection of the adaptive approach that enables SP to monitor both the absolute and the differential domain for detecting patterns, the size of the analyzer table present in the analyzer, and the size of the PHT present in the correlator. Sensitivity analysis of the SP is important, since not all the memory-intensive applications exhibit recurrence in both domains. Moreover, the sizes of the patterns, either in absolute or differential domain, are not similar for all applications.

To understand the sensitivity to performance of the individual key aspects, we conducted the following experiments:

- First, simulation results are presented for an infinite-sized SP that monitors only the absolute or the differential domain for detecting patterns. This experiment was conducted to establish the importance of the adaptive approach over an SP that only captures either the pattern of absolute values or the pattern of strides among the absolute values.
- Second, simulation results are presented where the size of the analyzer table is varied while the size of PHT is idealized. This is varied to determine the optimal analyzer table size to handle the variety of applications.
- Finally, simulation results are presented where the size of the PHT is varied for a specific-sized analyzer, selected from the second experiment.

For the experiments presented in the current and the following sections, we assume SP inspects the L2 miss stream and prefetch directly into the L2 cache. If a prefetch is issued to memory, the L2 cache is probed to ensure that the prefetch address is not present in the cache. The prefetch requests share

Table III. Characteristics of the Repeating Patterns for Memory-Intensive Benchmarks

Components	Configurations of the Components
Lag counter table (LCT)	A 4-way fully associative LCT is used for intercepting the frequencies with which the tags arrive in the miss stream
Max counter (MC)	In the beginning of the analysis stage, MC is made equal to the number of entries the analysis table can hold for analysis. For example, if the configuration of analysis table is j entry with k associativity, MC at the beginning of the analysis phase will be $j * k$.
History	This entry stores two prior observed tags in the miss stream (tag_n, tag_{n+1}).The hash function for indexing is: (3* tag_n) + (7* tag_{n+1})

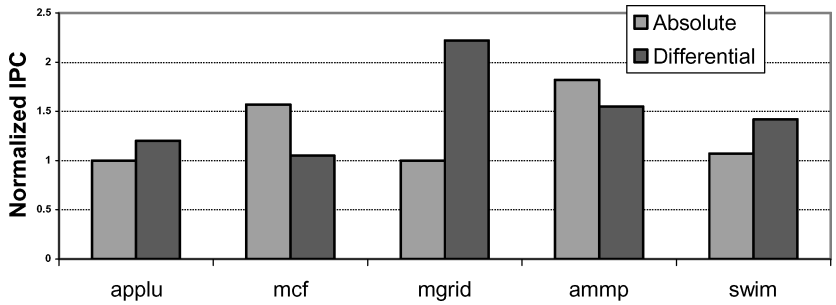


Fig. 16. Performance results of absolute and differential spectral prefetcher.

the L2 miss status handling register (MSHR) and are allowed to access the memory bus only when a free MSHR is available. To prevent the modification of the natural L2 demand miss stream by the prefetched lines, a one-bit prefetch flag is added to the cache lines. This flag is set when the prefetch line is written to the L2 cache. Whenever a cache access hits the prefetched line, the flag is cleared and the address is treated like a miss and is sent to the prefetching hardware. This approach of maintaining the L2 miss stream was first proposed by Nesbit et al. [2004]. The configurations of all the other components of the SP – LCT, MC, and the history are shown in the Table III. The threshold value of the saturation counter is chosen as 3 with memory divided into 512 TCzones. Unless stated otherwise, we assume this environment when presenting the results of SP.

5.1 Impact of Absolute or Differential-Only Mode of the Spectral Prefetcher

Figure 16 shows our first, and perhaps the most important experiment, where the results of SP in only absolute and differential mode are presented for all the memory-intensive benchmarks. As mentioned earlier, an ideal SP with infinite-sized analyzer table (2048 set, 4-way) and PHTs (2048 set, 4-way) were used when conducting these experiments. The results of this experiment confirm that applications exhibit better locality in one domain over the other, as neither approach shows speedup across all benchmarks.

There is a wide variety in the access behavior of the applications. For example, the miss tag values of *applu* and *mgrid* follow a continually increasing

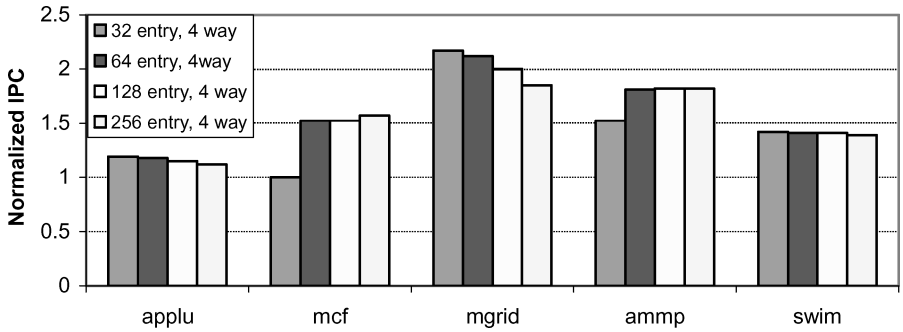


Fig. 17. Performance results of spectral prefetcher when the analyzer table is varied for memory-intensive benchmarks.

order and have minimal repeatability in the absolute domain. However, when observed in the differential domain, these applications exhibit repeating patterns, which are easily detected by the differential SP. On the other hand, *mcf* and *ammp* show promising speedup in absolute mode over differential mode. This is due to the introduction of high-frequency components in differential mode, discussed earlier in Section 3. In absolute mode, false high-frequency components do not exist, so the frequencies of the pattern are not misjudged. Since *ammp* does not introduce as many false high-frequency components as *mcf*, the speedup difference between the modes for *ammp* is not as dramatic.

Conversely, the high-frequency components introduced in the differential domain play an important role in boosting the performance of the *swim* benchmark. As mentioned in Section 3, *swim* shows repeatability in both the domains, but the size of the pattern in the absolute domain is so large that even an unrealistically large SP can not capture the entire sequence. The higher-frequency differential pattern, however, requires-less space, as discussed in Section 3.

These results motivated us to design an adaptive SP, so that it can monitor both the absolute and differential domain to select the patterns from either of the domains, rather than relying on any one mode alone.

5.2 Impact of Varying the Size of Analysis and Pattern History Table

In this subsection, we varied the sizes of the analyzer and the pattern history table to find the optimal size for the general case. These tables, as proposed, are directly accessed using an index value. As a result, if they are not large enough, the prefetcher will have difficulty in detecting and predicting the patterns. Figure 17 shows the impact of the analyzer table size (32 set, 4-way to 256 set, 4-way) for an adaptive SP. For this experiment, we chose an infinite-sized pattern history table (2048 set, 4-way).

As can be seen, *applu*, *mgrid,* and *swim* show counterintuitive behavior and their performance decreases with the increasing size of the analyzer table. This occurs because SP favors the absolute mode initially and all these applications show performance gain only in the differential mode. Moreover, the decision to transition from the absolute to differential mode particularly depends upon the size of the analyzer table, since the value of MC is initialized to the number of

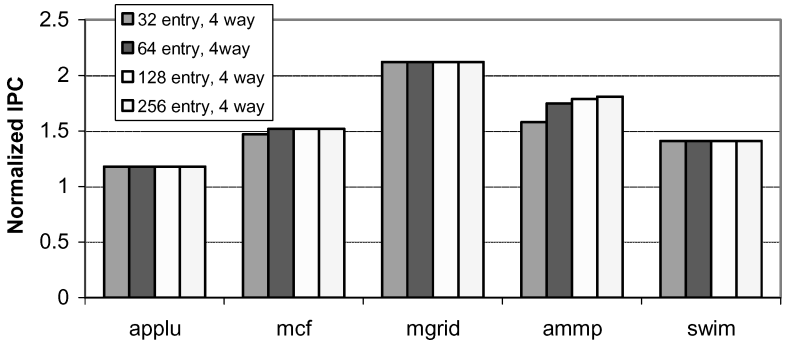


Fig. 18. Performance results of spectral prefetcher when the pattern history table is varied for memory-intensive benchmarks.

entries that the analyzer table can hold for analysis. As a result, there exists a switching delay, which increases with the size of the analyzer table, and is, thus, responsible for this unusual behavior. On the other hand, the results for *mcf* and *ammp* are straightforward; the performance increases with the size of the analyzer table. For a 32 set, 4-way associative analyzer table, *mcf* and *ammp* show negligible performance gains when compared to larger tables for the same benchmarks. The reason for this is smaller tables are not sufficiently big to contain the entire, large absolute mode pattern (which both these benchmarks favor). Subsequently, the analyzer switches from absolute to differential mode (and vice-versa) more frequently, which further deteriorates the performance gain for these applications.

It can be further observed that a 64 set, 4-way associative analyzer table captures almost all of the patterns for all the benchmarks. We believe that this configuration of the analyzer table is sufficient for detecting pattern among all the memory-intensive benchmarks.

Similarly, Figure 18 shows the impact of varying PHT size (32 set, 4-way to 256 set, 4-way) for a 64 set, 4-way analyzer table. It is worth noting that while there was no increase in performance for the benchmarks that favor differential mode (*applu, mgrid*, and *swim*) because of the potentially smaller pattern sizes, the performance of those that favor absolute mode (*mcf* and *ammp*) increased to some extent. To satisfy a general solution that works well across all benchmarks, a 64 set, 4-way PHT was chosen as the optimal size.

6. EVALUATION OF THE SPECTRAL PREFETCHER

In the prior section, we presented the individual solutions to the key issues that affect the performance of the spectral prefetcher. In this section, we combine these results into a single prefetching architecture and evaluate the new SP effectiveness in improving performance. We compare a 1 MB SP against a TCP with a 2-MB correlation history table. To gauge the best performance of TCP, we present results in absolute (as originally proposed) and differential mode. We also compare the results of SP with a 2-MB L2 against a 3-MB 12-way L2, which

has approximately equal storage cost. This is followed by a brief discussion of the coverage and accuracy of SP.

In accordance with the results presented in Section 5, we use an adaptive SP. The configuration of the analyzer table and the PHT are selected as 64 set 4-way associative. For performance evaluation, we consider a practical implementation of SP, where address prediction by the correlator takes seven cycles. In addition, the latency associated for the analysis of miss tag by the analyzer was also set to seven cycles. The latency of traversing the analyzer table to search for a pattern in the update stage was set to 20 cycles; the analyzer cannot serve any miss tag that arrives in the mean time. The latency associated for passing a correlation pair (hash history and the corresponding next tag or stride) was chosen to be four cycles. Thus, passing 10 correlation pairs from analyzer to the correlator will require 40 cycles and no analysis or prediction will take place during this period. The size of the analyzer and correlator can be calculated using the following formulas:

$$\begin{aligned}\text{Size of analyzer} = {} & \text{number of entries in analyzer table}^* \text{ size of (hash} + \text{value} \\ & + \text{lag} + \text{ time-stamp)} + \text{number of entries in LCT}^* \text{ size of} \\ & \text{(lag} + \text{saturation counter)} + \text{size of GC} + \text{size of MC} \\ \text{Size of correlator} = {} & \text{number of entries in PHT}^* \text{ size of (hash} + \text{value)}\end{aligned}$$

In our simulation environment, we selected the following sizes for the fields of analyzer table and PHT entries: 6-bit hash, 14-bit value, 8-bit lag, and 10-bit time-stamp, respectively. As mentioned in Table III, we selected a 4-way LCT and each entry maintains a 1-byte lag and a saturation counter. In addition, the sizes of both GC and MC were selected as 10 bits, making the size of the analyzer and correlator ∼1.19 and ∼0.625 KB, respectively. The number of TCzones was varied to find the optimal size of 512; the total size of SP used in the simulation became ∼929 KB or 1 MB.

We simulated TCP that inspects the L2 miss stream for L2 cache prefetching. The size of the correlating table was chosen as 2 MB with 65536 set and 8-way associativity. Every entry of the correlating table has two fields: a 14-bit missed tag and a 14-bit successor (tag) to that missed tag. The indexing scheme for correlating tables is similar to that used in SP, discussed in Table III. In addition, the hash generated for indexing is combined with the miss index of the cache so that each cache set has its own private space in the correlating table. As previously discussed, our simulation environment uses a 2-MB L2 with 4096 sets, thus making TCP divide memory into 4096 TCzones. Although TCP was originally proposed to capture pattern of tags (absolute), we also present results of TCP capturing pattern of strides among tags (differential). For unbiased evaluation of SP, we further present the results of TCP that divides memory into 512 TCzones.

Figure 19 shows the performance results of SP as compared to the different configurations of TCP. The prefetching schemes in Figure 19 are represented as an acronym followed by a numeral, where the acronym represents the name of the corresponding scheme, while the numeral identifies the number of partitions with which the memory is divided by that scheme. For example, *TCP 4096* represents a tag-correlating prefetcher that divides memory into 4096 TCzones.

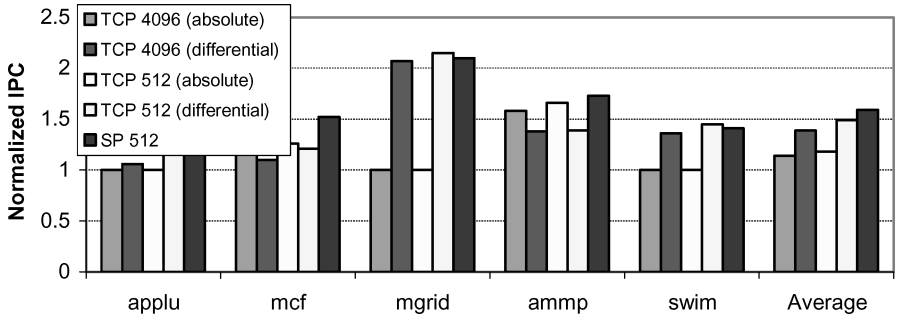


Fig. 19. Performance results of spectral prefetcher vs. different configurations of TCP.

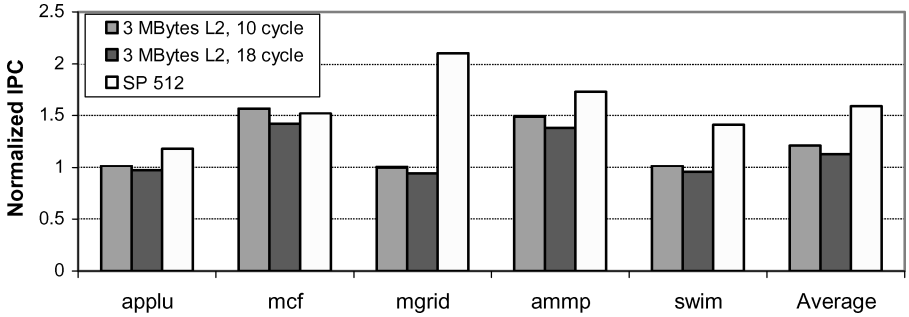


Fig. 20. Performance results of spectral prefetcher vs. 3-MB L2 with 10 and 18 cycles access latency.

In general, SP outperforms all the configurations of TCP. On average, SP achieves a 1.59 performance improvement for all the memory-intensive benchmarks, while TCP 4096 (absolute), TCP 4096 (differential), TCP 512 (absolute), and TCP 512 (differential) shows performance improvement of 1.14, 1.38, 1.18, and 1.49, respectively. SP gives the best performance for the benchmarks that have presence of random elements in their respective miss streams (*mcf* and *ammp*). This is due to the fact that SP has a unique ability to distinguish between the repeating patterns and the random elements, which makes the predictions of SP highly accurate. On the other hand, TCPs follow strict value locality by recording every instance of misses, i.e., both the patterns and the random elements for predictions, and are thus associated with low prediction accuracy. It is also worth noting that for the above indicated benchmarks (*mcf* and *ammp*), TCPs show better performance in the absolute as compared to the differential mode. The reason is that the differential domain introduces additional randomness for applications that already contain random elements (as explained in Section 3), which further decreases the accuracy of the differential TCPs. For benchmarks that favor the differential mode (*applu, mgrid,* and *swim*), TCP 512 (differential) marginally outperforms SP. This happens because SP favors the absolute mode initially and there exist a switching delay in transition from absolute to differential mode, as explained in the previous section.

Figure 20 compares 1-MB SP with a base 2-MB L2 (with access latency of 10 cycles) against a 3-MB L2 (4096 set 12-way). We evaluated both an

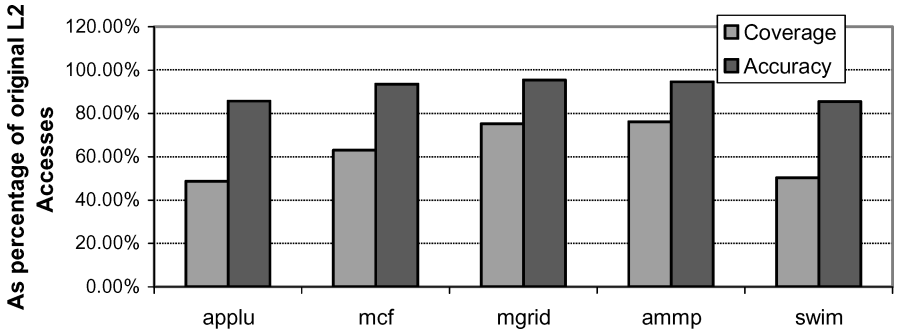


Fig. 21. Utilization of prefetched lines served by SP in terms of coverage and accuracy.

unrealistic (10-cycle access latency) and realistic (18-cycle access latency) implementations of large L2. The results indicate that for most of the benchmarks (*applu, mgrid, ammp,* and *swim),* SP is more cost effective than increasing the size of L2. For the *mcf* benchmark, an unrealistic L2 slightly outperforms SP, because the larger L2 captures a significant fraction of its working set. Conversely, in a realistic environment, SP outperforms a larger L2 for *mcf*, mainly because the out-of-order engine fails to overlap the realistic L2 latency. Moreover, larger L2 implementations show no increase in performance for benchmarks that have minimal repeatability (*applu, mgrid*, and *swim*) in their respective miss stream (as shown in Figure 4).

To fully evaluate SP, accuracy and coverage were also analyzed. The utilization of the prefetched lines that were served by SP is shown in Figure 21. The figure shows the utilization in terms of coverage and prediction accuracy. An ideal prefetcher would have a large coverage with high prediction accuracy. As can be seen, SP provides decent coverage and a 85–95% prediction accuracy across all the memory-intensive benchmarks, further showing the effectiveness of the spectral prefetcher.

Finally, we observed that SP neither hurts nor improves the performance of amiable benchmarks, since their working sets completely fit into the L2 cache.

7. RELATED WORK

Numerous hardware prefetching architectures have been proposed in literature, many of which have relied on capturing specific memory reference patterns. Chen and Baer [1992] investigated data references to detect regular strides in the access pattern for prefetching and proposed stride prefetchers that correlate strides with the PC of memory instruction. Jouppi [1990] proposed stream buffers to improve cache performance for sequential reference stream. Palacharla and Kessler [1994] extended the effectiveness of stream buffers by allocation filtering and nonunit stride-detection mechanisms. Charney and Reeves [1995] were the first to propose a correlation prefetching scheme for L1 miss reference stream. In this scheme, a hardware cache maintains parent–child pair information, where parent corresponds to the first cache line and child corresponds to the cache line accessed right after the first cache line. Joseph and Grunwald [1999] proposed a Markov model for prefetching that maintains

a set of states that are connected with transition arcs denoting transition from one state to another. In the case of data-prefetching, states denote the cache lines and transition arcs denote the probability of transition from one cache line to another. Roth et al. [1998] proposed a prefetching mechanism, for linked data structures that can run-ahead a pointer intensive application to mask the prefetch latency. Lai et al. [2001] were the first to propose hardware-based dead-block predictor based on PC traces of the memory instruction. In this scheme, the prefetcher predicts when the cache line becomes "dead" and what missed cache line will be referenced next by the processor. Cooksey et al. [2002] proposed content-directed data prefetching, which prefetches the connected linked data structure elements by examining the data contents of the missed cache line. Hu et al. [2003] proposed, a TCP that exploits the repeating behavior of the cache line tags for a given cache set to generate a prefetch. In their scheme, the prefetcher is placed between direct-mapped L1 and large associative L2 and prefetches are generated for L2 by looking into the L1 miss stream.

Compiler-based prefetching inserts explicit prefetching directives into the code to fetch data into the cache before the actual access is executed. Mowry et al. [1992] proposed a software solution for scientific applications by accurately predicting the likely missed references and were successful in hiding the memory access latency. Luk and Mowry [1996] and Lipasti et al. [1995] targeted pointer-intensive applications with recursive data structures. While Luk and Mowry proposed a greedy approach for pointer prefetching, Lipasti proposed heuristics that considers pointer passed as argument to the procedures for prefetching.

Ding and Zongh [2003] studied reuse (recurrence) for predicting the cache miss rates of programs. By using recurrence distance, they were able to predict the miss rates for all data inputs on all sizes of the fully associative or limited associative caches. Their work did not investigate the concept of using recurrence for prefetching as the SP does.

8. CONCLUSION

In this paper we proposed and evaluated SP, an adaptive method for prefetching data from main memory. SP divides the memory address space into TCzones and detects the pattern of tags (or strides), within each TCzone by dynamically adjusting to their frequency. The adaptive mechanism of SP dynamically determines whether the pattern of tags or pattern of strides will increase the effectiveness of prefetching and switches accordingly. This scheme overcomes the limitations of correlation-based prefetching, which follows strict value locality, and records only the repeating patterns for predictions.

We used a cycle-accurate aggressive out-of-order simulator that models bus occupancy, bus protocol, and limited bandwidth. Our experimental results show performance improvement of 1.59, on average, and, at best, 2.10 in the memory-intensive benchmarks we studied. Further, we show that SP outperforms the previously proposed scheme, with twice the size of SP, by 39% and a larger L2 cache, and with equivalent storage area by 31%.

REFERENCES

BURGER, D. AND AUSTIN, T. 1999. The Simplescalar Toolset, Version 3.0 Tech. rep., University of Wisconsin, Madison.

CHARNEY, M. J. AND REEVES, A. P. 1995. Generalized Correlation Based Hardware Prefetching, Tech. rep., School of Electrical Engineering, Cornell University.

CHEN, T. F. AND BAER, J. L. 1992. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA. ACM Press, New York.

COOKSEY, T., JOURDAN, S., AND GRUNWALD, D. 2002. A stateless, content-directed data prefetching mechanism. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA. ACM Press, New York.

CUPPU, V., JACOB, B., DAVIS, B., AND MUDGE, T. 2001. High performance DRAMS in workstation environments. *IEEE Transaction on Computers 50*, 11, 1133–1153.

DING, C. AND ZHONG, Y. 2003. Predicting whole program locality through reuse distance stateless analysis. In *Proceedings of International Conference on Programming Language Design and Implementation*. San Diego, CA. ACM Press, New York.

HENNING, J. L. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computers 33*, 7 (July), 28–35.

HU, Z., MARTONOSI, M., AND KAXIRAS, S. 2003. TCP: Tag correlating prefetchers. In *Proceedings of 9th International Symposium on High Performance Computer Architecture*, Anaheim, CA. IEEE Press.

JOSEPH, D. AND GRUNWALD, D. 1999. Prefetching using Markov Predictors. *IEEE Transactions on Computers 48*, 2, 121–133.

JOUPPI, N. P. 1990. Improving direct-mapped cache performance by the addition of the small fully associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, WA. ACM Press/IEEE, New York.

LAI, A. C., FIDE, C., AND FALSAFI, B. 2001. Dead-Block prediction and Dead-Block Correlating Prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg. ACM Press/ IEEE New York.

LIPASTI, M. H., SCHIMIDT, W. J., KUENEL, R., AND ROEDIGER, R. R. 1995. Software prefetching in pointer and call intensive environment. In *Proceedings of the 28th International Symposium on Microarchitecture*, Ann Arbor, MI. ACM Press/IEEE, New York.

LIPASTI, M. H., WILKERSON, C. B., AND CHEN, J. P. 1996. Value locality and load value prediction. In *Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA. ACM Press, New York.

LUK, C. K. AND MOWRY, T. C. 1996. Compiler based prefetching for recursive data structures. In *Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA. ACM Press, New York.

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA. ACM Press, New York.

NESBIT, K. AND SMITH, J. E. 2004. Prefetching with a global history buffer. In *Proceedings of 10th International Symposium on High Performance Computer Architecture*, Madrid, Spain. IEEE.

NESBIT, K., DHODAPKAR, A. S., AND SMITH, J. E. 2004. AC/DC: An adaptive data cache prefetcher. *IEEE PACT 2004*, Antibes Juan-les-Pins, France.

PALACHARLA, S. AND KESSLER, R. E. 1994. Evaluating stream buffers as secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL. ACM Press/IEEE, New York.

ROTH, A., MOSHOVOS, A., AND GURINDER, S. S. 1998. Dependence based prefetching for linked data structures. In *Proceedings of 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA. ACM Press, New York.

Received August 2005; revised November 2005; accepted December 2005