Feedback Mechanisms for Improving Probabilistic Memory Prefetching

Ibrahim Hur[†]

Calvin Lin[‡]

[†]IBM Corporation
Systems and Technology Group
Austin, TX
ibrahur@us.ibm.com

[‡]The University of Texas at Austin
Department of Computer Sciences
Austin, TX
lin@cs.utexas.edu

Abstract

This paper presents three techniques for improving the effectiveness of the recently proposed Adaptive Stream Detection (ASD) prefetching mechanism. The ASD prefetcher is a standard stream buffer that takes a probabilistic feedback-based probabilistic approach to identifying streams. Its strength lies in its ability to effectively prefetch streams that are as short as two consecutive cache lines, which allows it to exploit spatial locality even for programs that have irregular access patterns.

The first technique improves a stream buffer's ability to detect short streams, which significantly increases the potential of stream-based prefetching. For example, for the SPECFfp milc benchmark, this new technique doubles the number of detectable streams from 33% to 67%. The second technique improves the quality of the ASD prefetcher's feedback mechanism by adaptively adjusting the epoch *length—the length of time used to represent the recent past behavior—according to a simple similarity metric. The third technique improves the timing of prefetches by supporting variable-length prefetching of multiple cache lines.*

Collectively, these techniques almost double the effectiveness of the ASD prefetcher, improving the performance of the ASD prefetcher by 11.2% for the SPECfp benchmarks, by 10.3% for the NAS benchmarks, and by 13.2% for a set of commercial benchmarks that exhibit poor spatial locality. The improved performance in turn decreases DRAM energy consumption by 7.3%, 8.3%, and 9.4%, respectively, for the same three benchmark suites.

1. Introduction

As DRAM access times continue to grow relative to processor cycle time, latency-hiding techniques such as prefetching continue to grow in importance. One common commercially used prefetching mechanism is the stream buffer [12], which works well when programs exhibit large amounts of spatial locality in the form of sequentially accessed streams of data. Stream buffers have historically been biased towards long streams, because a useless prefetch operation is required to terminate the prefetching of a particular stream. Adaptive Stream Detection (ASD) is a recent improvement to the stream buffer [9] that increases prediction effectiveness by keeping a histogram of the lengths of recently detected streams. Given these histograms—known as Stream Length Histograms (SLHs)—the ASD prefetcher probabilistically determines whether to prefetch the next line of particular stream, which allows it to stop prefetching without incurring a useless prefetch. Thus, the ASD prefetcher can efficiently prefetch streams that are as short as two cache lines, which makes the technique profitable for irregular applications such as commercial workloads, because even these irregular applications exhibit locality in the form of very short streams.

In this paper, we significantly improve upon the ASD stream buffer through the use of three techniques.

First, to improve upon the stream buffer's basic stream detection mechanism, which has remain unchanged since Jouppi's first introduction, we introduce Length-Based Stream Detection. Length-Based Stream Detection removes the original stream buffer's bias towards long streams by decreasing the lifetime—that is, the amount of time that we are willing to wait for the next element of a stream—as the length of a stream increases. With this new detection mechanism, we find that most irregular applications exhibit a significantly larger number of short streams than was previously thought. For example, for the SPECfp milc benchmark, the number exploitable streams—those of length 2 or greater—grows from the previously reported 33% [9] to about 67%.

Second, we introduce the notion of Adaptive Epoch Lengths, which improves the effectiveness of the SLH feedback mechanism. Because SLHs vary over time—even for a given application—each SLH is computed for one time period, known as an *epoch*, and used in the next epoch. Previous results used a fixed epoch length, but we show that an adaptive epoch length improves performance by an average of about 5% for a set of benchmarks that includes representative subsets of the SPECfp benchmarks, the NAS benchmarks, and a set of IBM commercial benchmarks.

Third, we evaluate the concept of Variable-Length Prefetching, which uses the SLH information to prefetch multiple lines at a time. This idea was suggested previously [9] but never evaluated. We show that for this same set of benchmarks, Variable-Length Prefetching improves performance by an average of about 5%.

To summarize, this paper makes the following contributions.

- We introduce two new techniques that enhance the ASD prefetching approach, and we evaluate a third technique that had been proposed but never evaluated. While two of these techniques are specific to the ASD prefetcher, the other, Adaptive Epoch Lengths, defines a state machine that can be used by any epoch-based technique that determines current behavior based on a model of the recent past behavior.

- We evaluate our techniques using the SPEC2006 floating point suite, the NAS benchmarks, and a set of five commercial benchmarks. Using a detailed simulator for the IBM Power5+, we evaluate our new prefetcher as it would reside in the memory controller. We find that our new techniques almost double the performance improvements of the original ASD prefetcher. When compared with a stripped down Power5+ that has its processor-side prefetcher turned off, our enhanced ASD prefetcher improves performance on our three benchmark suites by 23.1%, 20.7%, and 21.4%, respectively.

- We evaluate the collective DRAM power and energy impact of our new techniques. For the three benchmark suites, we find that our enhanced ASD prefetcher increases DRAM power consumption—relative to the Power5+—by 1.5%, 3.2%, and 2.9%, respectively, while it decreases DRAM energy consumption by 7.3%, 8.3%, and 9.4%.

This paper is organized as follows. The next section places our work in the context of prior work. Section 3 describes the original ASD prefetcher, while Section 4 describes our enhancements. We present our experimental methodology in Section 5 before presenting our empirical evaluation in Section 6 and then concluding.

2. Related Work

Stream buffers [12] are a logical extension of next-line prefetching [22]. Over the years, the model of a stream has been enhanced by adding non-unit strides [19], by predicting strides [2, 7], and by supporting irregular strides using Markov predictors [11, 21]. The efficiency of stream prefetching has been improved by Nesbit and Smith [18], who introduce the *Global History Buffer* to improve prefetch effectiveness and reduce table sizes. None of these prefetchers successfully exploits low amounts of spatial locality. Instead of devising more complex models of streams, our work attempts to exploit the basic model more effectively.

Another line of research focuses on detecting and exploiting spatial locality without tracking individual streams [10, 14, 16, 5]. Instead, variations of the *Spatial Locality Detection Table*, introduced by Johnson *et al.* [10], track accesses to individual regions of memory so that spatially correlated data can be prefetched together. A problem with these approaches is the need for large tables to detect locality. Somogyi *et al.* [24] show that much smaller tables can be used by correlating spatial locality with the program counter in addition to parts of the data address. As a result, *Spatial Memory Streaming* can decrease table sizes to 64KB. Of these techniques, only Somogyi *et al.* have demonstrated success with commercial workloads, in particular, showing dramatic improvements for one benchmark. By contrast, Adaptive Stream Detection cannot prefetch as aggressively across irregular locality patterns but instead attempts to use a much smaller amount of hardware to prefetch the very small streams that likely make up these larger patterns.

Wenisch *et al.* [28] introduce *Streamed Value Buffer* to exploit temporal locality. Our approach is different and simpler to implement.

Our Variable-Length Prefetching is not the first to use feedback to modulate the aggressiveness of the prefetcher. *Scheduled Region Prefetching (SRP)* [15] prefetches large regions of memory, such as 4KB at a time, and uses the state of the system to reduce the opportunity cost of prefetches. Prefetches to open banks are given priority, and prefetched data are brought into the LRU position of the L2 sets. In addition, prefetched commands are given low priority in the memory controller. In particular, the SRP prioritizer issues prefetch commands only if the channels are idle and there is no pending request from the L2 cache. One issue with SRP is the high memory bandwidth pressure that it incurs because of its large regions. Wang *et al.* [27] solve this problem by using the compiler to help select the region size. Our solution instead uses a modest amount of hardware to prefetch at a much finer granularity.

Any instantaneous measure of utilization, such as whether a channel is idle, can be misleading because it does not consider the larger behavior of the system. Srinath *et al.* [25] address this issue by devising methods of estimating a prefetcher's accuracy, its timeliness, and its impact on cache pollution and showing how such feedback can effectively modulate the aggressiveness of the prefetcher.

Finally, others have studied memory-side prefetching [1, 4, 29, 30, 23] and have shown that memory-side prefetching is largely orthogonal to processor-side prefetching [4, 8]. Unlike our approach, previous methods do not monitor the status of the memory system, so they can increase latencies for regular memory commands.

3. Background

This section briefly describes the memory controller of the IBM Power5+, the original Adaptive Stream Detection (ASD) prefetcher [9], and the implementation of the ASD prefetcher on the Power5+.

3.1. The Power5+ Memory Controller

As shown in Figure 1, the Power5+ memory controller resides between the L2/L3 caches and DRAM. As memory commands enter the memory controller, they are placed in Reorder queues. On each cycle, the scheduler selects a command from the Reorder queues, which is then sent to the Centralized Arbiter Queue (CAQ), which in turn transmits commands to DRAM in FIFO order. Note that the Power5+'s processor-side prefetcher emits commands that bring data into the L2 and L1 caches, and these commands appear in the memory controller indistinguishable from any other command.

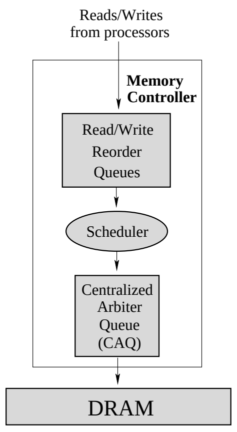
Reads/Writes from processors
Memory Controller
Read/Write Reorder Queues
Scheduler
Centralized Arbiter Queue (CAQ)
DRAM

Figure 1. The IBM Power5+ Memory System.

3.2. Prefetching Using Adaptive Stream Detection

The ASD prefetcher [9] uses Stream Length Histograms, SLHs, to capture spatial locality and guide prefetch decisions. Figure 2 shows an example SLH for the GemsFDTD benchmark from the SPEC2006 suite, where the height of the bar at location m represents the number of Read commands that are part of a stream of length m. Depending on the detected stream length of the current Read request, the prefetcher checks the SLH and determines whether to prefetch the next cache line.

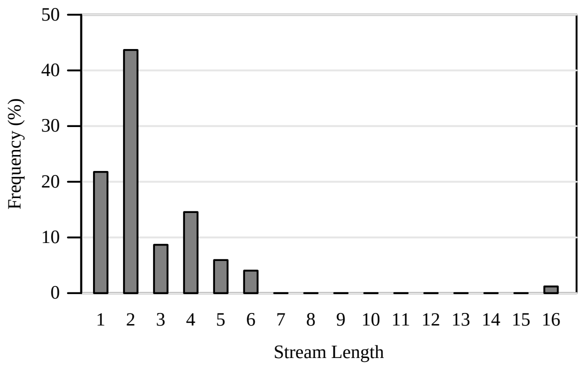

Figure 2. Stream Length Histogram (*SLH*) for an arbitrary epoch of the GemsFDTD benchmark.

The SLH of Figure 2 shows that 21.8% of all Read requests belong to streams of length 1 and that 43.7% of the Reads are part of streams of length 2. Thus, when a Read request, R_n, arrives and is the first element of a new stream, a prefetch request should be issued because R_n is more likely to be part of a stream of length 2 or longer (78.2% probability) than to be part of a stream of length 1 (21.8%). On the other hand, if R_n were the second line of a stream, a prefetch should *not* be issued because there is a 43.7% probability that R_n is part of a stream of length 2, which is greater than the 34.5% likelihood that it is part of a longer stream (34.5% $=$ 100% $-$ 21.8% $-$ 43.7%).

3.2.1. Making Prefetch Decisions

To determine whether to issue a prefetch for the next line, the ASD prefetcher checks whether the following condition is satisfied for a Read request, R_n, that is the i^{th} element of a stream:

$$P(i,i) < P(i+1, fs) \tag{1}$$

where $P()$ is defined as follows: $P(i,j)$ is the sum of probabilities that a Read is part of any stream of length k, where

$i \leq k \leq j$ and $1 \leq i, j \leq fs$, and where fs is the longest stream that we track. To simplify a subsequent proof and the hardware implementation, we define $P(i, j)$ in terms of $lht()$ as follows:

$$P(i, j) = \frac{lht(i) - lht(j+1)}{lht(1)} \quad (2)$$

where $lht(i)$ is the number of Reads that are part of streams of length i or longer, where $1 \leq i \leq fs$ and fs is the longest stream that is tracked. For any $i > fs$, $lht(i) = 0$.

Note that the value of the i^{th} bar of an SLH equals $P(i, i)$.

3.2.2. Prefetcher Design

Because memory access behavior typically varies over time, the ASD prefetcher creates a new SLH after every e Read requests, where e is known as an epoch. Thus, every epoch constructs an SLH for use in the next epoch.

Figure 3 shows the implementation of the ASD prefetcher as a memory-side prefetcher inside the Power5+ memory controller, with the gray boxes representing the ASD's additions to the memory controller. Read commands enter the memory controller and are sent to both the original memory controller and to the Stream Filter. The Stream Filter keeps track of Read streams and generates the SLH. This information from the Stream Filter is then fed to the Prefetch Generator, which decides whether a prefetch command should be issued, and if so, places the prefetch command in the Low Priority Queue (LPQ), where the Final Scheduler can consider it, along with other commands in the LPQ and CAQ, when selecting commands to issue to DRAM. Any prefetched data are then stored in the Prefetch Buffer.

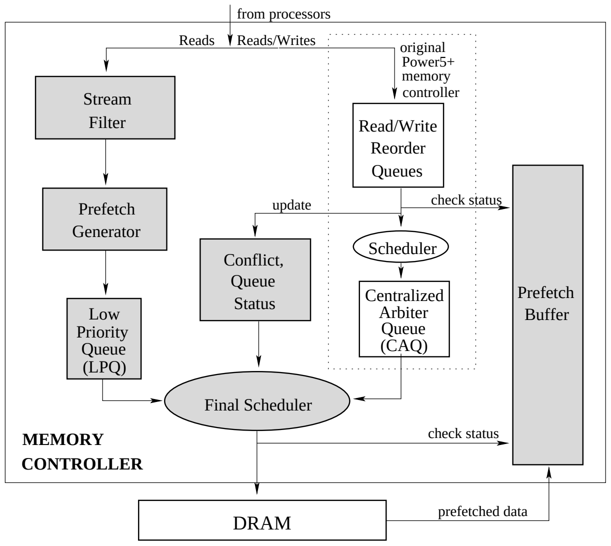

Figure 3. Overview of the ASD prefetcher.

4. Our Enhancements

In this section, we present our enhancements to the ASD prefetcher.

4.1. Length-Based Stream Detection

Length-Based Stream Detection improves the effectiveness of the stream detection mechanism, which operates as follows. Stream buffers detect streams based on a *stream lifetime*, which is the number of cycles that the Stream Filter will wait for the next element of its given stream. If the next element arrives in time, the Stream Filter's associated counter, t is reset to the stream lifetime; otherwise, the Stream Filter is free to be allocated to a new stream. The stream lifetime thus represents a tradeoff. If the lifetime is too short, it will underestimate a stream's length, so it will not prefetch all lines of a stream. If the lifetime is too long, it needlessly ties up the stream buffer, preventing the detection of other streams.

While existing stream buffers use a fixed stream lifetime, Length-Based Stream Detection decreases the stream's lifetimes as the stream's detected length becomes longer. In particular, for every unit increase in the stream length, the reset lifetime is reduced by half. For example, we use t cycles for streams of length 1, $t/2$ for streams of length 2, and $t/4$ for streams of length 3.

This policy may seem counter-intuitive, but the idea is to give the prefixes of streams a greater chance to be prefetched, which has the side effect of giving shorter streams a greater chance to be fully prefetched. The rationale is that the shorter the actual stream length, the greater the penalty of underestimating the stream length as a percentage of prefetch potential. For example, in the best case, streams of length 2 will have their second cache line prefetched (a stream buffer never prefetches the first line of a stream), but if the stream buffer were to always underestimate the length of such streams, it would not prefetch any cache lines from these short streams, thereby squandering the opportunity to prefetch 50% of these streams' cache lines. By contrast, if a stream buffer were to underestimate the length of a 5-line stream by one cache line, it would prefetch 3 of the 4 prefetchable cache lines, so the penalty for underestimating the stream length would be just 25% of the prefetchable cache lines. In general, as we detect longer streams, we are moving to the right on the SLH, so the likely payoff of holding onto the stream buffer decreases. More precisely, by definition of $P()$ in Section 4.3.1, for any two streams s_a and s_b, with lengths a and b, $a < b$, the probability of a new Read request being the next element of

s_a, $P(a+1, fs)$, is greater than the probability of it being the next element for s_b, $P(b+1, fs)$.[1]

4.2 Adaptive Epoch Length

The effectiveness of the ASD prefetcher depends on the faithfulness of the SLH—which was computed in the previous epoch—to the behavior of the data accesses in the current epoch. This subsection explains how we adaptively adjust the epoch length to produce SLHs that faithfully represent the current behavior.

The original ASD prefetcher uses a fixed epoch length for all applications, where the length is defined in terms of the number of Read requests. The epoch length is important to performance: If it is too short, the SLH becomes overly sensitive and small changes in memory access behavior can produce consecutive SLHs that are significantly more dissimilar than they should be; if the epoch length is too long, the prefetcher becomes too insensitive and may miss the fine details of the application's memory access behavior.

To adjust the epoch length according to program behavior, we first define a metric that indicates whether two SLHs are similar. We then define a state machine for changing the epoch length based on the similarity of the two most recent epochs. We now describe these two components in turn.

Similarity Metric. To determine whether the SLHs of the previous two epochs are similar, we first normalize the SLHs such that each represents the same number of Read commands. We then compare corresponding entries of the two SLHs to compute the average differences. More specifically, we sum the absolute differences of corresponding SLH entries and divide this sum by the normalized number of Read commands. If this average difference is smaller than some threshold, then the two SLHs are considered to be similar and the epochs are said to have similar behavior; otherwise the SLHs are considered to be dissimilar.

Determining the Epoch Length. At the beginning of each epoch, the similarity of the two most recent SLHs is checked, and the epoch length for the current epoch is modified according to the state machine given in Figure 4. Since the computation of epoch length is performed only once in an epoch, its cost is negligible.

In the state machine, each state indicates how the epoch length should be modified. Transitions among the states are labeled with the outcome of the similarity test: s indicates similar SLHs and d indicates dissimilar SLHs. The states have one of three labels. The states labeled "same" are *good* states, meaning that the state machine has found a desirable epoch length. The states labeled $2\times$ will double the epoch length, and those labeled $1/2\times$ will cut the epoch length in half.

[1]Length-Based Stream Detection might appear to adversely affect programs with long streams, but we have not seen this in our benchmarks. For example, results for daxpy show no performance difference at all.

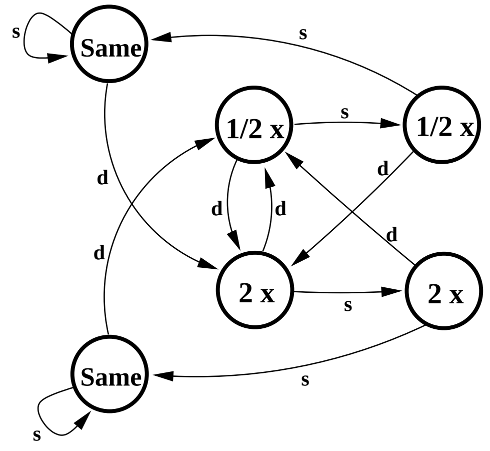

Figure 4. State diagram for SLH epoch length computation. $2\times$ doubles the epoch length, $1/2\times$ reduces it by half, $Same$ doesn't change it. Transitions are performed depending on the similarity of the previous two SLHs, s:similar, d:dissimilar.

The basic idea of the state machine is as follows.

- If the state machine is in a good state and the two most recent epochs have similar behavior, then the state machine remains in the same good state. We define a good state to be one which follows two consecutive pairs of similar epochs.
- The state machine maintains two distinct good states, so that when it detects a phase change as indicated by a pair of dissimilar epochs, it will know whether to increase or decrease the epoch length. If it had arrived at the good state by previously increasing the epoch length, it now decreases the epoch length, and vice versa. Without this affinity to change directions, the state machine would attempt to either always increase or always decrease the epoch length upon detecting dissimilar SLHs.
- If the state machine decreases (increases) the epoch length from a good state with a positive result, i.e., the next pair of epochs have similar behavior, then it again decreases (increases) the epoch length. Note that two consecutive successful decreases (increases) are needed to transition into a good state. If we allowed

just a single positive result to transition into a good state, then the epoch length would toggle between just two distinct epoch lengths.

- If the previous modification to the epoch length produced a negative result, i.e. dissimilar behavior, then the state machine changes directions. The state machine might now toggle between two bad states, but as soon as a phase change occurs that creates an appropriate-sized epoch length, the epoch length will move in the right direction to a good state.

- Finally, we set minimum and maximum values for the epoch lengths. Any action that would cause the epoch lengths to exceed these bounds is ignored. For our results, we set these bounds to be 256 and 8K memory commands.

4.3. Variable-Length Prefetching

We now explain how Variable-Length Prefetching can improve the timing of prefetch requests. If a line is prefetched too early, it can be overwritten in the prefetch buffer by other prefetched data, generating a useless prefetch that unnecessarily occupies the LPQ, the command bus, the data bus, and the DRAM banks. If the line is prefetched so late that the prefetch command is issued to DRAM after the actual memory request is issued, then the prefetch is useless and is squashed in the LPQ. This second problem occurs when memory requests arrive in bursts at the memory controller. By generating up to m consecutive prefetch requests at a time, multiline prefetching can reduce the occurrence of late prefetches. The key to profitability is to initiate multiline prefetches only when a burst of m consecutive memory requests is expected.

The next subsection shows how the prefetch decision logic can be modified to support multiline prefetching in anticipation of bursty requests. In addition, we can modulate the aggressiveness of multiline based on the occupancy of the Read Reorder Queue: If it is at least half full, then we suppress multiline prefetching because the memory controller is likely to be too busy. We refer to this suppression as a Queue Status Check, and we evaluate its impact in Section 6.2.

4.3.1. Prefetch Decision for Multiple Lines

The original ASD prefetcher decides whether to prefetch the next line by comparing the likelihood that a Read request will be the last element of a stream against the likelihood that it will be part of a longer stream. To support the prefetching of m lines at a time, we extend (1) as follows:

$$P(i, i+s-1) < P(i+s, fs), \forall s \in [1, m] \qquad (3)$$

which states that for all s, $1 \le s \le m$, the probability that the most recent Read request, R_n, is part of a stream of length between i and $i+s-1$ is smaller than it being a part of a stream of length longer than $i+s-1$. We can simplify inequality (3) as follows:

$$P(i, i+s-1) < P(i+s, fs), \forall s \in [1, m] \qquad (4)$$

$$\equiv \frac{lht(i) - lht(i+s)}{lht(1)} < \frac{lht(i+s) - lht(fs+1)}{lht(1)} \qquad (5)$$

$$\equiv lht(i) < 2 \times lht(i+s), \forall s \in [1, m] \qquad (6)$$

Our extended ASD prefetcher uses inequality (6) to make prefetch decisions for m lines.

5. Methodology

We now describe our simulation methodology, our simulated system, and the benchmarks that we use to evaluate our techniques.

5.1 Simulation Methodology

To evaluate performance, we use a cycle-accurate simulator for the IBM Power5+, which has been verified to within 1% of the performance of the actual hardware. This simulator, one of several used by the Power5+ design team, uses execution traces to simulate both the processor and the memory system. To simulate our benchmarks, which have billions of dynamic instructions, we use uniform sampling, taking 50 uniformly chosen samples that each consist of 2 million instructions. The Power5+ simulator is integrated with Memsim [20], a DRAM simulator that jointly models the power and performance of the main memory subsystem. Memsim models all the memory system activity, including refreshes, while synchronizing with the Power5+ simulator on every processor cycle.

5.2. Simulated System

The Power5+ [6, 13] has one memory controller and two processors per chip, where each processor supports two SMT threads and has split L1 D and I caches. The chip has a unified L2 cache shared by the two processors, along with an optional L3 cache. Our simulator models all three levels of the cache. The L1D cache is 32KB with 4-way set associativity and the L1I cache is 64KB with 2-way set associativity. The L2 cache is a 3×640KB shared cache, with 10-way set associativity and a line size of 128B. The off-chip L3 cache is 36MB. We simulate the DDR2 SDRAM chips running at 533MHz and the Power5+ running at 2.132GHz.

The Power5+ memory controller has two ports to memory. Each port is connected to memory via Synchronous

Memory Interface (SMI) chips [26]. We evaluate our techniques on a configuration with 4 SMIs and DDR2 SDRAM running at 533MHz, a common configuration for high-end Power5+ systems. More details about the DRAM chips that we model can be found in the datasheet from Micron [17].

The Power5+ [13] has an aggressive processor-side prefetching unit [26] that prefetches from memory to L2 and from L2 to L1. The prefetcher implements a sequential prefetching policy that waits to issue prefetches until it detects two consecutive cache misses. There are 12 entries in the stream detection unit, and eight streams can be prefetched concurrently. When the steady state is reached, each stream brings one additional line into the L1 cache, and one additional line into the L2 cache.

5.3 Benchmarks

Our evaluation uses the NAS [3] and SPEC2006fp benchmarks suites, along with a set of internal IBM commercial benchmarks. The commercial benchmarks consist of five server applications, namely, *tpcc*, *cpw2*, *trade2*, *sap*, and *notesbench*. Tpcc is an online transaction processing workload; cpw2 simulates the database server of an online transaction processing environment; trade2 is an end-to-end web application that models an online brokerage; sap is a database workload; and notesbench is a tool that evaluates the performance of a set of systems which are running Lotus Notes.

6. Evaluation

We compare the results of five configurations, where any mention of the processor-side prefetcher refers to the Power5+'s traditional stream buffer: no-prefetching (NP); processor-side prefetching only (PS); processor- and memory-side prefetching, where the memory-side prefetcher uses the original ASD approach (PMS); memory-side prefetching only using the enhanced ASD approach (EMS); and processor- and memory-side prefetching with the enhanced ASD prefetcher (EPMS).

We evaluate our ideas along several dimensions. We first present overall performance results for all three benchmark suites. To save space, we then use a subset of eight benchmarks to illustrate additional points, choosing the two best and two worst benchmarks—in terms of EPMS performance improvement over PMS—from SPEC, and the best and worst benchmarks from both the NAS and commercial benchmarks.

6.1. Benchmark Results

From our simulations, we find that the enhancements to ASD prefetching show substantial benefit. The EPMS configuration performs best, and the benefits from memory-side and processor-side prefetching are largely complementary but not completely orthogonal.

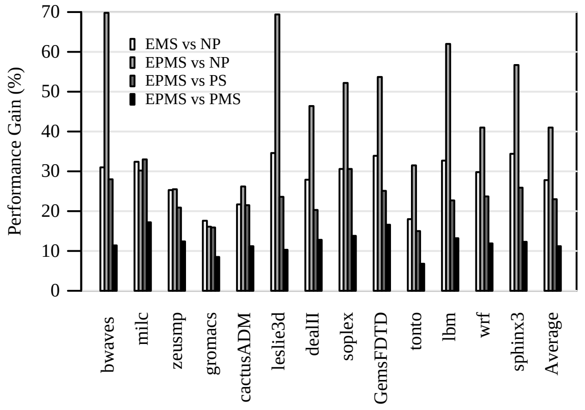

Figure 5. Performance improvements for the SPEC2006fp Benchmarks.

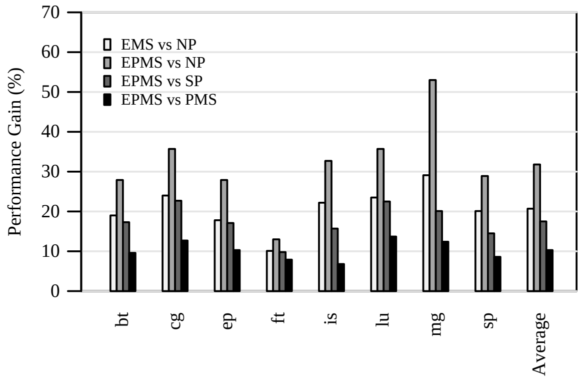

Figure 6. Performance improvements for the NAS Benchmarks.

We use the 13 most memory intensive benchmarks of the SPEC2006fp suite (Figure 5), and we find that the average performance benefit of EMS over NP is 27.8%.

SMT Results. We have repeated the above experiments on a system that uses two SMT threads on the same processor, and we now summarize our findings. In the SMT experiments we use two threads of the same benchmark, but we start the second thread one million instructions after the first thread. For these experiments, we leave the Prefetch Buffer size unchanged, but we double the size of the Stream Filter, so that each thread can track its own set of streams. We find that SMT performance improvements are about the same as for the single-threaded case. For example, EPMS improves performance over PMS by 10.1%,

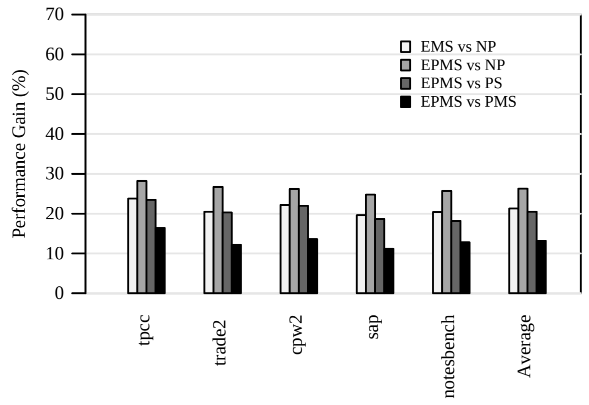

Figure 7. Performance improvements for the commercial benchmarks.

9.6%, and 12.6%, respectively, for the SPEC2006fp, NAS, and commercial benchmarks. The improvements for EPMS over NP are 34.2%, 18.9%, and 20.2%, respectively.

6.2. Analysis

We now analyze our enhancements and policies in more detail.

Effects of Individual Enhancements. In Figure 8, we show the performance effects of each of the three enhancements to the ASD prefetcher. We find that Variable-Length Prefetching alone improves performance by 3.0%-7.3%; Length-Based Stream Detection alone improves performance by 3.2%-7.2%; and Adaptive Epoch-Length improves performance by and 1.6%-9.1%. We see that these three methods are not completely orthogonal to each other.

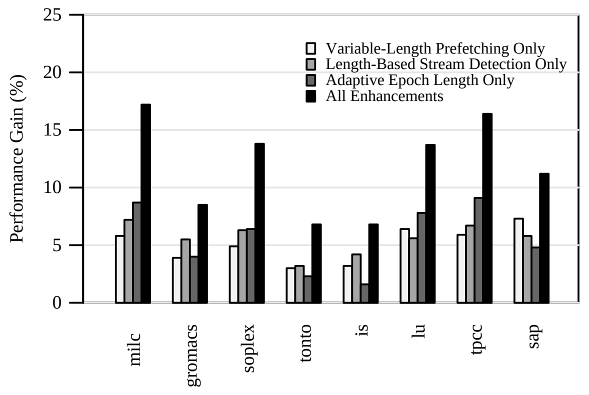

Figure 8. Performance improvements of the enhancements of EPMS over PMS.

Impact of Length-Based Stream Detection. We find that Length-Based Stream Detection significantly increases the potential impact of stream-based prefetching. As an example, in Figure 9 we show this effect for the milc benchmark. The vertical dotted line on the left represents the percentage of Read commands that might possibly be prefetched (that is, those that are not the first line of a stream) using the traditional fixed-lifetime stream detection scheme. The line on right represents the percentage of possibly prefetched Read commands using Length-Based Stream Detection.

Figure 9 also indicates how close our enhanced ASD Prefetcher comes to a *perfect* memory-side prefetcher, which we define as a prefetcher that can predict what to prefetch and when to issue prefetch requests such that *x*% of all Read requests find their data in the prefetch buffer and no memory commands are delayed because of the prefetch requests. As we vary the value of x from 0% to 100%, where $x = 100\%$ represents the *ideal* memory-side prefetcher, we get the solid line that represents a family of perfect prefetchers. The distance of the two “+” signs from the solid line and vertical dotted lines shows that there is still considerable room for improvement for the benchmark.

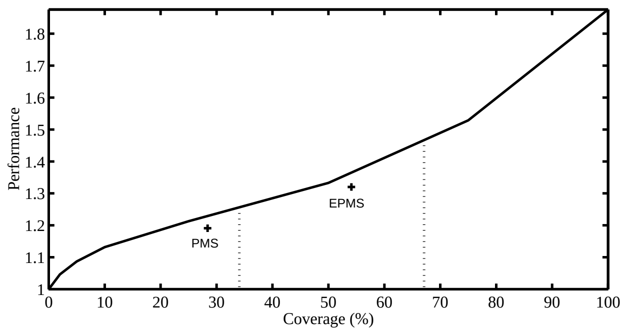

Figure 9. Comparison of the perfect prefetcher (solid line) to PMS and EPMS.

Impact of Adaptive Epoch Length. Table 1 provides a closer look at the benefits of Adaptive Epoch Length. Using the EPMS configuration, the table compares the use of Adaptive Epoch Length against various fixed epoch lengths, where each table entry represent speedup relative to a fixed epoch length of 256 memory commands. For each row, the value in bold font represents the best fixed epoch length for that benchmark. We observe that no single fixed epoch length yields the best performance for all benchmarks. We

Benchmark	Epoch Length							
	256	512	1024	2048	4096	8192	Average	Variable
milc	1.00	1.01	1.03	1.03	**1.05**	1.04	1.03	1.13
gromacs	**1.00**	1.00	0.98	0.97	0.97	0.97	0.98	1.01
soplex	1.00	0.99	1.00	1.01	**1.02**	1.02	1.01	1.08
tonto	**1.00**	0.95	0.98	0.99	0.99	0.99	0.98	1.01
is	1.00	**1.01**	1.01	1.01	1.01	1.00	1.00	1.02
lu	1.00	1.02	1.02	**1.03**	1.03	1.03	1.02	1.11
tpcc	1.00	1.03	**1.04**	1.02	1.00	0.99	1.01	1.08
sap	1.00	**1.01**	1.01	1.00	0.99	0.97	1.00	1.02

Table 1. Performance of EPMS with fixed and variable length epochs. Results are normalized to that of a fixed epoch length of 256. Among the fixed epoch lengths, none is always best. Variable length epochs always achieves better performance than any fixed length epoch.

also find that using variable epoch length is always superior to all of the fixed epoch lengths.

How effective is the state machine given in Figure 4? To graphically answer this question, Figure 10 shows the epoch lengths over time for the milc benchmark, as generated by our state machine. We see that all 6 epoch lengths are well represented, and we see that there is reasonable stability in the epoch length, as illustrated by the fairly wide lines. To answer the above question quantitatively, we measure the percentage of memory requests that reside in an epoch that is dissimilar (as defined in Section 4.2) from an adjacent epoch. The percentage is quite low when Adaptive Epoch Lengths is used, ranging from 4.1%-6.7% for our eight representative benchmarks, but it is quite high for the fixed epoch lengths, ranging from 24.2%-36.7%.

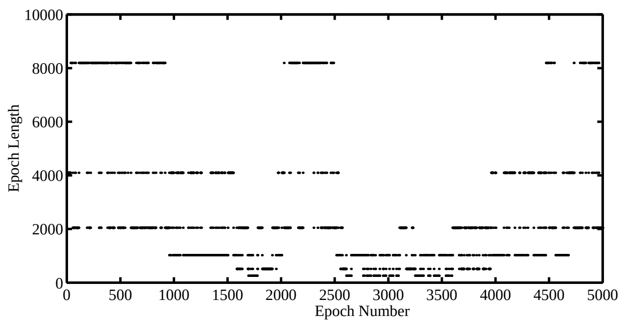

Figure 10. Changes in epoch length with the EPMS, for the milc benchmark. Epoch length varies between 256 and 8192 continuously during program execution.

Impact of Variable-Length Prefetching. Figure 11 shows the sensitivity of EPMS to Variable-Length Prefetching policies. We evaluate the performance effect of the number of lines to prefetch and the effect of the queue status check before generating multiple-line prefetches. We find that as the number of lines to prefetch increase from 2 to 3, the LPQ becomes occupied by multiple-line prefetches, and the benefit from the ASD prefetching approach degrades. We also find that the simple Queue Status Check heuristic, which suppresses multiline prefetching if the Read Reorder Queue is at least half full, has a significant positive impact on performance.

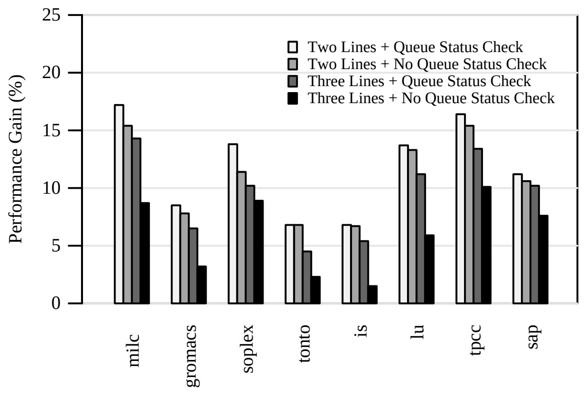

Figure 11. Performance effects of the variable-length prefetch generation policies, EPMS vs PMS.

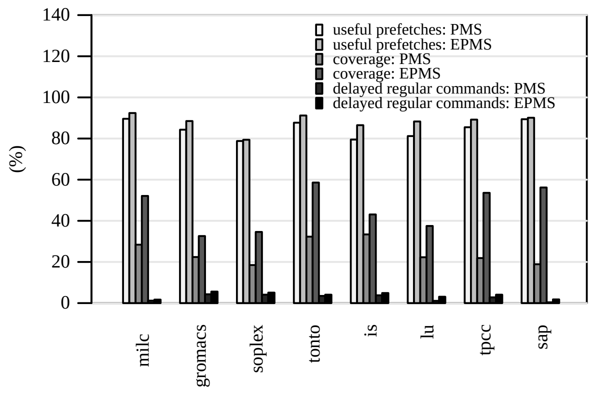

Figure 12. Effectiveness of the prefetching approaches.

Prefetch Effectiveness. Figure 12 illustrates the effectiveness of our enhancements by comparing PMS and EPMS using three metrics: (1) the percent of useful prefetches, (2) the prefetch coverage, that is, the percent of

Read commands (including processor-side prefetches) that get its data from the prefetch buffer, and (3) the percentage of regular memory commands—both Reads and Writes—that are delayed because of memory-side prefetches. The values in the figure pertain only to prefetches generated by the memory-side prefetcher, not the processor-side prefetcher. We see that for both PMS and EPMS the percentage of useful prefetches is between 78.4% and 92.1%. Although our enhancements increase the number of prefetches, the percentage of useful prefetches does not decrease with EPMS. Furthermore, the percentage of delayed regular memory commands increases only slightly. The prefetch coverage, on the other hand, improves significantly with EPMS. With the enhancements that we introduce to the PMS method, the average coverage for the benchmarks increase from 24.8% to 45.7%.

Sensitivity to Prefetch Buffer Size. Figure 13 shows, for the EPMS, the performance effect of the prefetch buffer size. In our simulations we use a configuration with a 16-block prefetch buffer, and we find that increasing buffer size beyond this configuration improves performance but with diminishing returns.

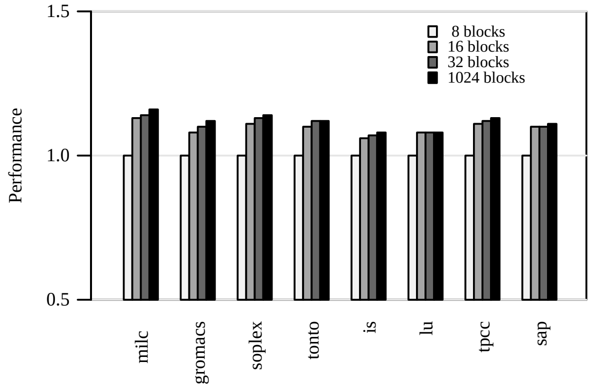

Figure 13. Sensitivity of the EPMS to prefetch buffer size.

Power and Energy Effects. To understand the power and energy effects of our techniques. in Figure 14, we compare DRAM power usage and energy consumption of PMS and EPMS with respect to PS. We find that, on average, PMS and EPMS increase power consumption by 3.2% and 3.3%, respectively, and they reduce energy consumption by 10.4% and 17.8%, respectively. Of course, the implementation of the enhancements in the EPMS itself also consumes power. We do not have benchmark-specific analyses of this power usage, but using an area-based estimation we calculate it to be negligible.

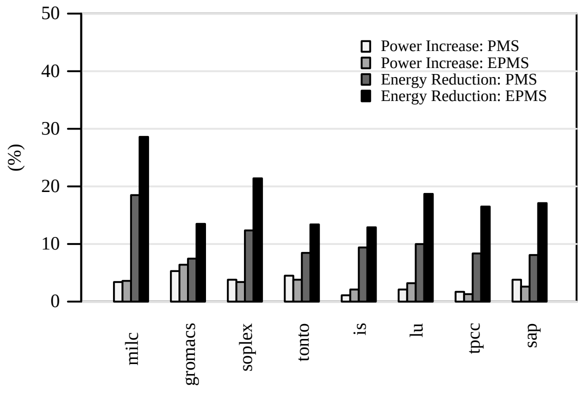

Figure 14. DRAM power and energy effects.

6.3. Hardware Costs

The current Power5+ memory controller occupies about 1.61% of the entire chip area, with the dominant portion of the memory controller being control logic. Using detailed estimates of transistor counts, earlier work [9] finds that the original ASD prefetcher increases the area of the memory controller by about 6.08%, resulting in a 0.098% increase in the total chip area. Our enhancements in this paper do not increase any table sizes in the memory controller, but using the same methodology, we estimate they do increase control logic of the memory controller by about 6.54%, resulting in a 0.106% increase in the total chip area compared to the original Power5+.

7. Conclusions

This paper has evaluated three techniques for improving the ASD prefetcher.

One technique, Adaptive Epoch Lengths, improves the quality of the feedback that is used to guide prefetching decisions. More broadly, this technique provides a framework that can be useful for other adaptive microarchitectural structures that base their current behavior on information gathered from some fixed-sized window into the recent past; this technique is generally useful because, due to phase behavior, the best choice of a window size typically varies over the lifetime of a workload. The specifics of the finite state machine for adapting behavior and the specifics of the similarity metric would, of course, need to be modified to be suitable for the particular structure in question.

A second technique, Length-Based Stream Detection, changes the mechanism that detects streams, thereby discovering a significantly larger number of short streams than was possible with previous stream buffers. The lesson is that if we wish to support both short and long streams, we need to revisit all aspects of the stream buffer so that they do not bias against short streams.

The third technique, Variable-Length Prefetching, is a tiny conceptual advance over the previously evaluated single-line ASD prefetcher. Variable-Length Prefetching supports multiple-line prefetching, where the number of lines prefetched is selected based on the Stream Length Histogram and state of the Read Reorder Queue, with the latter state being used to decrease the aggressiveness of the prefetching in cases where the memory system is busy.

The three techniques share some common characteristics. Each is adaptive and simple, using a simple measure of past behavior to guide its behavior. Each is effective, as each alone improves the performance of the ASD prefetcher by about 5% on our representative set of benchmarks. Collectively, these techniques improve the effectiveness of stream buffers, particularly for short streams, which leads to both improved performance and decreased energy consumption. When coupled with the Power5+'s processor-side stream buffer, the stream buffers provide a combined performance boost of 41.0% for the SPEC2006fp benchmarks, 20.8% for the NAS benchmarks, and 21.3% for a set of commercial benchmarks.

Acknowledgments. We thank the entire IBM Power5 team and the anonymous referees for their valuable comments. This research was supported by Air Force Research Laboratory contract FA8750-07-C-0035 from the Disruptive Technology Office and from a grant from the Intel Research Council.

References

[1] T. Alexander and G. Kedem. Distributed prefetch-buffer/cache design for high-performance memory systems. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 254–263, 1996.

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

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks (94). Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[4] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 70–79, 1999.

[5] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, pages 276–287, 2004.

[6] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson. Design and implementation of the Power5 microprocessor. In *Proceedings of the 41st Annual Conference on Design Automation*, pages 670–672, 2004.

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

[8] C. Hughes and S. Adve. Memory-side prefetching for linked data structures. Technical Report UIUCDCS-R-2001-2221, University of Illinois at Urbana-Champaign, 2001.

[9] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 397–408, 2006.

[10] T. L. Johnson, M. C. Merten, and W.-M. W. Hwu. Run-time spatial locality detection and optimization. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 57–64, 1997.

[11] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, 1997.

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

[13] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[14] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368, 1998.

[15] W. F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 301–312, 2001.

[16] W. F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak. Filtering superfluous prefetches using density vectors. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, pages 124–132, 2001.

[17] Micron. http://download.micron.com/pdf/datasheets/dram/ddr2/512MbDDR2.pdf, 2004.

[18] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, pages 96–105, 2004.

[19] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st*

Annual International Symposium on Computer Architecture, pages 24–33, 1994.

[20] K. Rajamani. *MEMSIM User's Guide*. IBM Research Report RC23431, 2004.

[21] S. Sair, T. Sherwood, and B. Calder. A decoupled predictor-directed stream prefetching architecture. *IEEE Transactions on Computers*, 52(3):260–276, March 2003.

[22] A. Smith. Sequential program prefetching in memory hierarchies. *IEEE Transactions on Computers*, 11(12):7–12, December 1978.

[23] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–182, 2002.

[24] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 252–263, 2006.

[25] 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 the 13th International Symposium on High-Performance Computer Architecture*, pages 63–74, 2007.

[26] J. M. Tendler, J. S. Dodson, J. S. Fields Jr., H. Lee, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[27] 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 the 30th Annual International Symposium on Computer Architecture*, pages 388–398, 2003.

[28] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 222–233, 2005.

[29] C.-L. Yang and A. R. Lebeck. Push vs. pull: data movement for linked data structures. In *Proceedings of the 14th International Conference on Supercomputing*, pages 176–186, 2000.

[30] L. Zhang, Z. Fang, M. Parker, B. Mathew, L. Schaelicke, J. Carter, W. Hsieh, and S. McKee. The Impulse memory controller. *IEEE Transactions on Computers*, 50(11):1117–1132, November 2001.