Enhancement for Accurate Stream Prefetching

Gang Liu[1], Zhuo Huang[1], Jih-Kwon Peir[1], Xudong Shi[2], Lu Peng[3]

[1]Computer & Information Science & Engineering
University of Florida
Gainesville, FL 32611, USA
{galiu,zhuang,peir}@cise.ufl.edu

[2]Google Inc
1600 Amphitheater Pkwy
Mountain View, CA 94043
xdshi@google.com

[3]Electrical & Computer Engineering
Louisiana State University
Baton Rouge, LA 70803, USA
lpeng@lsu.edu

Abstract

One fundamental approach to improve memory hierarchy performance is to prefetch the needed instructions and data before their usage when cache capacity is limited. A stream prefetcher captures a sequence of nearby misses when their addresses follow the same positive or negative direction in a small memory region [17]. Once a stream of cache misses is identified, the prefetcher prefetches consecutive blocks in the same direction.

In this paper, we describe several enhancement techniques to improve weakness of existing stream-based prefetchers. First, the existing stream prefetcher does not take streams with constant stride into consideration. For long-stride accesses across more than one block, prefetching consecutive blocks is wasteful and pollutes the cache. Second, in 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 different addressing direction than the traversal direction through the nodes. Due to this noise, the stream may not be identified and the prefetch opportunity is wasted. Third, regular streams for array accesses are often repeated. Penalty occurs for re-establishing a repeated stream. Fourth, an established stream may be ended before being removed from the prefetch table. A hit to a dead stream causes inaccurate prefetches. Performance evaluations based on SPEC applications show that the enhanced stream prefetcher improves 37.6%, 41.6%, and 54.5% of CPI for the three cache configurations with respect to the base design without prefetching. In comparison with the original stream prefetcher, the improvements are 1.8%, 17.6%, and 18.7% respectively.

1. Introduction

With fast advances in processor technology, the speed gap has been continuously widening between processors and main memory. It takes hundreds of processor cycles to access the memory. Caches play a critical role in bridging this performance gap. Recently, many efficient caching mechanisms have been proposed for on-die CMP caches [15]. However, due to limited cache capacity, the required working set of applications may not fit into the cache and causes frequent accesses to the memory. Therefore, the memory access is often the bottleneck in processor performance.

Besides caches, the other fundamental approach to improve memory hierarchy performance is to prefetch the needed instructions and data before their usage. Existing data prefetching methods are based on two general behaviors of the missing block addresses: *regularity* and *correlation*. Existing sequential prefetcher [9], stride prefetcher [5], distance prefetcher [6,13] and stream prefetcher [17] dynamically capture the regularity of a sequence of the missing block addresses to speculatively prefetch the subsequent blocks. Correlated prefetcher [2], Markov prefetcher [8], and hot-stream prefetcher [4], on the other hand, records the history of nearby miss addresses and their correlations to trigger prefetches assuming such a correlation will be repeated.

A 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, the prefetcher prefetches consecutive blocks in the same direction. Two key parameters: prefetch *distance* and *degree* control the aggressiveness of stream prefetching. The prefetch distance defines the 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-based prefetching scheme has been used in commercial processors [17].

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 of more than one block, instead of prefetching consecutive blocks, the stream prefetcher prefetches blocks according to the 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 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 dead by remembering and using the stream's starting block address. Fourth, a short stream may be dead before being replaced from the stream table. Subsequent hit to a dead stream initiates prefetching without going through the needed training stage. By detecting and removing aging short streams from the stream table, it provides more accurate prefetching.

Performance evaluations based on a set of SPEC2000 and SPEC2006 applications show that the enhanced stream prefetcher makes significant improvement over the original stream prefetcher. For the three L2 cache configurations requested by the competition committee, the enhanced stream prefetcher improves 37.6%, 41.6%, and 54.5% of their CPIs with respect to the base design without prefetching. In comparison with the original stream prefetcher, the improvements are 1.8%, 17.6%, and 18.7% respectively.

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

2. Enhancement Techniques

2.1. Constant Stride Optimization

Examing *scanner.c* in Art as shown in Figure 1, we can identify constant-stride accesses that across multiple blocks. The memory references in this routine causes 80% of all misses on the baseline 1MB L2 cache without prefetching. In the memory allocation part, the size of each element of array *bus* is 88 bytes, so is the same for *tds*. Each element of two arrays *bus* and *tds* 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. In this routine, a regular stream prefetcher still prefetches consecutive blocks for accessing array *bus*. As a result, 2 out of 3 prefetched blocks are wasted. In the enhanced prefetcher, the stride in term of bytes is recorded and used to calculate the correct blocks to prefetch.

```
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++) {
    Y[tj].y = 0;
    if ( !Y[tj].reset )
      for (ti=0;ti<numf1s;ti++)
        Y[tj].y += f1_layer[ti].P * bus[ti][tj];
  }
```

Figure 1. Code segment from Art

2.2. Noise Removal

When a training stream is failed, the subsequent misses must first fetch from one direction, then turn around to the opposite direction. After careful 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). This is due to the fact that each record occupies two adjacent cache blocks. Multiple accesses to the record create a positive one distance. However, the subsequent accesses to the next record are in the negative direction. To remedy this problem, we allow a training stream to stay untrained when subsequent misses occur in the opposite direction but one of the misses in the opposite direction is accessing the adjacent block. This adjacent block access is ignored in training the stream.

2.3. Early Launch of Repeat Stream

The repetition of a stream can also be observed from *scanner.c* in Figure 1. The nested loop causes the stream of accessing array *bus* repeated for 11 times with exactly the same start and end addresses. The repetition of the stream is separated only by a few instructions. Therefore, it is beneficial to trigger the stream prefetch again from the starting address when the current stream is coming to an end. Early launching a repeated stream can reduce the penalty of initiating a new stream.

2.4. Dead Steam Removal

When a short stream is inactive for longer than 100K cycles, it is likely that the stream 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 incorrect prefetches. These prefetches may pollute the cache and cause memory bandwidth contention. Simulations reveal that given a stream history table of 128 entries, Ammp has 60.5% of 21297 trained streams stay in the stream table longer than 100K cycles without triggering any prefetch. Art has 88.2% of 6057 trained streams turned dead before being replaced. Simulation results also indicate that very few blocks prefetched by a dead stream get reused. Although a smaller table size can naturally replace streams before they die, however, smaller tables may suffer insufficient space to keep all the active streams. Hence, by removing dead streams dynamically, all active streams can be maintained without holding many dead streams.

3. Prefetcher Design

Before a stream is established for prefetching in a stream prefetcher, it must be qualified through a *training* stage. A stream is trained when three cache misses are located in a small memory region (called a training window or region) and their addresses follow the same positive or negative direction. Afterwards, when another memory request falls into the monitored region of a trained stream, it triggers the prefetches of the subsequent blocks. The monitored region is defined by the prefetch *distance* and the number of consecutive prefetched blocks is referred as the *degree* of prefetch. A *training* table and a *stream* table are used to record the streams in the respective training and prefetching stages. The detailed operations of these two stages and the contents of the two tables are described in the following subsections.

3.1. Training

The procedure of training a stream works as follows.

1. For accommodating the stream training, each entry in the *training* table consists of a 26-bit starting *block address* to record the first miss address in the region, and a 5-bit *distance* to record the distance (blocks) from the second miss to the first miss in the region. In our experiment, the training window has 32 blocks.
2. When a cache miss occurs, it first looks for an entry in the training table whose starting address is within positive or negative 16 blocks, i.e. falling into the training window of an existing stream. If none is found, create a new entry for the missing block.
3. If a match is found and the entry has not been marked by a second miss, the entry is then marked by recording the distance in terms of the number of blocks from the first miss.
4. When a miss matches a marked entry, the third miss is identified within the training window. In this case, there are three conditions.
   a. If the third miss is in same direction as the second miss, the stream is trained and moved to the *stream* table. The stream's monitored region is initialized in the stream table ready for prefetching the stream.
   b. If the third miss is in the opposite direction from the second miss, but the distance between the first and the second miss is a single block, then ignore the second miss address, and re-establish the distance from the first miss using the third miss for continuing the training. In this case, the second miss is treated as a *noise*.
   c. If the third miss is in the opposite direction from the second miss, and the distance between the first and the second miss is not a single block, then remove the first miss from the entry. The second and the third misses are treated now as the new first and second misses for continuing the training.

3.2 Prefetching

The procedure of enhancing the stream prefetching consists of the following steps.

1. An entry is created when a trained stream is moved from the training table to the *stream* table including the following information.
   a. *Start block address* (26 bits): block address of the first miss that initiates the stream.
   b. *End address* (32 bits): the ending byte address of the stream monitored region, i.e. the last prefetching address. Note that byte address is necessary for stride prefetching.
   c. *Last address* (16 bits): the distance from the last access to the ending address. This distance is used to figure out the stride.
   d. *Direction* (1 bit): the direction of the stream.
   e. *Stride* (9 bits): stride distance up to 8 blocks.
   f. *Stride-enable* (1 bit): stride or stream.
   g. *Repeat* (1bit): indicator of a repeated stream.
   h. *Timestamp* (32 bits): the time of last prefetch
2. All L2 requests look up the *stream* table for triggering stream or stride prefetching. In case a match is found in a monitored region of a stream, it prefetches the subsequent blocks based on the prefetch *distance, degree, and stride*. In this experiment, we detect prefetches in a 64-block region with the prefetch degree of 4.
3. After prefetching 4 blocks, the monitored region is moved forward by 4 blocks in the prefetching direction similar to the original stream prefetcher.
4. The default of stream prefetcher is to prefetch consecutive blocks. However, constant stride over one block can be detected dynamically and used for stride prefetching. The stride distance of the current access is always saved in each entry of the stream table. In case the next stride matches the previous stride, a *stride-enable* indicator is turned on and the subsequent prefetches will be based on the recorded *stride*. However, whenever a new stride mismatches the recorded stride, the stride-enable is turned off and the

prefetch is reset to the default prefetching of consecutive blocks.

5. To detect *repeated* streams, a search to the stream table is carried out whenever a stream monitor region is forwarded after prefetches. The detection and early prefetching of repeated stream works as follows. A *timestamp* is inserted for each stream in the stream table. The timestamp indicates the time when the last prefetch was triggered by the respective stream. When the forwarded window overlaps with monitored region of another stream, a *repeated* stream is discovered under three conditions. First, the timestamp of the overlapped stream must be sufficiently old. Second, the two streams must have their starting addresses closely to each other. Third, the length of both streams must be sufficiently long. Upon detecting a repeated stream, prefetching of the initial 4 blocks is triggered without any delay. Meanwhile, a *repeat* indicator is turned on for updating the starting address for the repeated stream when the address becomes available.
6. Finally, for early removal of a dead stream, the stream table is searched periodically. A dead stream is identified when its timestamp shows the stream has not triggered any prefetch for a long time and the length of the stream is very short. Upon discovery, the dead streams are removed from the stream table.

4. Evaluation Methodology

To demonstrate the advantages of the enhanced stream prefetcher, we selected twelve benchmarks with high L2 Misses-Per-thousand-Instructions (MPKI) from SPEC2000 and SPEC2006. Trace-driven simulations were carried out using the CMPsim tool set provided by the First JILP Data Prefetching Championship competition committee [21]. 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

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 as requested by the competition committee.

We evaluate and compare three prefetch schemes, including the PC-based Distance prefetcher using a Global History Buffer (*GHB-Distance*), the original Stream prefetcher (*Stream*) and the Enhanced Stream prefetcher (*Enhance-Stream*). All prefetchers prefetch memory blocks directly into the L2 cache. The simulated table sizes for the three prefetchers are listed in Table 2. Both the prefetch width and depth for *GHB-distance* are 16. The prefetch degree and distance are 4 and 64 respectively for both stream-based prefetchers. Under the allowable 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.

Table 2. Prefetcher Configurations

Prefetcher	Table configuration	Size
GHB-distance	256 IT entries, 256 GHB entries	4KB
Stream	8 combined entries	64B
Enhance-Stream	16 training entries, 8 stream entries	256B

5. Experimental Results and Analyses

Figure 2 shows the CPI comparisons of the three prefetching schemes where the CPIs are normalized to the base CPI without any prefetching scheme. 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 an average of all workloads, the performance improvements over the base CPI are 26.8%, 36.5%, and 37.6% for *GHB-Distance, Stream,* and *Enhance-Stream* prefetchers under the *c1* configuration, 16.2%, 29.1%, and 41.6% for the *c2* configuration, and 26.0%. 44.1% and 54.5% for the *c3* configuration, respectively. Overall, *Enhance-Stream* outperforms *GHB-Distance* and *Stream* by about 30.3% and 14.1%.

Second, different benchmarks show very different results with respect to the three prefetching schemes. *Enhance-Stream* is most effective for Art and Mcf which have the highest MPKI. Detailed analysis shows that both Art and Mcf benefit stride prefetching significantly. Third, *GHB-Distance* performs significantly worse than the other two schemes for most applications. However, it has slight edge on Ammp and Omnetpp. Fourth, among the four enhancement techniques, stride prefetching gains the most benefit. For Art and Mcf, the performance gains from stride prefetching are about 44.6% and 27.6% respectively. Noise removal is effective for Soplex showing a performance gain about 1%. Early prefetching of repeated stream works well for Art with about 2.2% improvement. The dead stream removal is very effective

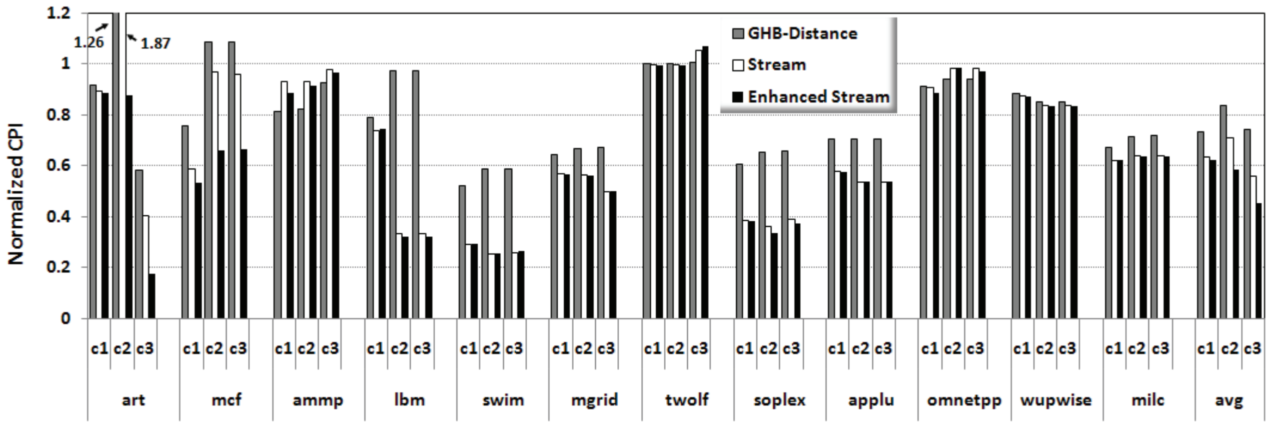

Figure 2. CPI comparisons for the three prefetching schemes

in many applications. We also observe that integrations of multiple enhancement techniques may cause interferences with one another in different applications. The results of *Enhance-Stream* are simulated with the combination of all four techniques.

The sizes of the training table and the stream table impact the overall prefetch performance. In Figure 3, we show the overall average CPI of all three configurations of different table sizes used in *Enhance-Stream.* For the stream table, we simulate six table sizes with 4, 8, 16, 32, 64, and 128 entries. For each stream table size of *n* entries, we simulate three training table sizes with the number of entries equal to *n*, 2*n*, and 4*n*. 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 can degrade 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. The performance improvement is rather insensitive to the training table size. In general, the training table that has twice as much entry than that in the stream table shows the best overall performance.

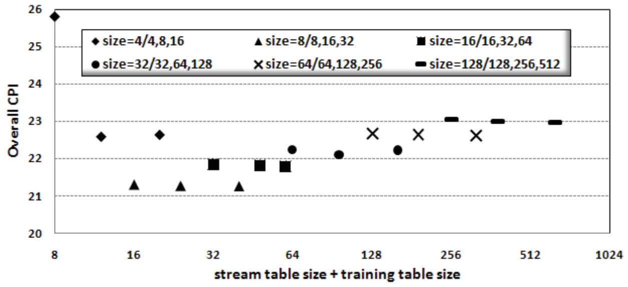

Figure 3. Sensitivity on stream history table sizes; (*notation:* size = stream table size / three training table sizes)

6. Related Work

There have been many software and hardware oriented prefetching proposals to alleviate performance penalties on cache misses [11,19,10,1,20]. Traditional hardware-oriented sequential or stride-based prefetchers work well for applications with regular memory access patterns [3,9]. Stream prefetchers [14,17,18] are widely used in industry due to its simplicity and effectiveness. Stream prefetchers prefetch sequential blocks after misses in the same direction are detected. The prefetching stream continues as long as accesses fall on its monitored region. However, in many modern applications and runtime environments, dynamic memory allocations and Linked Data Structure (LDS) accesses are very common. It is difficult to accurately prefetch the LDS due to their irregular address patterns. Correlated and Markov prefetchers [2,8] record patterns of miss addresses and use the past miss correlations to predict future cache misses. These approaches require a huge history table to record the past miss correlations. Global History Buffer [12] can efficiently use table size and keep history fresh. It can be used to implement both stride/distance based prefetchers and the correlated prefetchers. A tag-correlating prefetcher uses much bigger block correlations to reduce the history size [7]. Besides the history size, correlation-based prefetchers also face challenges in providing accurate and timely prefetches. A memory-side correlation-based prefetcher is presented in [16] where the history size becomes less of a problem by moving it to memory. To handle timely prefetches, a chain of prefetches based on pair-wise correlation history can be triggered. Accuracy and memory traffic, however, remain difficult issues.

7. 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, our evaluation results show that the enhanced stream prefetcher improves 37.6%, 41.6%, and 54.5% of CPI for the three cache configurations with respect to the base design without prefetching. In comparison with the original stream prefetcher, the improvements are 1.8%, 17.6%, and 18.7% respectively. We also show that the space overhead of implementing an enhanced stream prefetcher is very minimum.

Acknowledgement

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] B. Cahoon, K.S. McKinley, **“**Data Flow Analysis for Software Prefetching Linked Data Structures in Java”, *Proc. of 10th PACT*, 2001.

[2] M. Charney and A. Reeves. “Generalized Correlation Based Hardware Prefetching,” *Technical Report EE-CEG-95-1,* Cornell University, February 1995.

[3] T. Chen, J. Baer, “Reducing Memory Latency Via Non-Blocking and Prefetching Caches,” *Proc. of 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems,* Oct. 1992, pp. 51-61.

[4] T.M. Chilimbi, “On the Stability of Temporal Data Reference Profiles. *Proc. of 8th PACT,* Sept. 2002.

[5] J. Fu, J. Patel, B. Janssens: Stride directed prefetching in scalar processors. MICRO 1992: 102-110.

[6] G. Handiraju and A.Sivasubramaniam,”Going the Distance for TLB Prefetching: An Application-driven Study”, *Proc. of 29th ISCA*, June 2002.

[7] Z. Hu, M. Martonosi, S. Kaxiras, “TCP: Tag Correlating Prefetchers,” *Proc. of 9th Ann Int’l Symp. on HPCA*, Feb 2003.

[8] D. Joseph, and D. Grunwald, “Prefetching Using Markov Predictors,” *Proc. of 26th ISCA*, Jun 1997.

[9] N. P. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” *Proc. of 17th ISCA,* May 1990.

[10] C. Luk, T. C. Mowry, “Compiler-Based Prefetching for Recursive Data Structures”, *Proc. of 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems,* Boston, MA, Oct. 1996, pp. 222-233.

[11] T. C. Mowry, M. S. Lam, A. Gupta, “Design and Evaluation of a Compiler Algorithm for Prefetching”, *Proc. of 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems,* Oct. 1992.

[12] K. Nesbit and J. Smith, “Data Cache Prefetching Using a Global History Buffer,” *Proc. of 10th HPCA,* Feb. 2004.

[13] K. Nesbit, A. Dhodapkar and J. Smith, “AC/DC: An Adaptive Data Cache Prefetcher”, *Proc. of 10th PACT,* Sept. 2004.

[14] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. *Proc. of 21st ISCA*, June 1994.

[15] M. Qureshi A. Jaleel, Y. Patt, S.Steely, J. Emer, “Adaptive Insertion Policies for High Performance Caching”, *Proc. of 34th ISCA*, June 2007

[16] Y. Solihin, J. Lee, and J. Torrellas, “Using a User-Level Memory Thread for Correlation Prefetching,” *Proc. of 29th ISCA*, May 2002.

[17] Santhosh Srinath, Yale N. Patt, “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. *Proc. of 13th Ann Int’l Symp. on HPCA*, Feb 2007.

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

[19] S. Vanderwiel, and D. Lilja, “Data Prefetch Mechanisms,” *ACM Computing Surveys*, June 2000.

[20] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt and C. C. Weems, “Guided Region Prefetching: a Cooperative Hardware/Software Approach,” *Proc. of 30th ISCA*, June 2003.

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