Journal of Instruction-Level Parallelism 13 (2011) 1-15 Submitted 3/10; published 1/11

Enhancements for Accurate and Timely Streaming Prefetcher

Gang Liu GALIU@CISE.UFL.EDU
Zhuo Huang ZHUANG@CISE.UFL.EDU
Jih-Kwon Peir PEIR@CISE.UFL.EDU
Department of Computer & Information Science & Engineering
University of Florida
Gainesville, FL 32611 USA

Xudong Shi XDSHI@GOOGLE.COM
Google, Inc
1600 Amphitheater Pkwy
Mountain View, CA 94043 USA

Lu Peng LPENG@LSU.EDU
Department of Electrical & Computer Engineering
Louisiana State University
Baton Rouge, LA 70803 USA

Abstract

In this paper, we describe several enhancement techniques to improve the state-of-the-art stream prefetcher. First, the enhanced stream prefetcher takes streams with long stride into consideration to avoid wasteful prefetches. Second, accessing a node in a tree or graph structure may have a different direction than the traversal direction through the structure. The enhanced stream prefetcher eliminates this type of noise for establishing the stream. Third, regular streams for array accesses are often repeated. Initiating penalty can be avoided by early re-establishing a repeated stream. Fourth, an established stream may be dead before being removed from the stream prefetching table. A dead stream removal scheme reduces inaccurate prefetches. Performance evaluations based on SPEC applications show that the enhanced stream prefetcher improves 38%, 42%, and 55% of CPI for the three tested cache configurations provided by the 1st JILP Data Prefetching Championship Committee [19] with respect to the base design without prefetching. In comparison with the original stream prefetcher, the improvements are 2%, 18%, and 19% respectively.

1. Introduction

With fast advances in processor technology, the speed gap has been continuously widening between processors and main memory. It usually takes hundreds of processor cycles to access the off-die memory. Caches play a critical role in bridging this performance gap by retaining the recent accessed instructions and data for fast accesses. Recently, many advanced caching mechanisms have been proposed for on-die CMP caches [1, 2, 3, 4]. However, due to limited cache capacity, the required working set of applications may not fit into the cache and causes frequent accesses to memory.

Prefetching is an important mechanism to reduce memory access penalty. Existing data prefetching methods are based on two general behaviors of the missing block addresses: *regularity* and *correlation*. Existing sequential [5], stride [6, 7], distance [8, 9, 10] and regular stream [11, 12] prefetchers dynamically capture the regularity of a sequence of missing block addresses to speculatively prefetch subsequent blocks. Correlation-based prefetchers, such as correlated [13], Markov [14], hot-stream [15], temporal streaming [16], spatial-streaming [17] and a combination of temporal/spatial streaming [18] prefetchers, on the other hand, record the history of nearby missing addresses to trigger prefetches assuming such miss correlations will be repeated. This approach, however, incurs a significant space overhead for the miss history.

Among the regularity-based prefetchers, the stream prefetcher captures a sequence of nearby misses when their addresses follow the same positive or negative direction in a small memory region. Once a stream is identified, a demand miss targeting the current active stream triggers prefetches of consecutive blocks in the detected direction. Two key parameters: prefetch *distance* and *degree* control the aggressiveness of stream prefetching. The prefetch distance defines the stream monitor region and how far ahead to trigger the prefetch. The prefetch degree defines the number of consecutive blocks to be prefetched. Due to its simplicity and effectiveness, the stream prefetcher has been implemented in commercial processors [11, 12].

In this paper, we evaluate several enhancement techniques for optimizing a stream prefetcher. First, the stream prefetcher is augmented with a stride distance. Upon detecting memory accesses with a constant stride (measured in bytes) of longer than one block, the stream prefetcher prefetches blocks according to the detected stride. Second, we observe that in a few applications, accessing a data structure through pointers such as trees and graphs, each node in the data structure may occupy more than one block. Hence, accesses within a node may have a different addressing direction than the traversal direction through the nodes. When such a case is detected, we allow a stream to be formed by ignoring the noise, i.e. an adjacent block access in the opposite direction. Third, we observed that regular streams for array accesses are often repeated. To reduce the penalty of re-initiating a stream, we allow a previous repeated stream to launch again after the old stream is caught up by a new stream. Fourth, a short stream may be dead before being replaced from the stream table. Subsequent hits to a dead stream initiates inaccurate prefetching without going through the needed training stage. Detecting and removing aging short streams from the stream table leads to more accurate prefetching.

Performance evaluations based on a set of SPEC2000 and SPEC2006 benchmarks show that the enhanced stream prefetcher makes significant improvement over the original stream prefetcher. For the three L2 cache configurations listed in Table 1, the enhanced stream prefetcher improves 38%, 42%, and 55% of their Cycle-Per-Instructions (CPIs) with respect to the base design without prefetching. In comparison with the original stream prefetcher, the improvements are 2%, 18%, and 19% respectively.

The remainder of this paper is organized as follows. Section 2 provides the basic design of stream prefetcher. Section 3 describes the benefit of constant stride, stream repetition, as well as the challenge in handling stream noise and dead stream removal. Section 4 describes the details of the enhancement techniques. Section 5 provides the evaluation methodology and Section 6 presents the evaluation results. Related work is given in Section 7 followed by a brief conclusion in Section 8.

2. Stream Prefetcher Basics

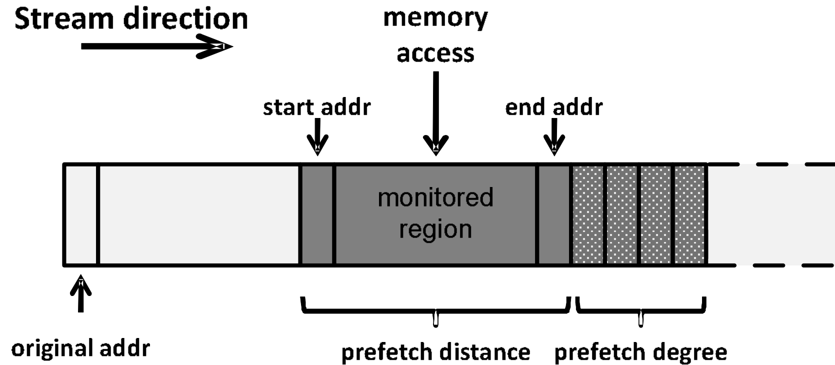

Figure 1: Basic design of a Stream Prefetcher.

The stream prefetcher we model is based on the one presented in [11] which is originated from the IBM POWER4 processor [12]. The stream prefetcher uses a *stream table* to keep track of multiple access streams. All established streams (also referred as *trained* streams) in the stream table are monitored against the cache misses. When an incoming memory request falls into the current monitor window of a trained stream, the stream prefetcher prefetches consecutive blocks according to the direction of the trained stream.

A demand missing block address enters the stream table in an *untrained* state if the missing block has not been recorded. An untrained stream becomes *trained* based on the following conditions. First, the next two consecutive misses located in the same memory region as an untrained miss are examined. In the design reported in [11], the memory region covers 16 blocks before and after the original missing block. Second, these three consecutive misses are in the same ascending or descending direction starting from the original miss.

The aggressiveness of a stream prefetcher is controlled by the *distance* and the *degree* of the stream as illustrated in Figure 1. The original miss address and the current stream window defined by the *start* and the *end* block address of each trained stream are recorded in the stream table. The number of blocks from the start to the end blocks determines the *prefetch distance*, which indicates the stream monitor region as well as controls how far ahead of the demand access stream that the prefetcher can prefetch. The *prefetch degree*, on the other hand, controls the number of consecutive blocks for each stream prefetcher. When a new memory request falls into the current monitored region of a trained stream, the stream prefetcher prefetches the next n consecutive blocks starting from the end of the monitored region, where n is the prefetch degree. Afterwards, the monitored start and end address are advanced by n blocks to keep the stream moving along the stream direction.

3. Stream Enhancement Techniques

3.1. Constant Stride Optimization

Memory Allocation:
```
for (i=0;i<numf1s;i++) {
        //numf1s = 10000,numf2s = 11
        bus[i] = (double *)malloc(numf2s*sizeof(double));
        tds[i] = (double *)malloc(numf2s*sizeof(double));
}
```

Memory Access:
```
for (tj=0;tj<numf2s;tj++) {
        …
        for (ti=0;ti<numf1s;ti++)
                Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Figure 2: Code segment from scanner.c in SPEC2000 Benchmark *art*.

The first enhancement is to integrate long-stride prefetching into the stream prefetcher. In the original stream prefetcher, streams are prefetched by consecutive blocks. In real applications, it is not uncommon that memory accesses are followed a constant stride across multiple blocks. Although such long-stride accessing patterns can be captured by a stream prefetcher, stream prefetching of consecutive blocks wastes memory bandwidth and pollutes the caches. For example, in examining *scanner.c* in *art* from SPEC2000 Benchmark (Figure 2), we can identify a constant-stride access pattern across multiple blocks. Note that the memory references in this routine causes 80% of all misses in *art* on the baseline 1MB L2 cache without prefetching. As shown in the memory allocation part, the size of each element of the two arrays *bus* and *tds* is 88 bytes. Each element of two arrays are allocated one-by-one in a round-robin fashion. Therefore, two adjacent elements in *bus* are 192 bytes apart after padding the arrays with 16 bytes. Given a cache line size of 64 bytes, consecutive accesses to array *bus* span across 3 blocks since &*bus*[i+1][j] - &*bus*[i][j] = 192 bytes. As a result, 2 out of 3 prefetched blocks are wasted by the original stream prefetcher.

In the enhanced stream prefetcher, a straight-forward solution dynamically detects the stride distance information. If a constant stride is detected, instead of prefetching consecutive blocks, the constant stride is used to calculate the correct blocks to avoid extra prefetches. More design details will be given in the next section.

3.2. Noise Removal

In training a stream in a stream prefetcher [11, 12], three consecutive misses addressing in a small memory region are examined. These three misses must follow the same ascending or descending direction to successfully train the stream. When a training stream fails, it is likely that two

consecutive misses are addressing memory blocks from the opposite directions with respect to the first miss. After examining the unsuccessfully trained streams in *soplex*, it is interesting to see that out of 11484 unsuccessfully trained streams, 7343 are due to an access to the next adjacent block in the positive direction (i.e. a positive distance of one block). We traced the memory reference pattern of *soplex*, and found it behaves as illustrated in Figure 3. Although the overall direction of the stream is descending, the positive one-block jumps keep preventing the stream from being trained.

To remedy this problem, we allow a training stream to stay untrained when the subsequent miss occurs in the opposite direction from the current miss and one of the misses in the opposite direction is accessing the adjacent block. In other words, the adjacent block access is considered as *noise* and is ignored in training the stream as illustrated in Figure 3.

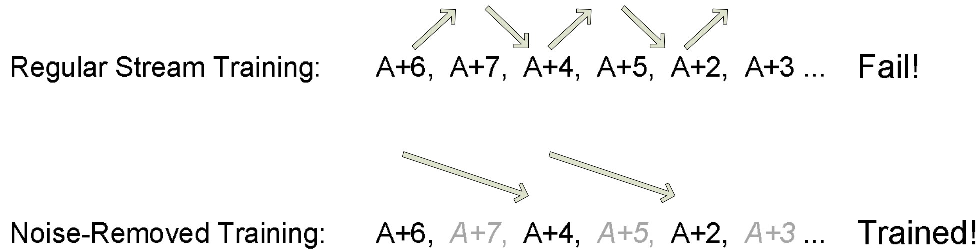

Figure 3: An abstract example of stream training with noise removal.

3.3. Early Launch of Repeat Stream

Besides the constant stride accesses in *scanner.c* (Figure 2), we also observe that a stream access is often repeated. In the nested loop of the example, the streaming array *bus* is accessed repeatedly 11 times with exactly the same start and end addresses. In addition, the repetition of the streaming accesses is separated only by a few instructions. It is beneficial to initiate the stream prefetching again from the recorded original stream address when the current stream is coming to an end. By keeping previous long streams in the stream table, we can detect repeated streams once two streams overlap in the monitored region. Early launching a repeated stream can reduce the penalty of initiating a new stream prefetching.

3.4. Dead Steam Removal

Given the relatively loose condition in training a stream as described in Section 2, many streams can be established even if they are not an accurate stream for prefetching. Furthermore, we also observe that for many short streams that remain inactive for a long period of time, the stream likely has come to an end. These dead streams may still stay in the stream table and can accidentally catch an incoming memory access to trigger prefetches. These incorrect prefetches pollute the cache and waste memory bandwidth.

The number of dead streams in the stream table goes up with the table size. Simulation results show that with a 128-entry stream table, roughly 88% and 62% of the trained streams have stayed in the table longer than 100K cycles without triggering any prefetches respectively for *art* and *ammp*. When a prefetched block is triggered by an old stream that has not triggered any prefetch over 100K cycles, it is useless 92% of the time.

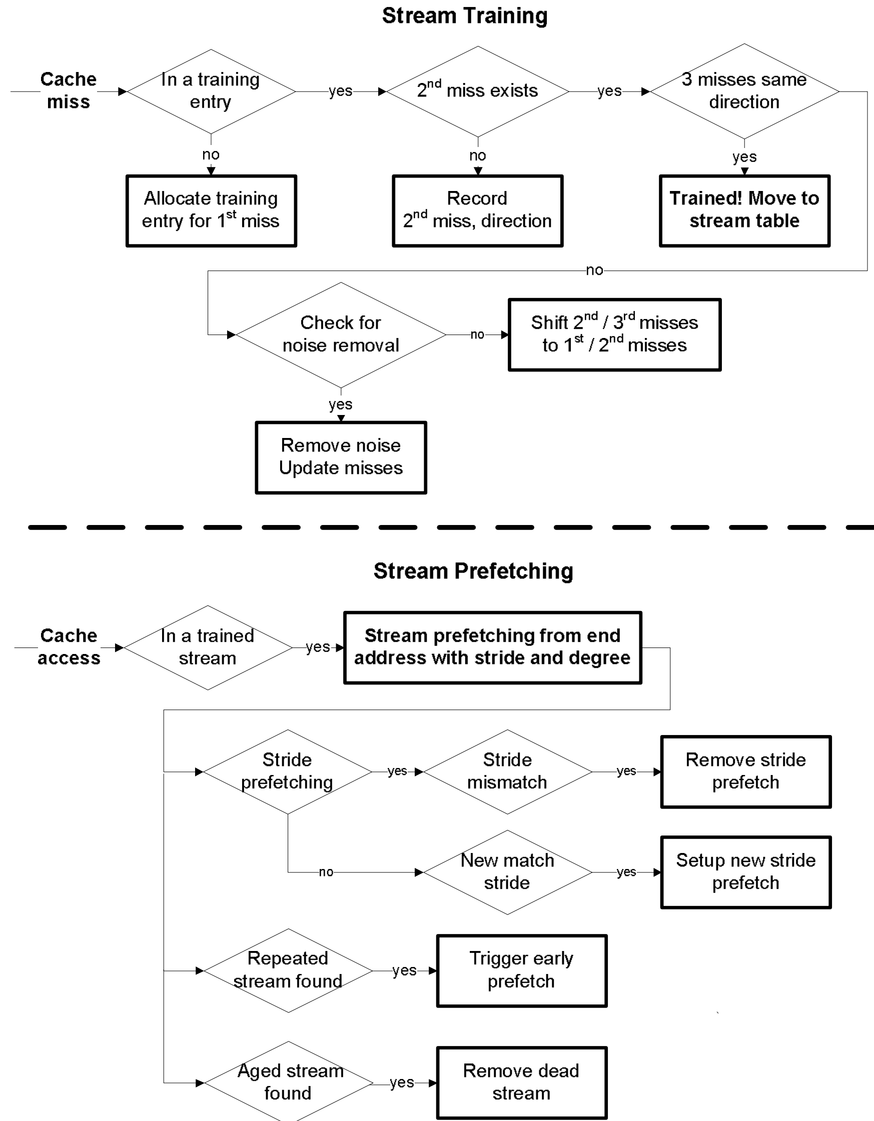

Figure 4: Flowchart of enhanced stream training and prefetching.

Although a smaller stream table can naturally replace streams before they die, smaller tables may suffer insufficient space to keep all the active streams. Hence, by removing dead streams dynamically based on their *ages* in a reasonable-size stream table, the active streams can likely be maintained without holding many dead streams and causing incorrect prefetches.

4. Enhanced Stream Prefetcher Design

In this section, we describe the detailed designs and operations of integrating the four enhancement techniques into the original stream prefetcher. The same two steps: *training* and *prefetching* are performed in the enhanced stream prefetcher. In the training stage, the noise

removal technique is integrated to screen accesses with the noise behavior as shown in Figure 3. In the prefetching stage, correct memory blocks with constant-stride accesses are detected and prefetched. In addition, repeated streams are captured and triggered earlier when the current stream catches up an existing stream in the stream table. Furthermore, dead streams are removed from the stream table based on the stream age.

Error! Reference source not found. shows the flowchart of the enhanced stream prefetching. The basic designs and data structures are given in the following subsections. Note that for simplicity, we use separate *training* and *streaming* tables in the respective stages. They can be combined into a unified table.

4.1. Stream Training

1st Miss	***2nd Miss***	***3rd Miss***	***Direction***	***Noise Flag***
26 bits	5 bits	5 bits	1 bit	1 bit

Figure 5: Training table entry.

Each entry in training table is depicted in Figure 5. Three consecutive misses in a small training window need to be captured for stream training. All the misses are represented by cache block addresses, with 26 bits for cache block address of the *1st miss*, and 5 bits representing a distance of +/- 16 blocks from the *1st miss* to the *2nd miss* or the *3rd miss.* Hence the training window of each stream has 32 blocks. *Direction* is an indicator of an ascending or descending stream. The *Noise Flag* marks the activation of the noise removal in training. The training stage follows several steps.

1. When a cache miss occurs, both the training and the stream tables are searched. If the miss falls into the monitored region of a trained stream, the miss will not be trained again.
2. If the miss block does not fall into any training window (i.e. within positive and negative 16 blocks of the recorded miss) in the training table, a new entry is created to record the new miss for training and the LRU entry is replaced.
3. If the miss is within a training window of a recorded miss, three actions are followed.
    a. Record the block distance from the *1st miss* in case the miss is the *2nd miss* of the training stream.
    b. If the *3rd miss* is detected and all three misses are following the same direction, the stream is trained and is moved to the stream table for prefetching.
    c. If the three misses are not in the same direction, there are two conditions.
        i. If an access to the adjacent block is detected which is in the opposite direction of the stream training, such an access is treated as a noise and removed.
        ii. If the three misses do not satisfy the noise removal condition, the *2nd miss* and the *3rd miss* replace the *1st miss* and the *2nd miss* in the corresponding training stream for continuing the training.

4.2. Stream Prefetching

Orig Addr	Start Addr	End Addr	Last Addr	Direction	Stride	StrEn	Repeat	TimeStamp
26 bits	9 bits	32 bits	16 bit	1 bit	9 bits	1 bit	1 bit	32 bit

Figure 6: Stream table entry.

After a stream is successfully trained, the stream is moved from training table to stream table. The information recorded in each stream table entry is given in Figure 6. *Orig Addr* is the cache block address of the *1st miss* from the training table. *Start Addr* and *End Addr* form the monitored region, with 32-bit full address for *End Addr*, and 9 bits of block distance from *Start Addr* to *End Addr*. *Last Addr* records the actual last memory access in form of byte distance from *End Addr* for the purpose of detecting constant stride accesses. *Direction* is the indicator of an ascending or descending stream. *Stride* records the last stride distance in bytes. *StrEn* flag enables stride prefetching based on the detected stride distance. *Repeat* flag is used to mark a repeated stream. Finally, *TimeStamp* stores the *age* in CPU cycles of the last stream prefetching triggered by the recorded stream. The enhanced stream prefetching follows several steps.

1. When a memory access falls into the monitored region of a trained stream, stream prefetching of subsequent *n* blocks is triggered where *n* is the prefetch *degree*. The prefetching starts from the block following the *End_Addr* according to the stream direction for *n* consecutive blocks.
2. A stream with constant-stride over the length of one block can be detected dynamically and used for accurate stride prefetching. When a memory access occurs in the monitored region of a trained stream, the memory address is saved in *Last Addr*, and the access stride in byte granularity is recalculated. In case the new *Stride* matches the previous *Stride*, the *StrEn* flag is turned on and the subsequent prefetches will be based on the recorded *Stride*. Note that whenever a new *Stride* mismatches the recorded *Stride*, the *StrEn* flag is turned off and the prefetcher is reset to the stream prefetching of consecutive blocks.
3. The detection and early prefetching of repeated streams work as follows. When the forwarded monitored region of an active stream overlaps with the monitored region of another inactive stream in the stream table, a repeated stream is discovered under two conditions. First, the two streams have their starting addresses closely to each other, i.e., the two streams start from nearby addresses. Second, both streams are long streams which cover more than 256 blocks. Upon discovery of such a repeated stream, prefetching of the inactive stream is triggered from its original address.
4. The *age* of an existing stream is used to identify and remove potential dead streams from the stream table, where the *age* is measured from the last time when a stream prefetching is triggered. A global *TimeStamp* is used to calculate the stream age. The current *TimeStamp* is saved into the stream table whenever a memory access occurs to a stream. The age of all the streams in the stream table are checked periodically. The potential dead stream is identified and removed when the stream has been idled for a long period time (> 10K cycles) and the stream is a short stream covering less than 256 blocks.

5. Evaluation Methodology

To demonstrate the advantages of the enhanced stream prefetcher, we selected twelve benchmarks with high L2 Misses-Per-Kilo-Instructions (MPKI) from SPEC2000 and SPEC2006. Trace-driven simulations were carried out using the CMPsim tool set provided by the 1st JILP Data Prefetching Championship competition committee [19]. The traces were collected from each benchmark by fast-forwarding 40 billion instructions, and then collected traces for the next 100 million instructions.

Table 1: Simulator Configuration.

Issue width	4
Instruction Window	128 entries
L1 cache	32KB, 8-way,I/D caches, 1 cycle
L2 cache	512KB/2MB, 16-way, 20 cycles
Memory latency	200 cycles
Configuration 1 (*c1*)	2MB L2, 1000 requests/cycle
Configuration 2 (*c2*)	2MB L2, 1 request/10 cycles
Configuration 3 (*c3*)	512KB L2, 1000 requests/cycle

Table 2: Prefetcher Configurations.

Prefetcher	Table configuration	Size
GHB-distance	256 IT entries, 256 GHB entries	4KB
Stream	16 combined entries	128B
Enhanced-Stream	8 training entries, 8 stream entries	256B

The simulation framework models an out-of-order core with the basic parameters as outlined in Table 1. Two L2 cache sizes and two memory bandwidths are considered resulting in three L2 cache configurations.

We evaluate and compare three prefetch schemes, including the PC-based Distance prefetcher using a Global History Buffer (*GHB-Distance*) [9], the original Stream prefetcher (*Stream*) [11] and the Enhanced Stream prefetcher (*Enhanced-Stream*). All prefetchers prefetch memory blocks directly into the L2 cache. The simulated table sizes for the three prefetchers are given in Table 2. Both the prefetch width and depth for GHB-distance are 16 and the prefetch degree and distance are 4 and 64, respectively, for both stream-based prefetchers. Under the allowed space budget, we simulate multiple table sizes for maintaining the stream history and selected the size that demonstrates the highest performance for both stream-based prefetchers. For achieving the best performance, the results show that both stream prefetchers require very little history information.

6. Experimental Results

6.1. Performance of Enhanced-Stream Prefetcher

Figure 7 shows the normalized CPI comparison of the three prefetching schemes where the CPIs are normalized to the baseline design without any prefetching. In this figure, the twelve selected benchmarks are sorted from left to right in the descending order of the MPKI. We can make several important observations. First, on the arithmetic average of all workloads, the performance improvements over the base CPI are 27%, 37%, and 38% for *GHB-Distance*, *Stream*, and *Enhanced-Stream* prefetchers for the *c1* configuration, 16%, 29%, and 42% for the *c2* configuration, and 26%, 44%, and 55% for the *c3* configuration, respectively. Overall, *Enhanced-Stream* outperforms *GHB-Distance* and *Stream* respectively by about 30% and 14%.

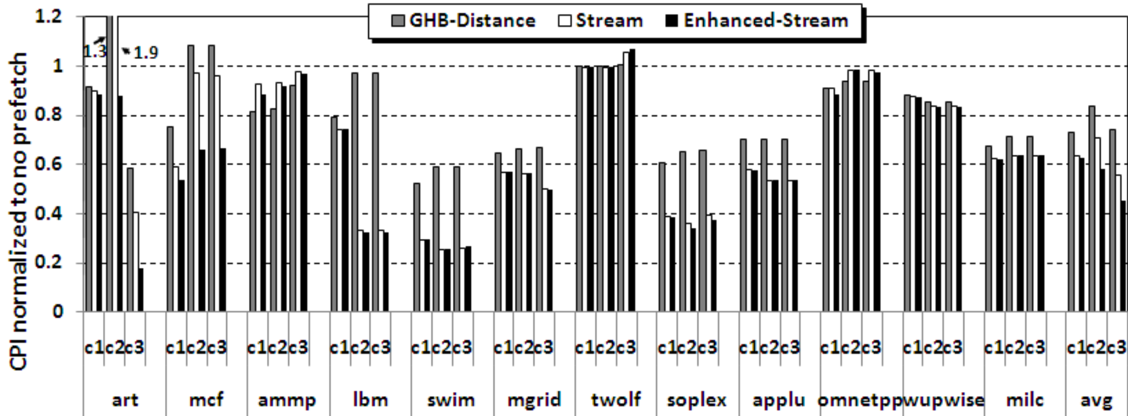

Figure 7: CPI comparisons for the three prefetching schemes.

Second, among the four enhancement techniques, stride prefetching gains the most benefit. For *art* and *mcf*, the performance gains from stride prefetching are about 45% and 28%, respectively, revealing that stream prefetching of consecutive blocks is wasteful and inaccurate in these applications.

Third, different benchmarks show very different results with respect to the three prefetching schemes. *Enhanced-Stream* is most effective for *art* and *mcf* which have the highest MPKI and most beneficial from stride prefetching. Early prefetching of repeated stream works well for *art* with about 6% improvement on 512KB L2 cache. The noise removal scheme shows some impacts on *soplex* for a minor performance gain about 1%. The dead stream removal scheme is very effective in many applications when large stream tables are used. We will show the sensitivity study results in Section 6.2.

Fouth, *GHB-Distance* performs worse than the other two schemes for most applications. However, it shows slight edge over stream-based prefetchers on *ammp* and *omnetpp*. This is due to the long and constant distances are covered in the *GHB-Distance* prefetcher.

Finally, we also observe that integrations of multiple enhancement techniques may cause performance interferences among one another. The results in Figure 7 are simulated with the combination of all four enhancement techniques.

6.2. Sensitivity Study

The sizes of the training table and the stream table impact the overall prefetch performance. In Figure 8, we collect the average CPI simulated with the three cache configurations listed in Table 1, and compare different combinations of various table sizes in *Enhanced-Stream*. For the stream table, we simulate six table sizes with 4, 8, 16, 32, 64, and 128 entries. For each stream table size, we simulate three associated training table sizes with the number of entries equal to the same, double, and quadruple of the stream table size.

The results are plotted in Figure 8, where the notation n / m1, m2, m3 represents the size of the stream table (n) and three associated training table sizes (m1, m2, m3). It is interesting to observe that the stream table of 8 entries has the lowest overall CPI indicating the number of active streams is very small in all applications. Increasing the stream table size beyond 8 degrades the performance. This is due to the fact that many inactive streams are kept in the stream table and cause inaccurate prefetching. When the number of the stream table entries reduce to 4, the insufficient space to hold all active streams reduce the overall improvement. We can also observe that the performance improvement is rather insensitive to the training table size.

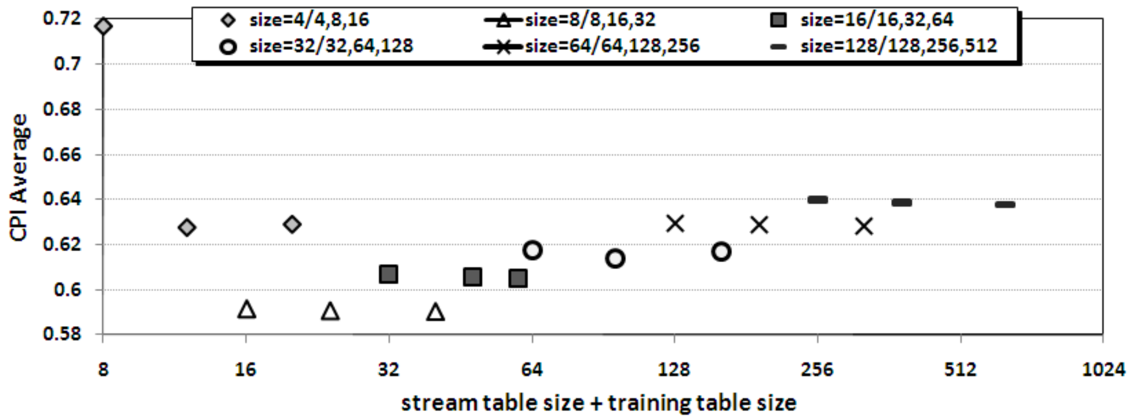

Figure 8: Sensitivity on stream history table sizes. (Notation: size = stream table size / three training table sizes).

We also evaluate the effectiveness of dead stream removal with different stream table sizes as shown in Figure 9. The left figure shows the average performance with the three cache configurations for all the workloads, with three dead stream removal techniques: no removal, removal streams with the age of 100K cycles, and removal streams with the age of 10K cycles. As can be observed, when stream table size is larger than 8, dead stream removal is effective in reducing the damage of inaccurate dead-stream prefetching. However, when the stream table size reduces to 8 or smaller, dead stream removal has very little impact. This is due to the fact that small table sizes can naturally replace dead streams dynamically.

Optimal stream table size is not always the same for individual workloads. The right figure in Figure 9 demonstrates that optimal stream table size varies from 4 to 64 for individual workloads. For example, with 16 stream entries, *swim* can improve 5% over 8 stream entries. Hence in practice, it would be a better choice to have a relatively large stream table integrated with an effective dead stream removal scheme.

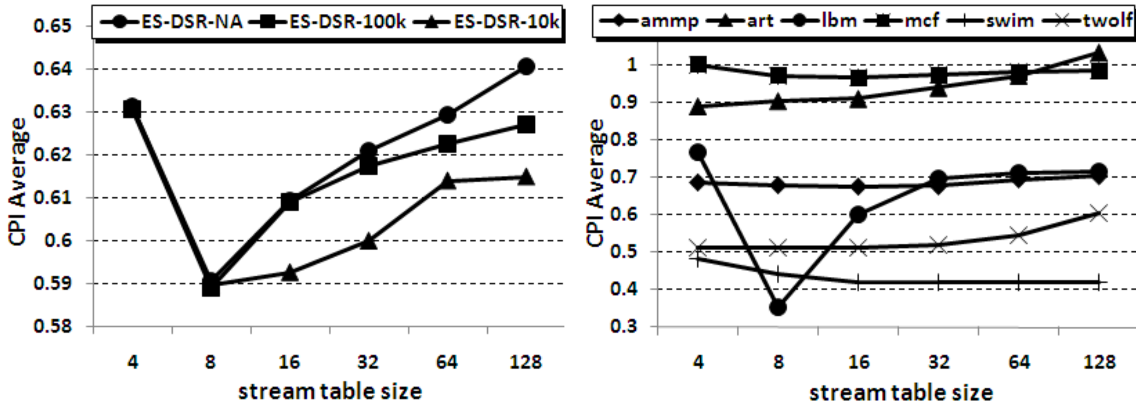

Figure 9: Overall (left figure) and individual (right figure) performance for Dead Stream Removal. *ES-DSR-NA* is without Dead Stream Removal, *ES-DSR-100k/10k* are with Dead Stream Removal and dead streams are defined as having been inactive for more than 100k/10k CPU cycles.

7. Related Work

There have been many software and hardware based prefetching methods to alleviate performance penalties on cache misses [20, 21, 22, 23, 24]. Traditional hardware-oriented sequential [4], stride [5, 6], distance [7, 8, 9], and stream [1, 10] based prefetchers work well for applications with regular cache miss patterns. These prefetchers dynamically capture the regularity of a sequence of missing block addresses to speculatively prefetch the subsequent blocks. Among them, the stream prefetcher [11, 12] has been adapted in commercial processors due to its simplicity and effectiveness. Stream prefetchers prefetch consecutive blocks according to the streaming access direction when consecutive misses within a small memory region follow the same direction. The prefetching stream continues as long as subsequent memory requests fall in the monitored region of the active streams.

However, in many modern applications and runtime environments, dynamic memory allocations and Linked Data Structures (LDS) are very common. It is difficult to accurately prefetch the LDS due to their irregular address patterns. Miss-correlation prefetchers [13, 14] record patterns of miss addresses and use the past miss correlations to predict future cache misses. To be effective, these approaches require a huge history table to record the past miss correlations. The Global History Buffer [8, 9] proposes a general FIFO structure for recording and identifying nearby missing address patterns. It can be used to implement both stride/distance based prefetchers as well as the correlation-based prefetchers. To reduce the space overhead, Tag-Correlating prefetcher extends the miss correlation to much bigger blocks [25]. Coterminous Group prefetcher [26] records and prefetches only the nearby missing blocks with equal reuse distance.

8. Conclusion

In this paper, we report enhancement techniques to improve the stream prefetcher. Based on the simulation model and workloads provided by the prefetch competition committee [19], our evaluation results show that the enhanced stream prefetcher improves 38%, 42%, and 55% of CPI for the three cache configurations listed in Table 1 with respect to the base design without prefetching. In comparison with the original stream prefetcher, the improvements are 2%, 18%, and 19% respectively. We also show that the space overhead of implementing an enhanced stream prefetcher is very small.

Acknowledgements

This work is supported in part by a research donation from Intel Corp. and by the National Science Foundation under CRI collaborative awards 0751112, 0750847, 0750851, 0750852, 0750860, 0750868, 0750884, and 0751091. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Intel Corp. and the NSF.

References

[1] N. Megiddo and D. S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm," Computer, vol. 37, pp. 58-65, August 2004.

[2] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr, and J. Emer, "Adaptive insertion policies for high performance caching," in Proceedings of 34th International Symposium on Computer Architecture, pp. 381-391, June 2007.

[3] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in Proceedings of 39th International Symposium on Microarchitecture, pp. 423-432, December 2006.

[4] K. Rajan and R. Govindarajan, "Emulating optimal replacement with a shepherd cache," in Proceedings of 40th International Symposium on Microarchitecture, pp. 445-454, December 2007.

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

[6] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," in Proceedings of 5th International Conference on Architectural Support Programming Languages and Operating Systems, pp. 51-61, October 1992.

[7] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in Proceedings of 25th International Symposium on Microarchitecture, pp. 102-110, December 1992.

[8] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "AC/DC: an adaptive data cache prefetcher," in Proceedings of 13th International Conference on Parallel Architecture and Compilation Techniques, pp. 135-145, October 2004.

[9] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in Proceedings of 10th International Symposium on High Performance Computer Architecture, pp. 96-105, Febrary 2004.

[10] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for TLB prefetching: An application-driven study," in Proceedings of 29th International Symposium on Computer Architecture, pp. 195-206, May 2002.

[11] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in Proceedings of 13th International Symposium on High Performance Computer Architecture, pp. 63-74, Febrary 2007.

[12] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," IBM Technical White Paper, Oct 2001.

[13] M. Charney and A. Reeves, "Generalized correlation-based hardware prefetching," Technical Report No. EE-CEG-95-1, Cornell University, February 1995.

[14] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in Proceedings of 24th International Symposium on Computer Architecture, pp. 252-263, June 1997.

[15] T. M. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 199-209, June 2002.

[16] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," in Proceedings of 32nd Interntional Symposium on Computer Architecture, pp. 222-233, June 2005.

[17] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in Proceedings of 33rd International Symposium on Computer Architecture, pp. 252-263, June 2006.

[18] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in Proceedings of 36th International Symposium on Computer Architecture, pp. 69-80, June 2009.

[19] The 1st JILP Data Prefetching Championship (DPC-1). Available: http://www.jilp.org/dpc/

[20] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 222-233, December 1996.

[21] B. Cahoon and K. S. McKinley, "Data flow analysis for software prefetching linked data structures in Java," in Proceedings of Internatinal Conference on Parallel Architectures and Compilation Techniques, pp. 280-291, September 2001.

[22] S. P. Vanderwiel and D. J. Lilja, "Data Prefetch Mechanisms," ACM Computing Surveys, vol. 32, pp. 174-199, June 2000.

[23] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in Proceedings of the 5th International Conference on Architectural Support Programming Languages and Operating Systems, pp. 62-73, October 1992.

[24] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided region prefetching: a cooperative hardware/software approach," in Proceedings of 30th International Symposium on Computer Architecture, pp. 388-398, June 2003.

[25] Z. Hu, M. Martonosi, and S. Kaxiras, "TCP: tag correlating prefetchers," in Proceedings of 9th International Symposium on High Performance Computer Architecture, pp. 317-326, February 2003.

[26] X. Shi, Z. Yang, J.-K. Peir, L. Peng, Y.-K. Chen, V. Lee, and B. Liang, "Coterminous locality and coterminous group data prefetching on chip-multiprocessors," in Proceedings of 20th International Parallel and Distributed Processing Symposium, pp. 69, April 2006.