Timing Local Streams: Improving Timeliness in Data Prefetching

Huaiyu Zhu, Yong Chen and Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
{hzhu12,chenyon1,sun}@iit.edu

ABSTRACT

Data prefetching technique is widely used to bridge the growing performance gap between processor and memory. Numerous prefetching techniques have been proposed to exploit data patterns and correlations in the miss address stream. In general, the miss addresses are grouped by some common characteristics, such as program counter or memory region they belong to, into localized streams to improve prefetch accuracy and coverage. However, the existing stream localization technique lacks the timing information of misses. This drawback can lead to a large fraction of untimely prefetches, which in turn limits the effectiveness of prefetching, wastes precious bandwidth and leads to high cache pollution potentially. This paper proposes a novel mechanism named stream timing technique that can largely reduce untimely prefetches and in turn increase the overall performance. Based on the proposed stream timing technique, we extend the conventional stride prefetcher and propose a new stride prefetcher called Time-Aware Stride (TAS) prefetcher. We have carried out extensive simulation experiments to verify the design of the stream timing technique and the TAS prefetcher. The simulation results show that the proposed stream timing technique is promising in reducing untimely prefetches and the IPC improvement of TAS prefetcher outperforms the existing stride prefetcher by 11%.

Categories and Subject Descriptors

B.3 [**Memory Structures**]: Design Styles

General Terms

Design

Keywords

Data Prefetching, Cache Memory, Prefetching Simulation, Prefetching Performance

1. INTRODUCTION

The rapid advance in semiconductor technology allows the processor speed or the aggregate processor speed on chip with multicore/manycore architectures grows fast and steadily. The memory speed or the data load/store performance, on the other hand, has been increasing at a snail's pace for over decades. This trend is predicted to continue in the next decade. This unbalanced performance improvement leads to one of the significant performance bottlenecks in computer architectures known as "memory-wall" problem [25]. Multiple memory hierarchies have been the primary solution to bridging the processor-memory performance gap. However, due to the limited cache capacity and highly associative structure, large amount of off-chip accesses and long data-access latency still spike the performance severely. Hardware-based data prefetching has been widely recognized as a companion technique of memory hierarchy solution to overcome the memory-wall issue [25].

The principle of data prefetching technique is that the prefetcher is able to fetch the data from a lower level memory hierarchy to a higher level closer to the processor *in advance* and *in a timely manner*. This principle decides two critical aspects of a data prefetching strategy, *what to prefetch* and *when to prefetch*. Although extensive existing studies have been focused on the problem of what to prefetch, the other critical issue, when to prefetch, has long been neglected. The ignorance of the timing issue of prefetches can significantly affect the prefetching effectiveness. Large amount of untimely prefetches that do not arrive within a proper time window can result in cache pollution, bandwidth waste, and even a negative impact on overall performance. In general, untimely prefetches can be categorized into two types: *early prefetches* and *late prefetches*. A prefetch is defined to be *late* if the prefetched data are still on the way back to cache when an instruction requests the data. In this case, the late prefetch might not contribute much to the performance even though it is an accurate prefetch. A prefetch is defined to be *early* if the prefetched data are kicked out by other blocks due to the limited cache capacity before such prefetched data are accessed by the processor. Apparently, the early prefetch is not merely useless, but also imposes negative effects by causing cache pollution and waste of bandwidth. It is critical to control the number of untimely prefetches within an acceptable range to lessen the adverse impact and exploit the benefits of data prefetching.

In this paper, we present a *stream timing* technique, a novel mechanism aiming to improve the timeliness of prefetches. The proposed technique is based on *stream local-*

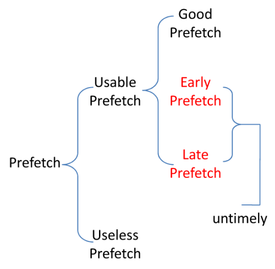

Figure 1: Classification of prefetches.

ization technique, a widely-used technique to classify miss addresses into various streams according to a specific criteria such as from the same instruction (program counter), within the same memory region, etc., to improve the prediction accuracy and prefetch coverage. The basic idea of stream timing technique is to keep the timing information for each stream and chain them according to the time. The time distances among accesses within a stream or across different streams are taken into account to direct prefetching. This approach can improve the timeliness of prefetches and generate prefetch candidates with high confidence. Those problems such as cache pollution and bandwidth waste caused by untimely prefetches can be effectively mitigated. We also incorporate this technique with a well-known conventional stride prefetcher and propose a new *Time-Aware Stride (TAS)* data prefetcher.

The rest of this paper is organized as follows. Section 2 describes the background and motivation of this study. Section 3 introduces the concept of stream timing technique and the design of a stream timing prefetcher instance, Time-Aware Stride prefetcher. Section 4 discusses evaluation methodology and simulation results. Section 5 reviews important related works and compares with our work. Finally, Section 6 concludes this study.

2. BACKGROUND AND MOTIVATIONS

2.1 Prefetching Accuracy, Coverage, Timeliness and Stream Localization

There are three commonly used metrics: *accuracy*, *coverage* and *timeliness* to evaluate a prefetching algorithm. prefetching accuracy is defined as the percentage of prefetches accessed before they are evicted from the cache out of the overall prefetches. A high accuracy helps the prefetcher avoid potential cache pollution caused by useless prefetches. The prefetching coverage measures the ratio of *raw misses (misses without prefetch)* reduced by prefetches. It describes the ability of detecting and correctly predicting the future misses by a prefetcher. prefetching timeliness represents the capability of issuing timely prefetches by a prefetcher. In this paper, we consider both late prefetches and early prefetches as untimely prefetches, and try to improve the timeliness by reducing them or converting them into good prefetches. Figure 1 shows our classification of prefetches. prefetches are classified as *useless* if they are not accessed during the whole lifetime of the application. These prefetches do not help reduce misses, but instead, lower the accuracy and lead to problems such as cache pollution and bandwidth consumption. As opposed to useless prefetches, usable prefetches are those accessed by the processor, which include *good*, *late* and *early* prefetches. The *good prefetches* are those issued by a prefetcher and then accessed by the processor later. For late and early prefetches, although they are correct predictions of future misses, they fail to hide the full memory-access latency efficiently since they are not issued at a proper time. If we can find a way to convert untimely prefetches into timely ones, the prefetching performance can be improved significantly.

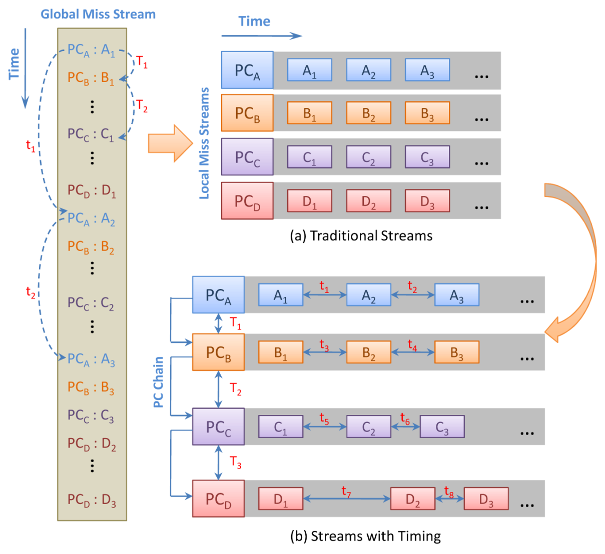

Figure 2: Stream localizing and timing.

The interplay of accuracy, coverage and timeliness are usually complicated. In most cases, a method improving one of them is not able to juggle the other two. To provide high prefetching accuracy, stream localization technique was proposed and is widely used. The rationale of stream localization is that the prefetcher separates the global access stream (full history of memory-access addresses) according to a specific characteristic, and then makes the prediction based on local streams instead of the global one. The criteria for localizing streams can be the instruction address (i.e. program counter, or PC) that issues the access, memory region (memory location the access belongs to) and time period (during which period the access happens). Figure 2(a) shows the localized streams based on PC. Due to the high predictability of data patterns and correlations in local streams, the prefetcher is likely to achieve much better accuracy. However, this strategy limits the prefetching timeliness since the chronological order of the occurrence of accesses is missing in local streams. Thus, it is a challenging task for data prefetcher to generate timely prefetches.

2.2 Motivations

The poor timeliness support found in existing stream localization based prefetchers, as discussed previously, motivates us to propose a new stream timing mechanism to address this knotty problem. The traditional stream localization prefetcher predicts the future address according to the past access history kept in a table. The proposed stream timing mechanism allows the prefetcher to maintain timing information together with addresses so that the time when a particular miss happens is also predictable. This method is based on an important observation that the timing information of data accesses in a local stream exhibits perceivable patterns, i.e. the time intervals between accesses can be also discovered with patterns. For example, in a PC-localized constant-stride prefetcher, the time intervals between adjacent accesses in a certain local stream are most likely to have the same value. This observation has been verified in our simulation experiments and the result shows that the time prediction in our TAS prefetcher can reach an average accuracy around 90% (please see Section 4.2 for details). Figure 2(b) illustrates a sample scenario of stream timing, where t_n represents the time period between $miss_n$ and $miss_{n+1}$. T shows the time intervals among different streams, which is also used to chain local streams. The ability of reconstruction of the chronological order of accesses is critical in guiding timely prefetches. In the following section, we introduce the detailed design of stream timing technique; a stream-timing enabled stride prefetcher and presents its prefetching methodology.

3. DESIGN AND METHODOLOGY

In this section, we first introduce the detailed stream timing technique. Note that the proposed stream timing technique is a general idea aiming to address the timeliness problem in existing prefetchers. After explaining the stream timing idea, we incorporate it with the widely-used stride prefetcher and propose a new Time-Aware Stride (TAS) data prefetcher.

3.1 Stream Timing Technique

As discussed previously, when local streams are chained according to the timing information, an alternative way of selecting prefetch candidates is possible, which is to select prefetch candidates across streams following the chains, instead of selecting candidates only from a single stream. We term the selection of prefetch candidates within a local stream as *depth-first* selection, and the selection of candidates across chained streams as *width-first* selection. Figure 3 demonstrates how these two different prefetch strategies function. Our proposed method records the time when an access is requested, which is used to calculate the time interval between the last access from the same stream and the current one. Meanwhile, the time is regarded as the stream time that indicates the latest access from this stream, and is used to establish the stream chain.

Similar to address prediction, the time is highly predictable with the help of the historical time information. Thus, distinct from an uncertain speculation on when to prefetch, the proposed stream timing technique is able to provide both future access addresses and times. In essence, the proposed stream timing technique converts the traditional *one-dimensional* data prefetching (only considering the history of addresses) to a *two-dimensional* prefetching considering the history of both addresses and times. The basic methodology of taking advantage of timing information operates as follows. If the predicted access will happen too soon, a preset prefetch distance will be performed to avoid late prefetch and we follow the depth to select prefetch candidates. If the time interval between the current access and the predicted access is longer than a preset threshold, which indicates a case of an early prefetch, the prefetcher goes width in order to find timely prefetch candidates within other streams. The general process of a prefetching scheme with stream timing technique can be divided into four steps. Firstly, the prefetcher finds a suitable algorithm to identify data patterns. Secondly, the global stream is localized according to a certain criteria. Thirdly, the local streams are chained to regain the chronicle order. Finally, stream timing can be obtained and prefetches are selected and issued in a timely manner.

The choice of time measures is critical to stream timing technique. CPU cycle sounds an appropriate choice since the global cycle counter has been provided on most recent microprocessors. However, using CPU cycles to represent the time costs too much storage since the time interval between accesses can be extremely large. Other solutions like using an instruction counter, or load/store instruction counter will also waste lots of hardware storage. In this study, we choose a miss counter that ticks upon cache misses to measure the time, i.e. the miss number. It has several merits compared with other choices. First, the miss number is more accurate to represent the time. This is because that a large number of misses can indicate an early prefetch due to frequent cache replacement, whereas a long absolute time cannot if few misses appear in the future. For instance, in a local stream, miss block A triggers prefetch for block B and the history shows the time interval between A and B is 200 misses. In this scenario, the time interval (200) is too long, thus prefetch B will be identified as an early prefetch. It does make sense since the frequent replacement caused by 200 misses is likely to evict B from cache, which suggests that B is prefetched early. The second benefit of using miss counter is that the miss number between accesses is more stable than the actual time due to the program logic. Additionally, the miss counter does not need frequent updates and considerable hardware expense.

We would like to point out that, with the stream timing technique, a new way of carrying out prefetches is possible. The traditional prefetchers are triggered by the occurrence of misses, and we term them as *passive prefetching* since the prefetcher is not able to carry out prefetches unless a miss occurs. The stream timing technique offers a completely different way of issuing prefetches that we term as *active prefetching*. The active prefetching is capable of not only issuing prefetches upon miss events, but also carrying out prefetches proactively upon time events. Apparently, the hardware cost and operation complexity are high for active prefetching. Hence, we only consider passive prefetching in this study in order to maintain low hardware cost.

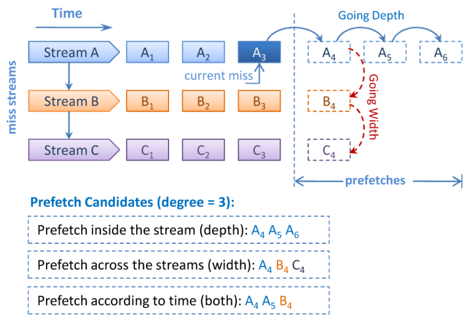
Time
Going Depth
miss streams
Stream A
A_1
A_2
A_3
A_4
A_5
A_6
current miss
Stream B
B_1
B_2
B_3
B_4
Going Width
Stream C
C_1
C_2
C_3
C_4
prefetches
Prefetch Candidates (degree = 3):
Prefetch inside the stream (depth): A_4 A_5 A_6
Prefetch across the streams (width): A_4 B_4 C_4
Prefetch according to time (both): A_4 A_5 B_4

Figure 3: Directions of prefetching.

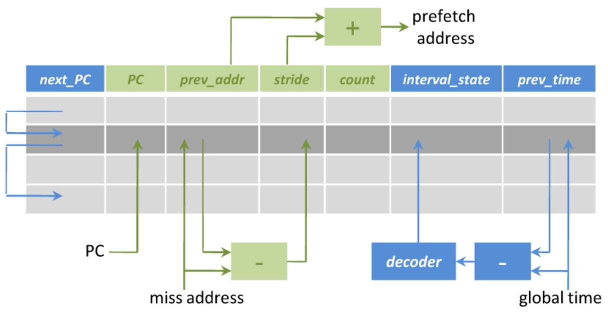

Figure 4: Enhanced RPT for TAS prefetcher.

3.2 Time-Aware Stride Prefetching

We propose a new Time-Aware Stride (TAS) prefetcher by incorporating stream timing technique with the widely-used stride prefetcher. The reason why we choose to incorporate with a stride prefetcher is that the stride access pattern is the mostly common one in practice. In addition, the stride prefetcher has been recognized as the most feasible and effective one in existing processor architectures [8][23].

3.2.1 TAS Design

The Reference Prediction Table (RPT) is widely used in stride prefetchers to keep track of previous reference addresses and associated strides [2]. In the proposed TAS prefetcher, we enhance the RPT table so that the time information of local streams can be stored. The traditional RPT entry includes four fields: PC, prev_addr, stride and state. The PC field represents the address of load/store instructions and is also the index of a local stream. The prev_addr field stores the last address referenced by the instruction specified in PC field. The stride field records the difference between the last two addresses in the local stream and is responsible for predicting the future address together with prev_addr. In addition, a two-bit saturating counter, increasing upon a stride match and decreasing upon a mismatch, can be used in each entry for making a prefetching decision. In the TAS prefetcher, we extend the RPT table with three new fields for each entry as shown in Figure 4:

- prev_time field: the global time when the address in the stream is last referenced
- next_PC field: a pointer linking to a local stream which is closest in time after current one
- interval_state field: a two-bit state indicator denoting time interval length between addresses in the local stream with four possible states: short, medium, long and very long

These new fields in each entry are used to make local streams chained and time-stamped. The stride prefetching works based on the identification of constant strides. We observe that once the stride pattern is found, which means the prefetcher is trained, the time intervals between addresses with stride patterns tend to be in the same range, i.e. either short, or medium, or long or very long. Thus, the time interval between the last two addresses in a local stream can be used to represent the one between any two consecutive addresses. Figure 4 shows that the time interval is derived by subtracting prev_time from current global_time, and it

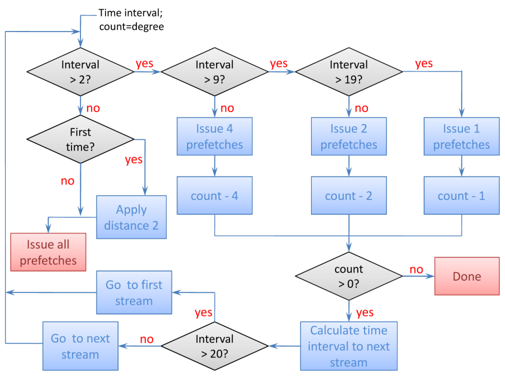

Figure 5: Prefetching algorithm for TAS, degree=8.

is used to determine the interval_state. Compared to the conventional stride prefetcher, the new entries in the table build relations between different local streams by connecting and timing them. Thus, this new TAS stride prefetcher has higher confidence in finding timely and optimal prefetch candidates than existing stride prefetchers.

3.2.2 TAS Prefetching Methodology

As mentioned previously, we classify the time intervals between misses into four categories: "short time" ranges from 1 to 2 misses; "medium time" ranges from 3 to 9 misses; "long time" ranges from 10 to 19 misses and "very long time" is over 19 misses. These thresholds are chosen based on experiments and empirical experience, and our observations show that they are proper in classifying the temporal relations. With the classification of the time intervals, TAS operates as follows. First, the prefetcher finds the corresponding stream matching the PC of current miss, which is exactly the same with the conventional stride prefetcher. Second, the prefetcher checks the current interval_state and takes a corresponding action. If the time interval is short for current stream and it is the first time accessing the stream during this prefetching round, the prefetcher skips two blocks (prefetch distance) following the current miss to avoid late prefetches and issues all prefetches according to the prefetch degree. If the time interval is medium, it is not necessary to apply prefetch distance by skipping blocks because the longer interval between misses can tolerate the fetch latency well. Instead, we only issue half prefetches in order to prevent potential early prefetches. Different from medium time interval case, for the long or very long time interval cases, we issue one quarter of prefetches or one prefetch, respectively, for the current stream. Third, when there are still prefetches left after prefetching on the first stream, the prefetcher attempts going width following the next_PC pointer to find local streams that are within 20 time period and ready to prefetch, and then prefetches on these streams. The prefetching rules for these streams are same with the first one. If the prefetcher has traversed all the streams during the time period along the stream chain while prefetches are not completely issued, the prefetcher will go back to the first stream and repeat the operation until all of them are done. Figure 5 shows the algorithm of TAS operations.

In the next section, we present the simulation and performance analysis of the proposed timely prefetching technique. We also use a variant of stride prefetcher called *multi-level* *stride (MLS)* prefetcher to assess the performance of our stream timing mechanism. Similar to TAS, the MLS prefetcher chains local streams according to their times being accessed. However, MLS only issues one prefetch for each stream and always attempts to traverse the stream chain to find new streams. In other words, we can consider the conventional stride prefetcher is a special case of the TAS prefetcher that never goes width in finding prefetch candidates, while the MLS prefetcher is another special case of TAS prefetcher that never goes depth in finding prefetch candidates.

4. EVALUATION AND ANALYSIS

In this section, we present the simulation verification of the proposed prefetching technique and performance results in detail. We first verify the effects of stream timing mechanism including time interval distribution, future time prediction accuracy and the reduction of untimely prefetches. We, then, evaluate the overall performance of the prefetcher under various environments. Finally, the coverage, accuracy and more behaviors of the prefetching scheme are analyzed.

4.1 Experimental Setup

4.1.1 Simulation Environment

In this study, we conduct experiments with a trace-driven simulator called CMP$im that characterize the memory system performance of single-threaded and multi-threaded workloads. The first Data prefetching Competition (DPC-1) committee [1] released a prefetcher kit that provides partial interface to make it feasible to integrate with an add-on prefetching module. The prefetcher kit contains Pin tool [14] and CMP$im simulator [10] to generate traces and conduct simulation. We utilize these features to evaluate the proposed timely prefetching technique and the TAS prefetchers' performance. As shown in Table 1, the simulator was configured as an out-of-order processor with a 15-stage, 4-wide pipeline (maximum of two loads and maximum of one store can be issued every cycle) and perfect branch prediction (i.e. no front-end or fetch hazard). L1 cache is set as 32KB and 8-way set associative. L2 cache is 16-way set associative, and the capacity is varied from 512KB to 2MB in the experiment. The cache follows LRU replacement policy. The access latency is configured as 20 cycles for L2 cache and 200 cycles for memory.

The simulation was conducted with 20 benchmarks from SPEC-CPU2006 suite [20]. Several benchmarks in the set were omitted because of compatibility issue with our system. The benchmarks were compiled using GCC 4.1.2 with -O3 optimization. We collected traces for all benchmarks by fast forwarding 40 billion instructions then running 500 million instructions. We used the ref input size for all benchmarks.

4.1.2 Hardware Cost and Operation Complexity

The TAS prefetcher requires additional storage budget compared to the existing stride prefetcher. The hardware cost mainly comes from the enhanced parts of the reference prediction table. In our experiments, we set the table size as 512 entries and use 32-bit addresses. The next_PC field consists 9 bits (for referencing one of 512 entries) and the prev_time requires 16 bits. The total hardware storage cost for the RPT of the existing stride prefetcher is:

Table 1: Parameters of simulated CPU

Parameter	Value
Window Size	128-entry
Issue Width	4
L1 Cache	32KB, 8-way
L2 Cache	512KB/1MB/2MB,16-way
Block Size	64B
L2 Cache Latency	20 cycles
Memory Latency	200 cycles
L2 Cache Bandwidth	1 cycle/access
Memory Bandwidth	10 cycles/access

(32+32+32+2)*512=50176 bits (6.1 KB). After enhanced with three additional fields, the hardware storage required for the TAS prefetcher is: (32+32+32+2+9+2+16)*512=64000 bits (7.8 KB). The additional hardware cost for our prefetcher is 1.7KB, which is only 27% of the storage needed by the original stride prefetcher.

Aside from the additional storage demand, the TAS prefetcher involves extra operation time. They stem from updating new fields during table updating stage and traversal along the stream chain during prefetching stage. The major operation time overhead in comparison with the stride prefetching comes from additional "hops" it takes to move between different local streams upon a prefetching event. More details about the behavior analysis of the TAS prefetcher are given in section 4.4. The results show that, in general, the overall operation time of our prefetcher is within an acceptable range.

4.2 Effects of Stream Timing Mechanism

4.2.1 Time Interval Distribution and Prediction

The behavior of the TAS prefetcher is guided by the past history time intervals between consecutive misses in the local stream. Figure 6 shows the time intervals distribution of all selected benchmarks. As can be clearly observed from the figure, the time distance between two adjacent misses from the same instruction is likely to be small. In particular, 31% of the intervals are only one miss, which means that the corresponding misses are required by certain instruction continuously. Recall the thresholds we use to classify the time interval, the figure shows approximately 60% of the time intervals might be labeled as short. However, when the prefetch degree is high, the predicted future misses might occur long time apart from the present time, which potentially leads to early prefetches. The assumption behind stream timing mechanism is that when the data access addresses exhibit patterns, the time of the occurrence of these accesses are highly predictable. For TAS prefetcher, the data patterns are constant strides between addresses. Recall that we classify the time interval between addresses into four categories. Therefore, on the basis of stream timing principle, the time intervals between consecutive addresses in a local stream are supposed to be in the same category and can be used to predict the time of future accesses. Figure 7 shows the accuracy of the time prediction (prediction of the occurrence of the accesses) following this assumption can achieve an average of 90%. The high accuracy of time prediction verifies that the assumption holds well and ensures the feasibility of the proposed mechanism.

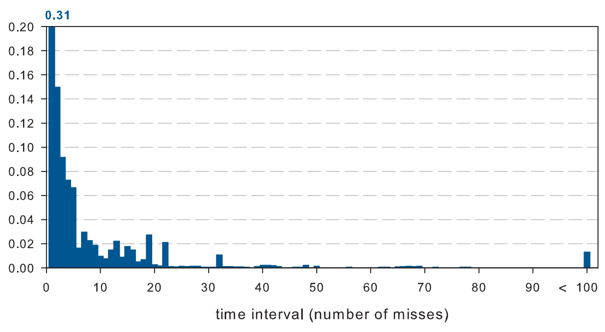

Figure 6: Distribution of time interval between misses in the same local stream.

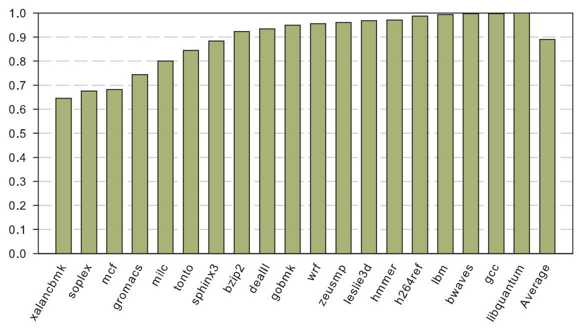

Figure 7: Time prediction accuracy.

4.2.2 *Timeliness Analysis*

In this subsection, we evaluate the effect of stream timing in controlling untimely prefetches and converting them into good ones. We only consider the usable prefetches including early, late and good ones and ignore the useless prefetches since we are not able to change them into good prefetches. Three prefetchers, stride, MLS and TAS, in which 100K prefetches are collected and categorized, are compared. Figure 8 (a-c) demonstrates the distribution of good, late and early prefetches of the three prefetchers respectively. From the figure we can see that the timeliness varies for different applications and for some of them, both late and early prefetches can occur. Particularly, late prefetches are more likely to appear in stride and MLS prefetchers. That is because we choose 8 as the prefetching degree which is a moderate one and is helpful to avoid early prefetches. When the degree is high, such as 32 or 64, more early prefetches will arise in both stride and MLS prefetchers. Note that MLS prefetcher performs worse than others in most of applications. In MLS prefetcher, only one prefetch is issued on each local stream, which is easy to be late when the instruction requests it soon. Another feature of MLS is that it always attempts to travel along the stream chain, which increases the chances of early prefetches if the stream is far away from the current one in time.

Comparing the three prefetchers in the Figure 8, it is clear that our TAS prefetcher, benefiting from stream timing, has less untimely prefetches than the other two schemes. Although TAS cannot guarantee reduction of both late and early prefetches simultaneously, it is able to enlarge the fraction of good prefetches for most of applications as shown in Figure 8(d). The only exception is *zeusmp*, in which stride prefetching achieves more timely prefetches. Our detailed analysis shows that in *zeusmp* the time interval between misses in local stream is very short while the time distance between different streams is long. That is the major reason that TAS has more early prefetches. Fortunately, this small portion of additional untimely prefetches does not hurt the overall performance.

4.3 Overall Performance Analysis

In this subsection, we present the overall performance results of all three prefetchers. We also vary the cache size from 512KB to 2MB to evaluate cache sensitivity and pollution.

4.3.1 *Cache Miss Reduction*

Figure 9 reports the percentage of L2 misses reduced by stride, MLS and TAS prefetcher respectively. On average, TAS reduces L2 cache misses by 56% approximately, which is the best of all three prefetchers compared to 43% of the stride prefetcher and 34% of the MLS prefetcher. Moreover, TAS outperforms others in 18 out of 20 applications and only underperforms the stride prefetcher in the other two. Two benchmarks, *gobmk* and *tonto*, are the only ones on which TAS loses effectiveness in reducing misses. Our detailed analysis reveals that, in *gobmk* and *tonto*, data accesses tend to hit a certain local stream instead of jumping across different streams, which leads to wrong predictions in TAS even though it is able to reduce some late prefetches. The considerable miss reduction in TAS mainly comes from the timeliness improved by reducing untimely misses.

4.3.2 *IPC Improvement*

Figure 10 shows the IPC (Instructions per Cycle) improvements with respect to the base case (without prefetching) of stride, MLS and TAS prefetchers. The simulation result shows that the TAS prefetcher significantly reduces the average data access latency and improves IPC considerably. The IPC improvement of the TAS prefetcher reaches the peak as much as 133% in *libquantum*, and achieves an average speedup of 33% among all benchmarks. Compared to the other two schemes, TAS outperforms them in 16 out of 20 benchmarks and underperforms little (less than 3%) in merely 4 of them. One case of negative performance is shown for *gobmk*. Since all three prefetchers use the same constant stride algorithm, neither stride nor MLS is able to achieve positive performance on this benchmark. Fortunately, TAS does not hurt the performance much, and the performance slowdown is only 3%. We also observe that most of the benchmarks on which TAS gains little performance or is outperformed by other prefetchers have a low miss rate (under 17%), which means that the efficiency of stream timing technique is throttled by limited quantity of misses. Two exceptions are *xalancbmk* and *milc* benchmarks. Although they have high miss rates, TAS performs slightly worse than the stride prefetcher with no more than 2% performance difference, and we consider them acceptable. Another observation is that the MLS prefetcher has lower average performance improvement than others. As mentioned in previous sections, the MLS prefetcher suffers the limitations of late prefetches, which causes the overall performance drop.

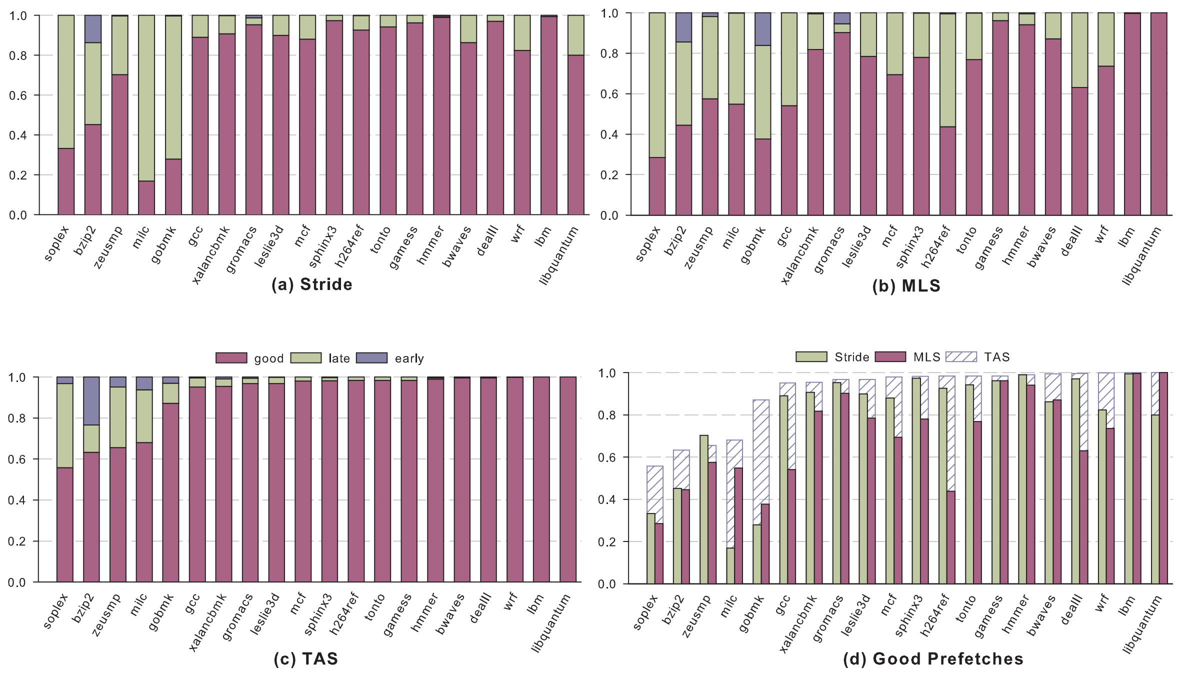
Figure 8: Breakdown of usable prefetches with prefetch degree 8 for (a) stride prefetching; (b) MLS prefetching; (c) TAS prefetching; (d) good prefetches comparison of three prefetchers.

4.3.3 *Cache Sensitivity and Pollution*

Cache pollution is considered as a critical side effect of data prefetching. In this subsection, we present analysis of cache size sensitivity and cache pollution of TAS prefetcher under various cache sizes. Figure 11 shows the cache size sensitivity of each benchmark. Figure 12 demonstrates the comparison of the performance improvement of the TAS prefetcher for different cache sizes. We observe that the increase of cache size lower the effectiveness of TAS prefetcher in 7 out of 20 benchmarks. These benchmarks roughly match the ones in Figure 11 whose performance are sensitive to cache sizes. This is because that, a larger cache helps to improve the performance significantly and leaves TAS prefetcher little space for further improvement.

From Figure 12, we observe that the TAS prefetcher gains substantial and stable performance improvement regardless of cache size in some insensitive benchmarks such as *bwaves*, *mcf*, *libquantum* and *lbm*. This result shows that the TAS prefetcher is successful in controlling the cache pollution, and therefore, TAS is able to maintain high efficiency even the cache is small. Recall that the stream timing in TAS improves the prefetching timeliness by making most of prefetches issued at the right time, which potentially avoids cache pollution resulting from early cache line replacements.

4.4 TAS Prefetching Characterization

To get a better understanding of the prefetchers' performance, we extend our experiment to evaluate advanced prefetching characteristics from three aspects: prefetching coverage, prefetching accuracy and prefetching behaviors. In this set of experiments, we use prefetching degree 8 and 512 KB L2 cache for each prefetcher.

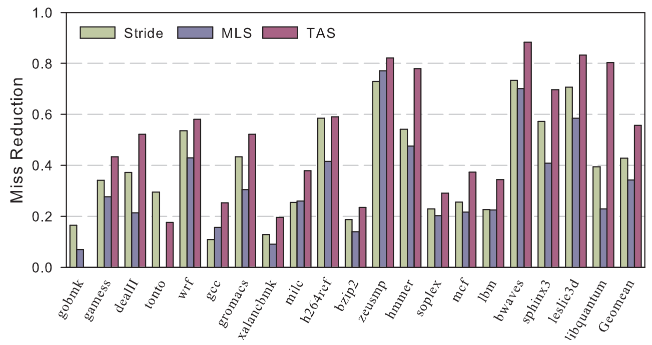
Figure 9: Miss Reduction of stride, MLS and TAS prefetchers with degree of 8 and 512KB LLC.

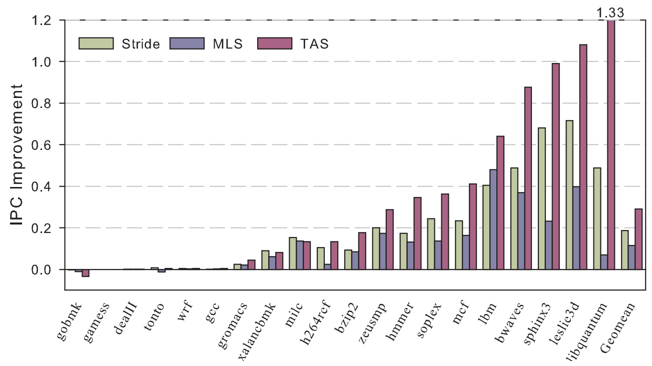
Figure 10: IPC improvement of stride, MLS and TAS prefetchers with degree of 8 and 512KB LLC.

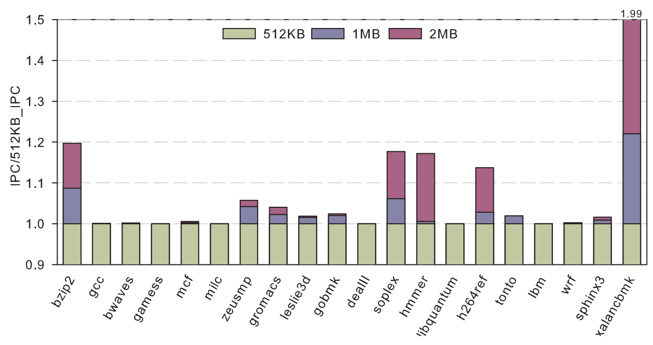

Figure 11: Normalized IPC for 512KB, 1MB and 2MB L2 cache without prefetching.

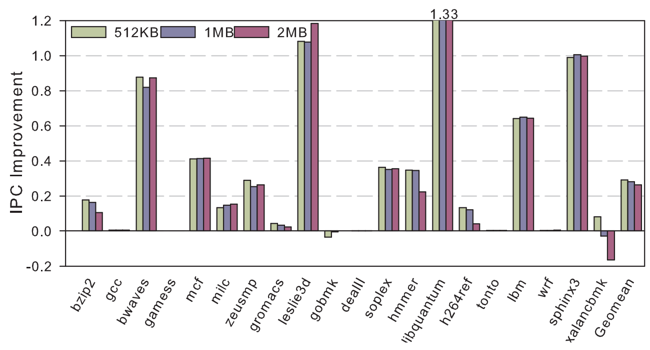

Figure 13: IPC improvement of TAS prefetcher for 512KB, 1MB and 2MB L2 cache.

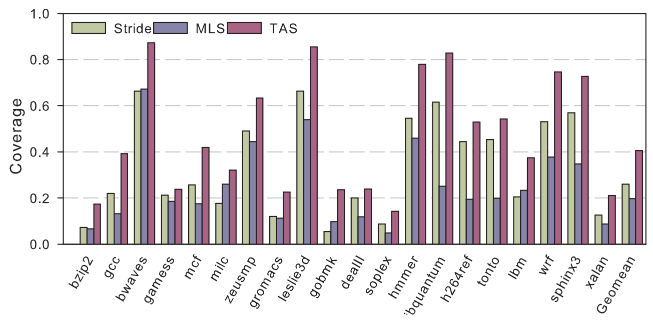

Figure 12: Prefetching coverage for Stride, MLS and TAS prefetchers.

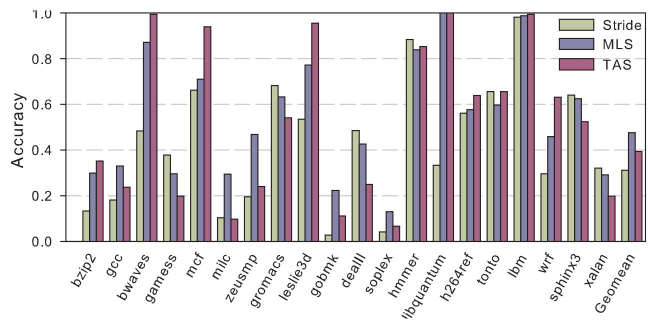

Figure 14: Prefetching accuracy for Stride, MLS and TAS prefetchers.

4.4.1 Prefetching Coverage and Accuracy

Figure 13 shows the prefetching coverage for the stride, MLS and TAS prefetchers. Benefiting from the stream timing, the number of timely prefetches of the TAS prefetcher largely increases; hence, the coverage of TAS is higher than others in every benchmark. Moreover, the geometric mean of all 20 benchmarks' coverage reaches as much as 41% which can be regarded as remarkably high for a prefetcher.

The prefetching accuracy of stride, MLS and TAS prefetchers are shown in Figure 14. The MLS prefetcher wins because the “one prefetch per stream” policy helps it effectively prevent data pattern change in the local streams, and therefore, issues much less useless prefetches than the other two strategies. The geometric mean of accuracy of TAS prefetcher is 40%, which is 8% lower than MLS prefetcher but 8% higher than stride prefetcher. This result shows that the stream timing technique plays a crucial role in helping the prefetcher to maintain a relatively high accuracy and coverage while improving the timeliness.

4.4.2 Prefetching Behaviors

We provide further analysis of how the stream timing mechanism works in TAS prefetcher. As previously mentioned, the local streams are chained according to their time stamps so that the next stream can be successfully found. The prediction accuracy of the next stream is useful for finding proper prefetching candidates. We consider the next stream prediction is accurate when the corresponding PC appears within the next five PCs. Table 2 shows the next PC prediction accuracy when the prefetching degree is 8. The geometric mean of the accuracy of all 20 benchmarks is 73%, which is very high and therefore is helpful to achieve a high prefetching accuracy.

Stream timing mechanism always tries to find a better solution of prefetching by either going depth or going width. For a certain prefetching event, the prefetcher probably issues prefetches only on one local stream. In this case, we term this prefetching event as a *single event*. Otherwise the prefetcher works on multiple streams and falls in other two cases. The first case is that the stream chain is so short that after issuing prefetches on each stream, the number of prefetches does not satisfy the prefetching degree yet. Since the prefetcher should go back to the first stream and start prefetching again, we term this prefetching event as a *cyclic event*. The last case is that the stream chain is long enough for prefetching and we term it as a *normal event*. As previously discussed, we adopt preset thresholds to classify the time interval between different misses into short, medium, long and very long categories. To understand how these thresholds affect the behavior of the prefetcher, we bring two more configurations as shown in Table 2. From the table we can see that the percentage of three prefetching events varies considerably among different benchmarks. For example, more than 99% of prefetching events are single ones in gcc (prefetcher acts similar to the conventional stride prefetching), which indicates that the time intervals between two adjacent misses in a stream are very short. This result also explains why little performance difference is shown between the stride and TAS prefetchers. However, in some benchmarks such as milc, mcf and lbm, the cyclic events become the dominant part. Another observation is that the config 1 tends to make the prefetcher go width (more cyclic

Table 2: Statistics of behavior of the TAS prefetcher

Benchmarks	Events (%) (d=8)						Total Hops per Event						NextPC Accuracy (d=8)	Configurations	
	config 1		*config 2*		*config 3*		*config 1*		*config 2*		*config 3*				
	single	*cyclic*	*single*	*cyclic*	*single*	*cyclic*	*d=8*	*d=16*	*d=8*	*d=16*	*d=8*	*d=16*			
bzip2	64.0	29.5	79.0	18.5	84.0	13.0	1.82	2.15	1.36	1.21	1.19	1.13	73%	***config 1***	
gcc	99.5	0.5	99.5	0.5	100.0	0.0	1.00	1.00	1.00	1.00	1.00	1.00	97%		
bwaves	33.0	27.0	57.5	11.0	71.5	7.0	2.21	2.00	1.55	1.47	1.34	1.27	67%	short	[1,1]
gamess	12.0	62.0	33.5	38.5	35.5	27.0	4.83	8.16	3.73	5.65	3.03	4.12	56%	medium	[2,4]
mcf	1.5	97.5	20.5	75.0	39.0	56.0	3.79	5.17	1.91	1.90	1.60	1.49	89%	long	[5,9]
milc	5.0	90.0	14.5	78.0	14.5	50.0	5.89	8.26	3.22	3.38	2.21	2.48	35%	very long	>9
zeusmp	14.5	47.0	31.0	37.5	89.0	4.0	2.13	2.15	1.72	1.71	1.11	1.09	97%		
gromacs	3.5	94.0	4.5	93.5	11.0	88.0	6.39	8.72	5.12	7.54	3.79	6.23	64%	***config 2***	
leslie3d	54.5	25.5	62.0	11.5	71.0	5.5	2.92	3.85	1.88	1.70	1.28	1.27	81%		
gobmk	44.5	52.0	49.0	44.5	54.5	37.5	3.57	5.85	3.18	5.31	2.81	4.32	66%	short	[1,2]
dealII	41.0	49.5	56.5	34.5	68.5	27.5	2.78	4.80	2.41	4.26	2.11	3.23	64%	medium	[3,9]
soplex	5.5	90.5	9.0	84.5	17.5	70.0	5.26	8.07	3.63	4.14	2.30	2.49	54%	long	[10,19]
hmmer	22.5	42.5	63.0	7.5	78.0	4.5	2.67	3.60	1.69	2.14	1.35	1.79	59%	very long	>19
libquantum	100.0	0.0	100.0	0.0	100.0	0.0	1.00	1.00	1.00	1.00	1.00	1.00	100%		
h264ref	62.5	28.5	73.5	15.0	85.5	5.5	1.77	2.06	1.53	1.87	1.23	1.23	86%	***config 3***	
tonto	71.5	24.5	76.0	20.5	80.5	18.5	2.16	1.88	1.49	1.22	1.19	1.09	98%		
lbm	0.0	99.5	0.0	94.5	0.0	84.0	6.70	10.10	4.65	5.70	2.61	2.47	91%	short	[1,4]
wrf	90.0	3.0	92.5	1.0	95.5	0.5	1.18	1.25	1.07	1.08	1.04	1.04	83%	medium	[5,19]
sphinx3	35.0	47.5	61.0	27.0	65.0	22.0	2.97	5.17	2.57	4.71	1.99	3.41	72%	long	[20,49]
xalancbmk	8.5	71.5	18.5	60.0	90.5	6.0	2.11	2.29	1.94	1.97	1.18	1.33	75%	very long	>50
Geomean							2.71	3.43	2.07	2.39	1.62	1.83	73%		

events) while the config 3 tends to make the prefetcher go depth (more single events). That is because the time interval classification is the criteria of prefetching directions.

As previously discussed, the stream timing mechanism does not only consume extra storage but also requires additional operation time, which mainly comes from the traversal time of the stream chain. We calculate the average hops required per prefetching event for various configurations and degrees. From Table 2 we can see that the number of Hops per Event (HpE) is related with configuration and prefetching degree. A high prefetching degree usually requires more HpE than lower ones since many issued prefetches cause frequent movement between streams. There is also notable difference of HpE between three configurations. Although the config 1 appears to be better than others since its range of short, medium and long time classification seems more reasonable, the HpE is much higher than other configurations which in turn adds more operation time. However, config 3 is also not good because its wide range affects the performance even though the operation time is low. Thus, in this study, we use config 2 and the geometric mean of HpE of all 20 benchmarks is 2.07 when the prefetching degree is 8. In this case, the extra operation time required by using stream timing mechanism that is caused by one more hop per prefetching event is very low.

5. RELATED WORK

In this section, we classify the representative works in several categories and discuss their basic ideas. We also discuss the latest work in improving timely prefetches and compare them with our proposed approach.

5.1 Data Prefetching Mechanisms

Data prefetching has been extensively studied in decades [2][3][4][11][6][21][22] and widely used in commercial processors [8][23]. Sequential prefetching [6] mechanism takes advantage of spatial locality and assumes the applications usually request consecutive memory blocks. Stride prefetching [2] detects the stride patterns in data access streams. A Reference Prediction Table (RPT) is used to keep track of strides of recent data accesses. Once the prefetcher is trained, blocks can be prefetched according to the stride recorded in RPT. Due to its simplicity and effectiveness, stride prefetching is widely used [8]. Markov prefetching was proposed in [11] to capture the correlation between cache misses and prefetch data based on a state transition diagram. Delta correlation prefetching is originally proposed for TLB [12] and can be used as PC/DC prefetching in data cache [16].

5.2 Stream Localization Mechanisms

To identify more accurate data patterns, the global miss stream is usually localized. Stream localization by PC is widely used in [2][16]. Concentration zone (czone) was proposed in [17] to localize the stream according to the memory region. AC/DC prefetching [15] uses GHB [16] and spatial localization to implement delta correlation prefetching. To predict memory access on a shared-memory multi-processor, spatial memory streaming was proposed [18], which identify data correlations for commercial workload. Different from PC localization and spatial localization, temporal localization proposed in [24] groups the addresses occurring in the same time period. However, after stream localization, the miss sequence order in global stream is lost, which might be useful for timely prefetching or controlling cache pollution. To regain this missing global order, a recent work called spatial-temporal streaming [19] that combines spatial and temporal streaming. With the similar goal, the stream chaining technique [7] attempts to reconstruct the chronological order of localized streams by chaining the local streams into one using control flow and prefetches along the chain. However, the overall sequence order is so difficult to obtain due to storage limitation that only relative order is available in most cases.

5.3 Timeliness Improvement Techniques

The localization mechanisms discussed in the previous subsection usually leads to untimely prefetching, which is probably the most critical issue that limits the performance of the prefetcher. Feedback directed prefetching [21] collects feedback during runtime and applies a proper prefetching distance to avoid late prefetches. Another mechanism targets to reduce late prefetches called epoch-based prefetching [5]. The misses are localized according to epoch (a fixed-length of time), and prefetching based on next epoch is helpful in preventing late prefetches. However, neither of them can deal with early prefetches, and actually they might cause additional early prefetches. Close to our work, the dead-block predictor (DBP) [9][13] attempts to predict when a cache block is dead by tracking the time duration between when it comes into the cache and when it is evicted. The timing information provided by DBP is used for triggering prefetches and making replacement decision. The main difference between timing characteristic in this paper to the one discussed in the dead-block predictor is that DBP is guided by the time duration a block should stay in the cache, while our method tries to acquire the time when a miss is supposed to occur in the future. The dead-block predictor is successful in reducing the cache pollution since it finds useless dead-block to replace. However, it does not take into account the case that a prefetch is brought into cache too early and evicted by others before used, which results in untimely prefetches. Our work can solve these issues well.

6. CONCLUSION

In this study, we advance the state-of-the-art of data prefetching technique via introducing a novel stream timing mechanism aiming to reduce untimely prefetches. The proposed stream timing technique can maintain the chronological order of localized streams and the accesses within each stream. These timing information are valuable to reduce untimely prefetches, potential cache pollution and bandwidth consumption and to improve the effectiveness of data prefetching. We have extended the conventional stride data prefetcher with stream timing technique and proposed a new Time-Aware Stride (TAS) prefetcher. With extensive simulation testing, we found that the prefetching timeliness can be improved with stream timing scheme, and this benefit can be transferred to significant performance improvement. We have also analyzed the detailed prefetching coverage, prefetching accuracy and the TAS prefetcher's characteristics. The results show that the TAS prefetcher can achieve high coverage and accuracy and outperforms existing stride prefetchers considerably. The detailed study of the TAS prefetcher's characteristics verifies that the hardware requirement for the proposed stream timing technique is trivial.

7. ACKNOWLEDGMENTS

This research was supported in part by National Science Foundation under NSF grant CCF-0621435 and CCF-0937877.

8. REFERENCES

[1] DPC-1 Homepage. *http://www.jilp.org/dpc*, 2008.

[2] T.-F. Chen and J.-L. Baer. Effective hardware based data prefetching for high-performance processors. *IEEE Trans. Computers*, 44(5):609–623, 1995.

[3] Y. Chen, S. Byna, and X.-H. Sun. Data access history cache and associated data prefetching mechanisms. In *SC*, 2007.

[4] Y. Chen, H. Zhu, and X.-H. Sun. An adaptive data prefetcher for high-performance processors. In *CCGrid*, 2010.

[5] Y. Chou. Low-cost epoch-based correlation prefetching for commercial applications. In *MICRO*, 2007.

[6] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *ICPP*, pages 56–63, 1993.

[7] P. Diaz and M. Cintra. Stream chaining: Exploiting multiple levels of correlation in data prefetching. In *ISCA*, pages 81–92, 2009.

[8] J. Doweck. Inside Intel Core microarchitecture and smart memory access. *Intel White Paper*, 2006.

[9] Z. Hu, M. Martonosi, and S. Kaxiras. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *ISCA*, pages 209–220, 2002.

[10] A. Jaleel, R. S. Cohn, C. keung Luk, and B. Jacob. CMP$im: a pin-based on-the-fly multi-core cache simulator. In *4th Annual Workshop on Modeling, Benchmarking and Simulation*, pages 28–36, 2008.

[11] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA*, pages 252–263, 1997.

[12] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. In *ISCA*, pages 195–206, 2002.

[13] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA*, pages 144–154, 2001.

[14] C.-K. Luk, R. S. Cohn, R. Muth, and et. al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[15] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: an adaptive data cache prefetcher. In *IEEE PACT*, 2004.

[16] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA*, 2004.

[17] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA*, pages 24–33, 1994.

[18] S. Somogyi, T. F. Wenisch, and et. al. Spatial memory streaming. In *ISCA*, 2006.

[19] S. Somogyi, T. F. Wenisch, and et. al. Spatio-temporal memory streaming. In *ISCA*, 2009.

[20] C. D. Spradling. SPEC CPU2006 benchmark tools. *ACM SIGARCH Computer Architecture News*, 2007.

[21] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, pages 63–74, 2007.

[22] X.-H. Sun, S. Byna, and Y. Chen. Server-based data push architecture for multi-processor environments. *JCST*, 22(5), 2007.

[23] J. M. Tendler, J. S. Dodson, J. S. F. Jr., and et. al. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[24] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *ISCA*, pages 222–233, 2005.

[25] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.