Memory Prefetching Using Adaptive Stream Detection

Ibrahim Hur[†]
ihur@cs.utexas.edu

Calvin Lin[‡]
lin@cs.utexas.edu

[†]Dept. of Electrical and Computer Engr.
The University of Texas at Austin
Austin, TX 78712

[†]IBM Corporation
Austin, TX 78758

[‡]Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

We present Adaptive Stream Detection, a simple technique for modulating the aggressiveness of a stream prefetcher to match a workload's observed spatial locality. We use this concept to design a prefetcher that resides on an on-chip memory controller. The result is a prefetcher with small hardware costs that can exploit workloads with low amounts of spatial locality. Using highly accurate simulators for the IBM Power5+, we show that this prefetcher improves performance of the SPEC2006fp benchmarks by an average of 32.7% when compared against a Power5+ that performs no prefetching. On a set of 5 commercial benchmarks that have low spatial locality, this prefetcher improves performance by an average of 15.1%. When compared against a typical Power5+ that does perform processor-side prefetching, the average performance improvement of these benchmark suites is 10.2% and 8.4%. We also evaluate the power and energy impact of our technique. For the same benchmark suites, DRAM power consumption increases by less than 3%, while energy usage decreases by 9.8% and 8.2%, respectively. Moreover, the power consumption of the prefetcher itself is low; it is estimated to increase the power consumption of the Power5+ chip by 0.06%.

1. Introduction

Numerous hardware solutions have been proposed to hide long memory latencies. Early prefetching techniques [13, 22, 19, 2, 7] focused on exploiting streaming workloads. While regular forms of spatial locality are easy to predict, it has traditionally been difficult to exploit irregular patterns of spatial locality and even more difficult to exploit low amounts of spatial locality.

Recently, a class of aggressive prefetching techniques has arisen from the notion of a Spatial Locality Detection Table [11]. These techniques track accesses to regions of memory so that spatially correlated data can be prefetched together [11, 15, 5, 17, 24]. The chief advantage of these techniques is their ability to exploit irregular forms of spatial locality. Their chief disadvantage is their reliance on large tables that occupy chip area and consume power.

We propose a new solution, which uses a simple technique to augment the effectiveness of stream prefetchers. Our technique is based on two observations. First, memory intensive workloads with low amounts of spatial locality are likely to still contain many very short "streams," if "stream" can be defined to be as short as two consecutive cache lines. Second, stream prefetchers could effectively prefetch these short streams if they only knew when to be aggressive.

To understand this second point, recall that stream prefetchers look for accesses to k consecutive cache lines, at which point the $k+1^{st}$ cache line is prefetched; prefetching continues until a useless prefetch is detected. Thus, the value of k determines the prefetcher's aggressiveness, and this value is typically fixed at design time. Even with a small value of k, stream-based prefetchers do not fare well on short streams because the stopping criterion is a useless prefetch. For example, on a workload in which every stream is of length 2, a $k = 1$ policy would successfully prefetch the second cache line of each stream, but each successful prefetch would be followed by a useless prefetch, so 50% of its prefetches would be useless.

Our solution, *Adaptive Stream Detection*, guides the aggressiveness of the prefetch policy based on the workload's observed amount of spatial locality, as measured by a *Stream Length Histogram (SLH)*. An SLH is a dynamically computed histogram that attributes each memory access to a particular stream length. For example, if the SLH indicates that 70% of the memory requests were parts of streams of length 2 and that 30% of the memory requests were parts of streams of length 1, then an effective strategy

would always prefetch the second cache line of a stream but never the third line. Thus, Adaptive Stream Detection can predict when to stop prefetching without incurring a useless prefetch. To adapt to changes in phase behavior, new Stream Length Histograms are computed periodically.

Adaptive Stream Detection provides two benefits. (1) It extends the notion of a stream to include streams as short as two cache lines. Thus, while it is inherently a stream-based approach, it provides benefits for workloads, such as commercial applications, that are not traditionally viewed as stream-based. (2) Because it is stream-based, it has low hardware costs, using small tables that have low static power leakage.

This paper describes how Adaptive Stream Detection can be implemented in the memory controller. In this context, we introduce a second idea, Adaptive Scheduling, that adjusts the priority of prefetched commands based on the measured frequency of conflicts that prefetched commands have caused. This adaptivity is useful because any fixed priority may be excessively conservative for some workloads.

This paper makes the following contributions:

- We introduce Adaptive Stream Detection, a probabilistic prefetching technique that adjusts the aggressiveness of stream prefetching based on Stream Length Histograms, which are inexpensive to gather. This technique addresses the question of *what* to prefetch.

- We use the idea of Adaptive Stream Detection to design a prefetcher that resides in the memory controller and prefetches from DRAM into a small Prefetch Buffer. This prefetcher uses Adaptive Scheduling to modulate the relative priority of prefetch commands to regular commands. We show that a prefetch buffer that holds 16 cache lines is effective. We also see that this memory-side prefetcher (MS) complements the IBM Power5+'s existing stream prefetcher (PS), which performs processor-side prefetching.

- We evaluate Adaptive Stream Detection using the SPEC2006 floating point suite, the NAS benchmarks, and a set of five commercial benchmarks. For single threaded workloads, when we compare our technique to a stripped down Power5+ with no prefetching (NP), we improve the performance of the SPEC2006fp, NAS, and commercial benchmarks by 14.6%, 11.7%, and 9.3%, respectively. When MS is combined with PS, forming PMS, its improvements over NP are 32.7%, 24.2%, and 15.1%, respectively. The performance improvements for the commercial benchmarks are noteworthy because these benchmarks exhibit low amounts of spatial locality. We get similar results for SMT workloads.

- We evaluate the energy and power impact of our approach. For our three benchmark suites, we find that DRAM power consumption increases by 2.7%, 1.6%, and 2.8%, respectively, while DRAM energy consumption decreases by 9.8%, 7.9%, and 8.2%, respectively. For the four SPEC2006fp benchmarks that have low memory bandwidth requirements, the DRAM power impact is negligible: DRAM power increases by an average of 0.12%, while energy consumption decreases by 0.47%.

- We evaluate Adaptive Scheduling and show that it improves upon a set of conservative fixed-priority policies by about 2.9%.

This paper is organized as follows. The next section places our work in the context of prior work. Section 3 describes our solution. We present our experimental methodology in Section 4 before presenting our empirical evaluation in Section 5.

2. Related Work

One line of hardware prefetching research has extended next-line prefetching [22, 13] by adding non-unit strides [19], by predicting strides [2, 7], and by supporting irregular strides using Markov predictors [12, 21]. Nesbit and Smith [18] introduce the *Global History Buffer* to improve prefetch effectiveness and reduce table sizes. None of these prefetchers has successfully exploited low amounts of spatial locality.

Another line of research focuses on detecting and exploiting spatial locality without tracking individual streams [11, 15, 17, 5]. Instead, variations of the *Spatial Locality Detection Table*, introduced by Johnson *et al.* [11], 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 how 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 use tables as small as 64KB. Moreover, Somogyi *et al.* show performance improvements for commercial workloads, indicating that their technique can handle locality patterns that span large regions of memory. 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.

Scheduled Region Prefetching (SRP) [16] prefetches large regions of memory, such as 4KB at a time, and introduces mechanisms for reducing the opportunity cost of prefetches. Prefetches to open banks are given priority,

prefetched data are brought into the LRU position of the L2 sets, and prefetched commands are given low priority in the memory controller. In particular, the SRP prioritizer receives feedback from the memory system and issues prefetch commands only if the channels are idle and there is no pending request from the L2 cache. By contrast, Adaptive Scheduling uses feedback from the memory system to select from among five different prioritization policies, where its most conservative policy is roughly equivalent to the SRP prioritization policy. Adaptive Scheduling can improve performance because for some workloads, the most conservative policy unnecessarily inhibits prefetches. For example, there may be pending demand requests that will not conflict with a prefetch command because they target different memory banks.

One issue with SRP is the high memory bandwidth pressure that it incurs because of its large regions. Wang *et al.* [26] 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.

Others have studied memory-side prefetching [1, 4, 27, 28, 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 accesses.

3. Our Solution

This section describes our new prefetcher, which resides in the memory controller. This prefetcher addresses two major questions: (1) How can we reduce the number of unnecessary prefetch requests? (2) How can we reduce the opportunity cost of prefetches? Adaptive Stream Detection addresses the first issue, and Adaptive Scheduling addresses the second. To provide context, we first briefly describe the Power5+'s memory controller. After explaining the basic idea behind Adaptive Stream Detection, we describe the mathematical details of how SLH's are used, discuss implementation issues, and present the organization of our prefetcher. Finally, we present details of Adaptive Scheduling.

The Power5+ Memory Controller. As shown in Figure 1, the Power5+ memory controller sits 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.

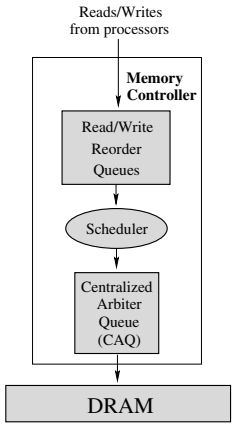

Figure 1. The IBM Power5+ Memory System.

3.1. Adaptive Stream Detection

Adaptive Stream Detection uses Stream Length Histograms, SLH, to capture spatial locality and guide prefetch decisions. For example, Figure 2 shows an SLH for one epoch of the *GemsFDTD* benchmark from the SPEC2006 suite. In an SLH, 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 how many, if any, sequential cache lines to prefetch.

In the example SLH of Figure 2, we see that 21.8% of all Read requests belong to streams of length 1, 43.7% of the Reads are part of streams of length 2, etc. The rightmost bar indicates that 1.2% of all Read requests were part of streams of length 16 or more. Given this information, 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 a Read request, R_n, is 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. With similar reasoning, prefetches should be issued for any Read request whose current stream length is 3 or greater than 6. This example shows that the use of the SLH allows a prefetcher to make rather sophisticated prefetching decisions based on the length of an individual stream.

The prefetcher can also use the SLH to decide whether to generate multiple prefetches—although we do not evalu-

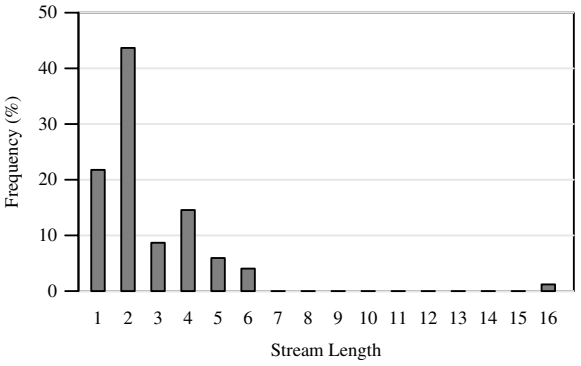

Figure 2. Stream Length Histogram (*SLH*) for an arbitrary epoch of the GemsFDTD benchmark.

ate this idea in this paper. For example, when R_n is part of a stream of length 1, the prefetcher decides whether to generate two consecutive prefetches by adding the probabilities of the first two bars and comparing the sum with the rest of the histogram. If the sum of the first two bars is less than the sum of the other bars, and if the prefetcher has already decided to prefetch one line, it generates a prefetch for the second line as well.

Because memory access behavior typically varies over time, our solution periodically creates an SLH after every e Read requests, where e is known as an epoch. Thus, in every epoch, our method constructs a new SLH for use in the next epoch. Figure 3 shows how epochs can vary widely over time. To keep track of increasing or decreasing streams, we need one SLH for each direction.

3.2. Using the SLH to Detect Locality

Our probabilistic approach to prefetching makes decisions 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. In this subsection, we derive inequalities that guide these prefetch decisions. Our discussion also establishes the transition from the SLH concept to its implementation that we present later in Section 3.4.

Definitions. To describe our method, we define two functions, $lht()$ and $P()$, which can be used to compute an SLH, as follows:

$lht(i)$: 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 we track. For any $i > fs$, $lht(i) = 0$.

$P(i, j)$: the sum of probabilities that a Read is part of any stream of length k, where $i \leq k \leq j$ and $1 \leq i, j \leq fs$. We can define $P(i, j)$ in terms of $lht()$ as follows:

$$P(i, j) = (lht(i) - lht(j + 1))/lht(1) \quad (1)$$

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

Prefetch Decision. To determine whether to issue a prefetch, we check whether the following condition is satisfied for a Read request, R_n, that is the i^{th} element of a stream:

$$P(i, i) < P(i + 1, fs) \quad (2)$$

This inequality states that the probability that the most recent Read request, R_n, is part of a stream of length i is smaller than it being a part of a stream of length longer than i. We can simplify the inequality (2) as follows:

$$P(i, i) < P(i + 1, fs) \quad (3)$$

$$\equiv \frac{lht(i) - lht(i + 1)}{lht(1)} < \frac{lht(i + 1) - lht(fs + 1)}{lht(1)} \quad (4)$$

$$\equiv lht(i) < 2 \times lht(i + 1) \quad (5)$$

Our technique uses the inequality (5) to make next line prefetch decisions. We provide, without proof, a generalized version of (5) to prefetch k consecutive lines after R_n:

$$lht(i) < 2 \times lht(i + k) \quad (6)$$

3.3. Prefetcher Design

The organization of our prefetcher is shown in Figure 4, where the gray boxes represent our 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.

The Prefetch Buffer is checked twice. It is first checked before Read commands are placed in the CAQ, so that Read commands can be satisfied by the Prefetch Buffer, in which case the latency of going to DRAM is saved and the Read command is squashed. The Prefetch Buffer is checked again when the Final Scheduler selects a Read command from the CAQ to send to memory; this check is useful because the desired data may have arrived in the Prefetch Buffer while the Read command was resident in the CAQ.

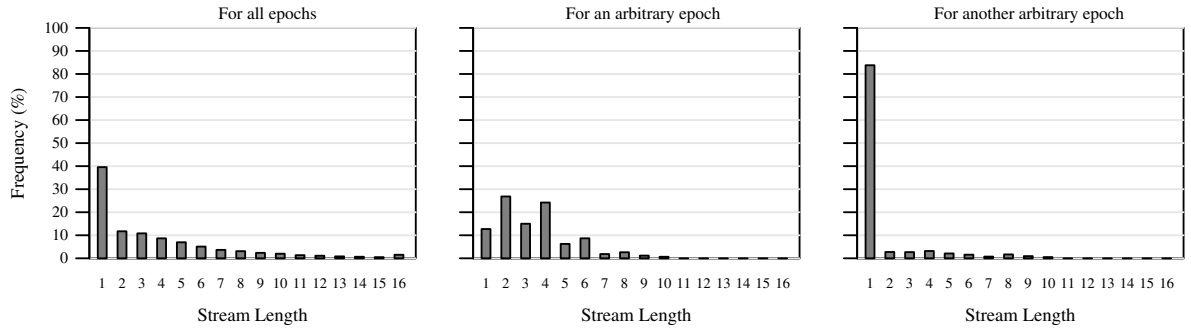

Figure 3. Stream Length Histograms (*SLH*) for the GemsFDTD benchmark from the SPEC2006fp suite show that the *SLH*'s vary widely at different points in time. Here the epoch length is 2000 reads.

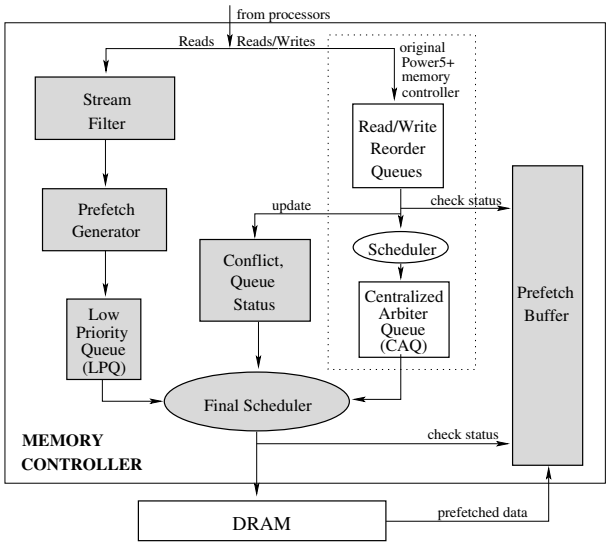

Figure 4. Overview of our prefetcher.

Stream Filter. To maintain information about Read streams, the Stream Filter uses one *slot* to track each Read stream. Each slot maintains (1) the last address accessed for this stream, (2) the length of the stream, (3) the stream's direction, and (4) the stream's *lifetime*, which indicates when the stream should be evicted. These slots are used as follows:

- If the Read, R_n, is not part of a stream and if there is a vacant slot in the Prefetch Filter, the last access field is set to the address of the Read request, the length field is initialized to 1, the lifetime is initialized to a predetermined value, and the direction is set to Positive.
- If R_n is not part of a stream and there is no available slot, no prefetch will be generated after R_n, but the SLH structure is updated as if a stream of length 1 had been detected.
- If R_n is the most recent element of a previously detected stream, the stream length is incremented by 1, the last access is set to the address of R_n, and the lifetime of the stream is incremented by a predetermined value.
- The direction of the stream is set to Negative if the length of the previous stream is 1 and the address of R_n is smaller than the last address of the stream.
- At every processor cycle, the lifetime fields are decremented by one. A stream is evicted from a slot when its lifetime expires. At this point, the SLH structure is updated using the length value in the Stream Filter.
- At the end of each epoch, all streams are evicted from the Stream Filter.

Prefetch Buffer. The Prefetch Buffer holds data that are fetched from memory by the memory-side prefetcher. We assume that this buffer is a set associative cache with an LRU replacement policy. When there is a write request to an address in the Prefetch Buffer, we invalidate the entry in the buffer. We also invalidate the entry if a regular Read request matches the address, because in such cases the data will likely be moved to the L1 or L2 cache, so it is unlikely to be useful in the Prefetch Buffer again.

3.4. Implementation of Adaptive Stream Detection

We now present details for implementing Adaptive Stream Detection. For simplicity, we restrict our explanation to streams with increasing addresses only, and we only discuss prefetching for one cache line. It is straightforward to generalize this approach to streams with decreasing addresses and multiple line prefetching.

Rather than implement the SLH explicitly, we construct the information in the SLH using two tables of length

fs. These *Likelihood Tables*, *LHTcurr* and *LHTnext*, correspond to the *lht()* function discussed previously. A given epoch uses and updates information from LHTcurr and gathers information for the start of the next epoch in LHTnext. LHTnext is updated using the information from the Stream Filter. When an entry of length k in the Stream Filter is invalidated, LHTnext[i] is incremented by k, for all i, where $1 \leq i \leq k$. At the end of an epoch, LHTnext is modified using the remaining valid entries in the Stream Filter; the contents of LHTnext are moved to LHTcurr; and LHTnext is re-initialized. Each entry of the tables is a $\log_2(m)$ bit counter, where m is the maximum epoch length.

LHTcurr is used to make prefetch decisions for the current epoch. This table has one comparator for each pair of consecutive table entries, *i.e.*, LHTcurr[i] and LHTcurr[i+1], for $1 \leq i < fs$. At the beginning of an epoch, the contents of LHTcurr are used to construct the SLH. As the epoch progresses, this information is modified using the observed stream lengths of the current epoch. When an entry of length k in the Stream Filter is invalidated, the value of LHTcurr[i] is decremented by k, for all i, where $1 \leq i \leq k$.

When the Stream Filter observes that a Read request is part of a stream of length k, prefetch requests are generated using the output of the comparison of LHTcurr[k] and LHTcurr[k+1], as in inequality (5). Instead of multiplying LHTcurr[k+1] by 2, for any k, the comparator for the (LHTcurr[k], LHTcurr[k+1]) pair takes the left shifted value of LHTcurr[k+1] as input.

3.5. Adaptive Scheduling

Clearly, speculative prefetch commands should be given lower priority than regular commands. But because memory systems are becoming increasingly complex, and because the Final Scheduler must make decisions whose effects may not be seen until the future, it is not obvious what policy provides the best performance. For example, a conservative policy that always gives prefetch commands lower priority than regular commands may unnecessarily block prefetch commands behind regular commands that cannot issue due to conflicts in the memory system. Thus, rather than dictate a particular policy at design time, Adaptive Scheduling uses feedback to dynamically select from one of five policies in order of decreasing conservativeness: Only issue a command from the LPQ (1) if the CAQ is empty and the Reorder Queues are empty, (2) if the CAQ is empty and the Reorder queues have no issuable commands, (3) if the CAQ is empty, (4) if the CAQ has at most 1 entry and the LPQ is full, (5) if the first LPQ entry has an earlier timestamp than the first CAQ entry.

To choose from among these policies, the memory controller tracks the number of times that a regular command in the Reorder Queues cannot proceed to the CAQ because it conflicts in the memory system with a previously issued prefetch command. As the occurrences of these conflicts grows (or shrinks), the policy becomes more (or less) conservative. The policy is adjusted using the same epoch size that is used to compute Stream Length Histograms. Thus, this approach determines the priority of prefetch commands based on a measure of memory system performance, rather than on some instantaneous property such as occupancy of a queue.

4. Experimental Methodology

4.1. Benchmarks

Our evaluation uses the SPEC2006fp benchmarks, the NAS benchmarks, and a set of IBM internal commercial benchmarks. The SPEC2006fp benchmarks are the latest versions of the SPEC floating point benchmarks. We do not use the SPEC2006int suite because with the Power5+'s large caches, these programs have low memory pressure. The NAS benchmarks [3] are a group of eight programs derived from computational fluid dynamics applications; we use serialized versions of the class B benchmarks. Our commercial benchmarks represent commercial server applications, namely *tpc-c*, *trade2*, *cpw2*, *sap*, and *notesbench*. Tpc-c is an online transaction processing workload; cpw2 is a Commercial Processing Workload that 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.

4.2. Simulated System

We evaluate performance in the context of the IBM Power5+ [6, 14], which is the latest member of the Power4 [25] line of processors. The Power5+ chip has more than 275 million transistors. The Power5+ 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.

We simulate a Power5+ running at 2.132GHz. 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.

The Power5+ [14] has an aggressive processor-side prefetching unit [25] 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.

4.3. 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, is on the order of 1 million lines of C and C++ code, and it uses execution traces to simulate both the processor and the memory system. To simulate SPEC and NAS benchmarks, which have billions of dynamic instructions (some of the SPEC2006fp benchmarks have trillions of instructions), we use uniform sampling, taking 50 uniformly chosen samples that each consist of 2 million instructions. For commercial benchmarks, we use special hardware to collect traces. The Power5+ simulator is integrated with Memsim [20], a DRAM simulator that jointly models power and performance of the main memory subsystem. In this simulation environment, Memsim models all the memory system activity while synchronizing with the Power5+ simulator at every processor cycle.

5. Experimental Results

We evaluate Adaptive Stream Detection along several dimensions. We present overall performance and power results for all three benchmark suites. To save space, we then use a subset of the benchmarks to illustrate additional points, choosing the two best-case and the two worst-case benchmarks—in terms of PMS performance improvement—from the SPEC and commercial benchmarks.

5.1. Hardware Costs

We evaluate a prefetcher that is configured as follows: Each thread has a Stream Filter with 8 slots and LHTnext and LHTcurr tables that each hold 16 entries. Because streams are tracked in both the positive and negative directions, LHTnext and LHTcurr each require 32 counters per thread. In addition to these per-thread resources, the prefetcher has one 16 entry Prefetch Buffer (2KB) and an LPQ with the same number of entries—3—as the CAQ. 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. Our extensions to the memory controller increase the area of the memory controller by about 6.08%, resulting in a 0.098% increase in the total chip area.

5.2. Benchmark Results

We now compare simulation results for four configurations: no-prefetching (NP), processor-side prefetching only (PS), memory-side prefetching only (MS), and processor- and memory-side prefetching together (PMS). In PMS, only the memory-side prefetcher uses Adaptive Stream Detection. In the following graphs, we present three different comparisons: (1) PMS vs. NP (2) MS vs. NP, and (3) PMS vs. PS.

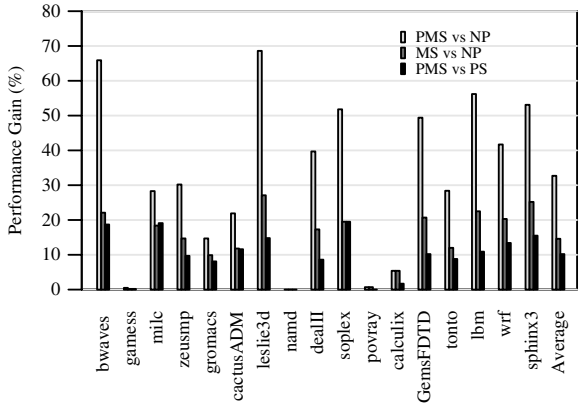

Figure 5. Performance improvements for the SPEC2006fp Benchmarks.

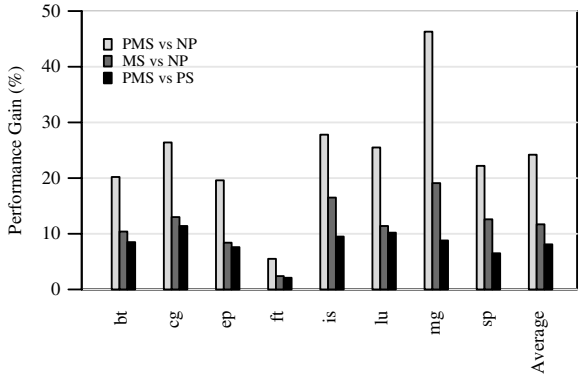

Figure 6. Performance improvements for the NAS Benchmarks.

We see that the PMS configuration performs best, and the benefits from memory-side and processor-side prefetching are largely complementary but not completely orthogonal.

For the SPEC2006fp benchmarks (Figure 5), we find that the performance benefit of PMS over NP is between

0-68.6%, with an average of 32.7%. MS improves performance over NP by an average of 14.6%, and PMS improves over PS by an average of 10.2%. For the NAS benchmarks (Figure 6), the PMS approach sees an average improvement of 24.2% over NP and 8.1% over PS. For the commercial benchmarks (Figure 7), the PMS approach sees an average improvement of 15.1% over NP and 8.4% over PS.

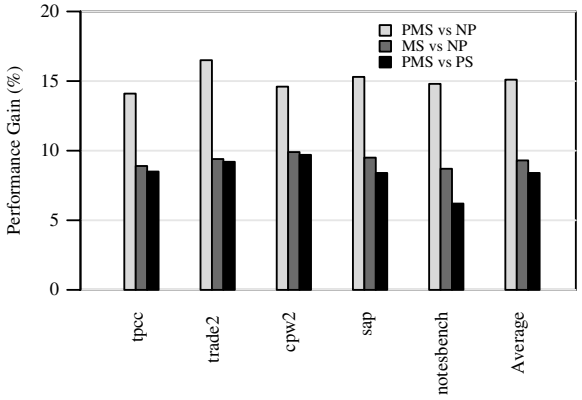

Figure 7. Performance improvements for the commercial benchmarks.

SMT Results. We have repeated the above experiments on a system that uses two SMT threads on the same processor, but for space reasons we omit these graphs. For these experiments, we leave the Prefetch Buffer size (16 cache lines) unchanged, but we double the size of the Stream Filter and the number of LHT tables, so that each thread can track its own set of streams. We find that SMT performance improvements are about the same as the single-threaded results. For example, PMS improves performance over PS by 10.7%, 9.2%, and 7.5%, respectively, for the SPEC2006fp, NAS, and commercial benchmarks. The improvements for PMS over NP are 28.5%, 20.4%, and 11.1%, respectively.

We find it critical to replicate the locality identification hardware—in our case the Stream Filter—for each thread. For our solution, this hardware is small, as opposed to many other solutions [17, 5, 24] for which large tables would have to be replicated.

5.2.1. Power and Energy Effects

In Figures 8, 9, and 10, we compare PMS to PS in terms of DRAM power usage and energy consumption. We find that PMS increases power consumption, on the average, by 2.7%, 1.6%, and 2.8% for SPEC2006fp, NAS, and commercial benchmarks, respectively. For the same benchmarks, PMS reduces energy consumption by 9.8%, 7.9%, and 8.2%. For the four benchmarks that are not mem-

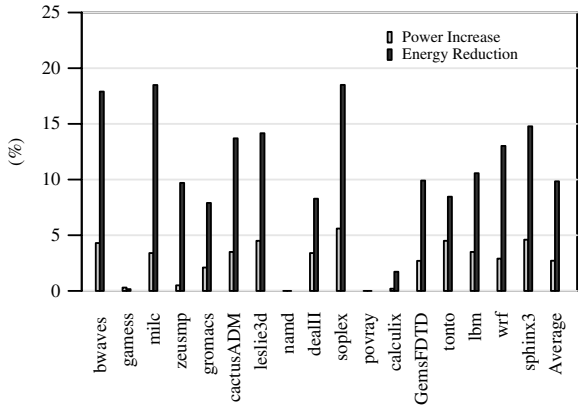

Figure 8. DRAM Power and Energy comparison for the SPEC2006fp benchmarks.

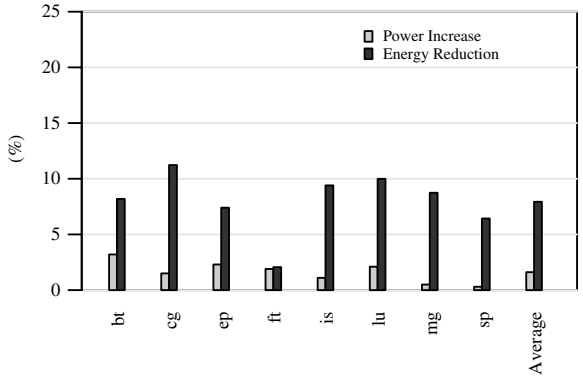

Figure 9. DRAM Power and Energy comparison for the NAS benchmarks.

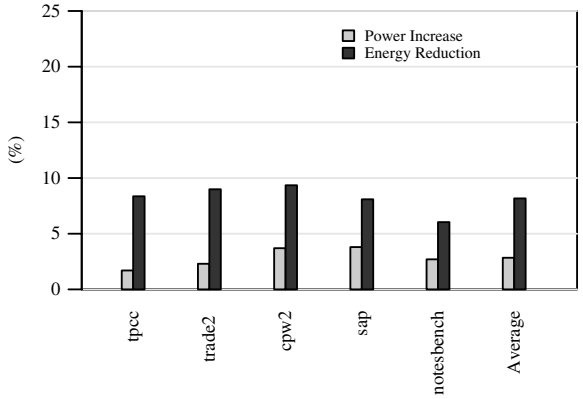

Figure 10. DRAM Power and Energy comparison for the commercial benchmarks.

ory intensive—*gamess*, *namd*, *povray*, and *calculix*— the power increase is negligible. Again, for SMT workloads, the DRAM power and energy results are similar to the single threaded case.

Other Power Costs. Of course, the implementation of the prefetcher itself also consumes power. We do not have benchmark-specific analyses of this power usage, but an analysis of the Power5+ chip and an area-based estimation of the MS prefetcher provides the following figures. The memory controller on the Power5+ consumes about 1% of the chip's power. The MS prefetcher increases the power of the memory controller by approximately 6%, which is 0.06% of the chip's total power. As a reference, the Power5+ chip typically consumes roughly four times the power as the DRAM chips for our workloads.

By contrast, if we were to add a 64KB table for detecting spatial locality, as suggested by other approaches, we would add four such tables—one for each thread—for the Power5+. We believe that each 64KB table would consume up to 25% of the power of a 64KB L1 I-cache (Loads constitute roughly 25% of all instructions), which for the Power5+ is about 0.6% of the chip's power. To support four such tables would increase the chip's active power by about 2.4%. Moreover, as leakage power becomes more important to future systems, the power effects of large tables will become more significant.

5.3. Detailed Results

Importance of Adaptive Stream Detection and Adaptive Scheduling. Figure 11 shows that both Adaptive Stream Detection (ASD) and Adaptive Scheduling contribute to performance gain. In this figure, the first bars in each cluster represent normalized execution times for our PMS approach. The next five bars compare the PMS against the five scheduling policies that we discussed in Section 3.5. We see that the Adaptive Scheduling improves performance upon these fixed policies between 2.3% and 3.6%. We conclude that the impact of Adaptive Stream Detection is much more significant than that of Adaptive Scheduling.

Figure 11 also provides a head-to-head comparison of Adaptive Stream Detection against both next-line prefetching (second bar from the right) and the Power5's processor-side prefetcher (rightmost bar) when all are implemented in the memory controller. We see that Adaptive Stream Detection provides performance that is 8.4% better than the next-line prefetcher. Somewhat surprisingly, in this context the Power5-style prefetcher yields worse performance than the next-line prefetcher.

Figure 12 shows that a significant portion of streams are of length five or shorter. These short streams are where Adaptive Stream Detection sees the most benefit. A next-line prefetcher generates useless prefetches for all streams of length one, and we see that the percentage of such streams is quite high for these benchmarks. There is also a significant number of streams of length 2-5, which is where a Power5-style stream-based prefetcher sees the worst performance: For these streams the useless prefetch that it issues before detecting the end of a stream represents a non-trivial fraction of the total prefetches. Finally, observe that even the four commercial benchmarks, which have poor spatial locality, have a significant percentage of streams of length 2-5: roughly 37% for *tpc-c*, 49% for *trade2*, 40% for *sap*, and 62% for *notesbench*. These percentages help explain why Adaptive Stream Detection is beneficial even for workloads with low spatial locality.

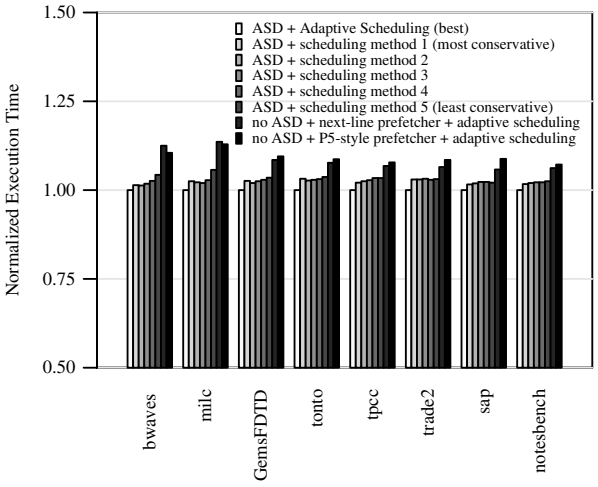

Figure 11. Impact of Adaptive Stream Detection and Adaptive Scheduling.

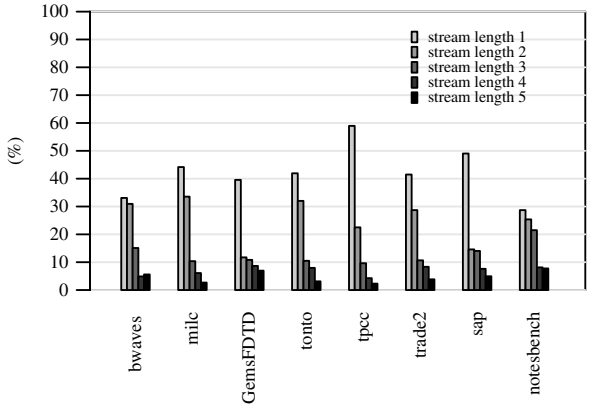

Figure 12. Stream Length Histograms of eight benchmarks. Streams of lengths between 1 and 5 constitute 78–96% of all streams.

Prefetch Efficiency. Figure 13 presents three measures of the effectiveness of Adaptive Stream Detection: (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 the regular memory commands—both Reads and Writes—that are delayed because of memory-side prefetches. These values pertain only to prefetches generated by the memory-side prefetcher, not the processor-side prefetcher. We see that the percentage of useful prefetches is between 82% and 91%. The coverage is between 19% and 34%, and only 1-3% of regular commands are delayed by the memory-side prefetch commands.

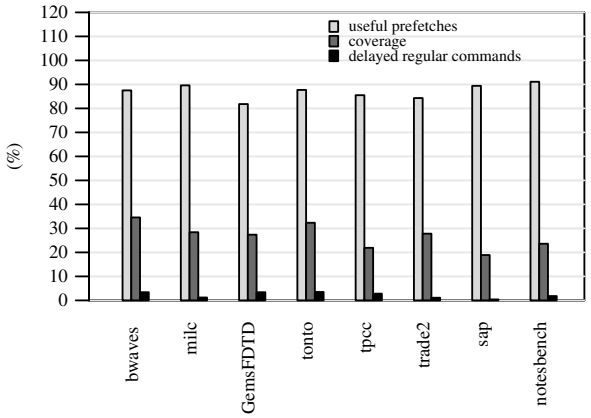

Figure 13. Effectiveness of our prefetching approach.

Sensitivity to Prefetch Buffer and Stream Filter Size. Figures 14 and 15 show, for our PMS approach, the performance effect of the size of the Prefetch Buffer and Stream Filter. In our simulations, we use a configuration with a 16-block prefetch buffer and an 8-entry stream filter. We find that increasing the size of the Prefetch Buffer or Stream Filter beyond this configuration improves performance but with diminishing returns.

Accurately Constructing Frequency Histograms. The success of Adaptive Stream Detection depends on the accuracy of the computed Stream Length Histograms, which are computed using the Stream Filter. Because the Stream Filters have finite size, the computed SLH is actually an approximation of a complete SLH. We have found that this approximation of the SLH closely matches the actual SLH, as shown in Figure 16, which is a sample epoch in the *GemsFDTD* benchmark.

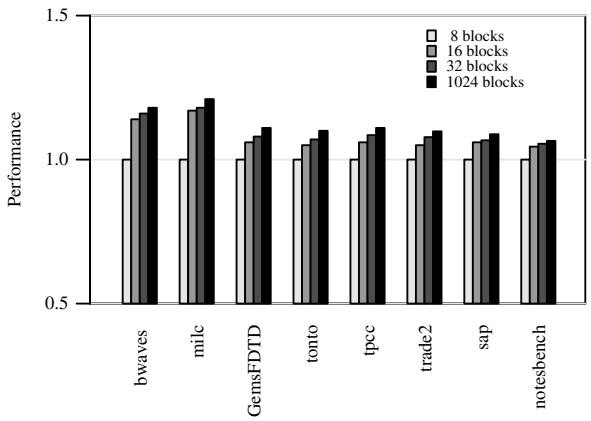

Figure 14. Sensitivity of PMS to prefetch buffer size.

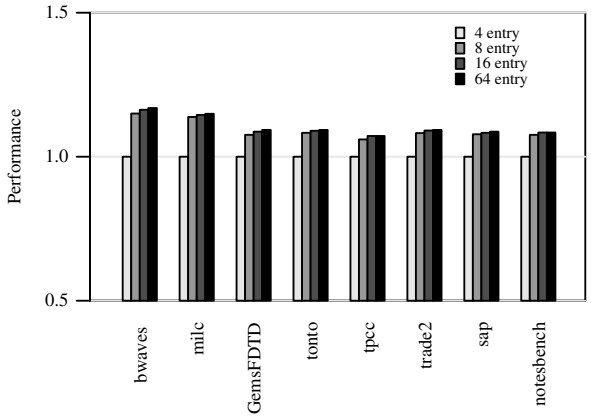

Figure 15. Sensitivity of PMS to stream filter size.

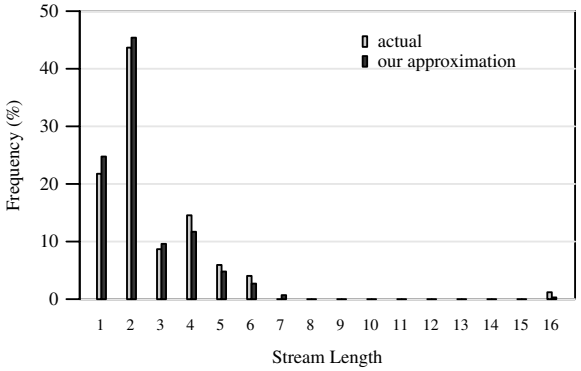

Figure 16. Accuracy of calculating Stream Length Histograms.

Interaction with the Memory Scheduler. The impact of a prefetcher can be sensitive to the choice of memory scheduler (Scheduler in Figures 1 and 4) that is used. For the results presented in this paper, we use the Adaptive History-Based memory scheduler (AHB) [9, 10], but to investigate the interaction between memory scheduling algorithms and our new prefetching technique, we also study two less sophisticated memory schedulers, *in-order* and *memoryless* [9], which provide reduced DRAM bandwidth compared to the AHB scheduler. When a simple in-order scheduler is used, the performance gain of our prefetcher is reduced by about 5%. For the better memoryless scheduler, the performance gain of our prefetcher is reduced by about 1%. These results indicate that the benefit of our prefetching approach increases as other bottlenecks in the memory subsystem are reduced.

6. Conclusions

We have introduced a new stream-based prefetching technique that is effective for streams of any length, including extremely short streams. The key idea is to monitor the amount of spatial locality in a program's execution to adjust the aggressiveness of a basic stream prefetcher. By capturing such spatial locality in a Stream Length Histogram, our prefetcher can probabilistically decide when to start and stop prefetching based on the recently observed behavior. A secondary contribution is the notion of Adaptive Scheduling, which adapts the aggressiveness of the prefetcher based on the observed number of conflicts between prefetch commands and regular commands. Previous techniques [16] have monitored specific aspects of the memory system, but we show that such fixed policies can be overly conservative.

Using extremely accurate simulators for a modern microprocessor and its memory system, we have shown that Adaptive Stream Detection and Adaptive Scheduling provide significant performance improvements, even for commercial workloads that have low spatial locality. This solution also has low DRAM power costs and modestly improves DRAM energy consumption. If implemented in the Power5+, our solution increases the area of the chip by less than 0.1%. Compared to other prefetching strategies, the hardware cost of our approach is minimal. Moreover, because its spatial locality detection component is small, the cost advantage of Adaptive Stream Detection improves—relative to other approaches that require large tables—as the number of hardware threads increases.

Conceptually, we have shown that simple stream-based prefetching can be effective, even for commercial workloads that exhibit low amounts of spatial locality. As future work, we will consider applying Adaptive Stream Detection to processor-side prefetching.

Acknowledgments. We thank Alper Buyuktosunoglu for his helpful expertise on power consumption. We thank Doug Burger and E Lewis for their comments on an early draft of this paper. We thank the entire IBM Power5 team, in particular, Men-Chow Chiang, Cheryl Chunco, Steve Dodson, John Griswell, Douglas Logan, John McCalpin, Gary Morrison, Stephen J. Powell, and Karthick Rajamani. This work was supported by NSF grant ACI-0313263, an IBM Faculty Partnership Award, DARPA contract F33615-03-C-4106, and DARPA contract NBCH30390004.

References

[1] T. Alexander and G. Kedem. Distributed prefetch-buffer/cache design for high-performance memory systems. In *HPCA '96: 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 Technical Report RNR-94-007, 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 *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 70–79. IEEE Computer Society, 1999.

[5] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 276–287. IEEE Computer Society, 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 *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–143, New York, NY, USA, 1997. ACM Press.

[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. Adaptive history-based memory schedulers. In *Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 343–354, December 2004.

[10] I. Hur and C. Lin. Adaptive history-based memory schedulers for modern processors. *IEEE Micro*, 26(1):22–29, 2006.

[11] 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, Washington, DC, USA, 1997. IEEE Computer Society.

[12] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, New York, NY, USA, 1997. ACM Press.

[13] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, New York, NY, USA, 1990. ACM Press.

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

[15] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368, Washington, DC, USA, 1998. IEEE Computer Society.

[16] W. F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA '01: Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 301–312, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[17] W. F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak. Filtering superfluous prefetches using density vectors. In *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, pages 124–132, Washington, DC, USA, 2001. IEEE Computer Society.

[18] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA '04: 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 *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[20] K. Rajamani. Memsim users' guide, IBM research report. Technical Report RC23431, October 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 *ISCA '02: 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 *ISCA '06: Proceedings of the 33th Annual International Symposium on Computer Architecture*, pages 252–263, New York, NY, USA, 2006. ACM Press.

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

[26] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 388–398, New York, NY, USA, 2003. ACM Press.

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

[28] 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.