Prefetching in an Intelligent Memory Architecture Using a Helper Thread [*]

Yan Solihin[†], **Jaejin Lee**[‡], and **Josep Torrellas**[†]
[†]University of Illinois at Urbana-Champaign
[‡]Michigan State University
{solihin,torrellas}@cs.uiuc.edu
jlee@cse.msu.edu
http://iacoma.cs.uiuc.edu/flexram

Abstract

Data prefetching is a popular technique for tolerating long memory access latencies. In this paper, we introduce a novel type of prefetching: memory-side correlation prefetching implemented in a user-level thread. The prefetching thread runs on a general-purpose processor embedded in the main memory. By allocating the correlation table in main memory, we can afford the large space required by the table. In addition, the scheme can be supported with few modifications to the L2 cache and no modification to the main processor core. We introduce a new organization of the correlation table and a new prefetching algorithm that enable fast and accurate far-ahead prefetching with high coverage. Overall, our evaluation shows that the algorithm effectively prefetches irregular applications, speeding up three applications by an average of 1.28. Furthermore, our scheme can work synergistically with a conventional processor-side prefetcher to deliver an average speedup of 1.36.

1 Introduction

Data prefetching is a popular technique to tolerate long memory access latencies. There have been many proposals using a *helper thread* to help prefetching for the *main thread*, such as [12, 15]. These proposals have focused on either SMT or CMP platforms. In this paper, we propose a prefetching thread scheme that is suitable for implementation in an *Intelligent Memory Architecture* (IMA). In IMA, the memory system is augmented with one or more memory processors. The nature of the problems in IMA is quite different than in SMT or CMP platforms. First, in SMT/CMP, *Processor-Side* prefetching is used, while in IMA, *Memory-Side* prefetching is used, because prefetch requests are generated by the processor in the main memory. Secondly, communication between the threads is cheap in SMT/CMP, while it is expensive in IMA. Thus, a suitable prefetching scheme is one that operates autonomously and that can be effective with coarse-grain communication between the prefetching and the main threads. In this work, we implement the prefetcher as a user-level thread that can prefetch irregular applications effectively using correlation prefetching algorithms. The only communication needed by the prefetching thread is the miss address stream of the main thread.

Memory-side prefetching is attractive for several reasons. First, it eliminates the overheads that prefetch requests and state bookkeeping introduce in the paths between the main processor and its caches. Secondly, it can be supported with very few modifications to the L2 cache and no modification to the main processor core. Thirdly, the prefetcher can exploit its proximity to the memory to its advantage. Memory-side prefetching has the additional attraction of riding the technology trend of increased chip integration. Indeed, popular platforms like PCs are being equipped with graphics engines in the memory system [16]. Some chipsets, like NVIDIA's nForce [13] even integrate a powerful processor in the North Bridge chip. Similar engines can be provided for prefetching, or existing graphics processors can be reused for prefetching when under-utilized. Moreover, there are proposals to integrate processing logic in DRAM chips, such as IRAM [8].

Using an engine for memory-side prefetching has been proposed elsewhere [1, 2, 4, 13, 14, 16, 18]. However, in most cases, these engines perform either very simple operations or highly-specific operations, such as prefetching linked data structures [4, 18]. Instead, what we would like, is a very flexible, general-purpose prefetcher.

While a memory-side prefetcher can support a variety of prefetching algorithms, one type that is particularly suitable is Correlation Prefetching [1, 3, 5, 11]. Correlation prefetching relies on correlation of miss addresses to predict and prefetch future misses based on the current state. Because the only information the prefetch thread needs is the miss address stream, correlation prefetching is suitable for an IMA platform.

In the past, general correlation prefetching has been supported by hardware controllers that require a large dedicated hardware table structure [1, 3, 5, 11]. In all but one case, these controllers have been placed between the L1 and L2 caches or between the L1 and the processor. While effective, the approach has a very high hardware cost. Furthermore, it does not prefetch far enough and tends to have a low coverage.

[*] This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, and EIA-0072102, by DARPA under grant F30602-01-C-0078, and by NCSA, Michigan State University, and gifts from IBM and Intel.

This paper introduces a novel prefetching scheme where memory-side correlation prefetching algorithms are implemented in software by using a user-level thread. The algorithms run on a general-purpose processor in the main memory system. The scheme allows prefetching algorithms to evolve with the applications, even after the computer system is shipped. In addition, the system can be supported with few modifications to the L2 cache, and no modifications to the main processor core.

We introduce a new organization of the correlation table and a new correlation prefetching algorithm that enable fast and far-ahead prefetching, with high coverage and accuracy. By allocating the correlation table in main memory, we can afford the large space required by the table. We demonstrate that the software algorithm can effectively prefetch data for irregular applications. Indeed, our scheme speeds up three SPECInt2000 applications by an average of 1.28. We also show that our scheme can work synergistically with a conventional processor-side prefetcher to deliver an average speedup of 1.36.

The rest of the paper is organized as follows: Section 2 discusses memory-side prefetching and correlation prefetching; Section 3 presents our design; Section 4 discusses our evaluation setup; Section 5 evaluates our design; and Section 6 concludes.

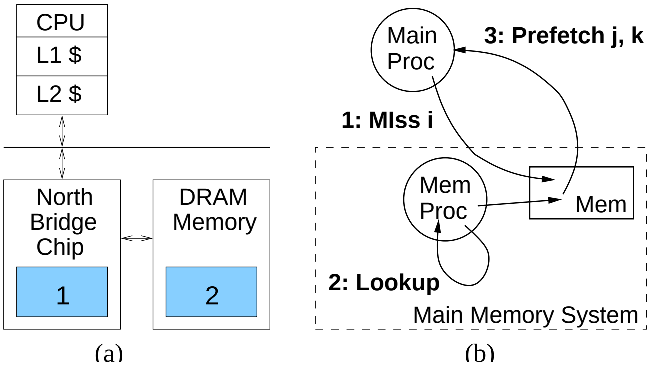
2 Related Issues

2.1 Memory-Side Prefetching

Memory-Side prefetching occurs when prefetching is initiated by one or a set of engines that reside in or beside the main memory (definitely beyond any memory bus). Chip manufacturers have integrated hardwired controllers that probably recognize very simple sequences like strides, such as NVIDIA's DASP engine in the North Bridge chip [13] and Intel's prefetch cache in its i860 chipset.

In this paper, we propose to use a simple general-purpose memory processor for memory-side prefetching. Although this idea is applicable to a generic memory system, we will illustrate it on a PC-like memory system depicted in Figure 1-(a). The memory processor can be placed in several places, such as in the North Bridge (Memory Controller) chip (1), or in the DRAM chips (2). The advantages of the first case are that it is simple to support, because the DRAM interface is not modified, and that the memory processor can be employed for other uses, such as a graphics engine. The second case, although more complicated to support, has the advantage of lower memory access latency and higher memory bandwidth due to higher integration. In this paper, we study the performance potential of the DRAM case.

Memory- and processor-side prefetching are not the same as *Push* and *Pull* (or *on-demand*) prefetching [18], respectively. *Push* prefetch occurs when prefetched data is sent to a cache or processor that has not requested it, while *pull* prefetch is the opposite. Clearly, a memory prefetcher can act as a *pull* prefetcher, by simply storing the prefetched data in a local buffer and supplying it to the processor on demand. In general, however, memory-side prefetching is most interesting when it performs *push* prefetching to the caches of the processor, because it can hide a larger fraction of memory access latency.



(a) (b)

Figure 1: Architecture of the system (a), and actions of the prefetches (b).

In our system, the memory processor observes the requests from the main processor that reach main memory. Based on them, and after examining some internal state, the memory processor prefetches other lines that it expects the main processor to need in the future (Figure 1-(b)).

In this paper, we concentrate on *push prefetching* into the L2 cache. Since the memory processor only sees L2 cache miss streams, it aims to eliminate L2 cache misses by pushing the prefetched data into the L2 cache. L2 cache miss penalty is the largest component of memory access latency, and it is the hardest to hide, even by an out-of-order processor.

Our scheme is inexpensive to support. The main processor core does not need to be modified at all. The L2 cache needs to have the following supports. First, as in many other systems [4, 7], the L2 cache controller has to be able to accept lines from the memory system that it has not requested. To do so, the L2 has to assign unused *Miss Status Handling Registers* (MSHRs) [10] to such lines. Secondly, if the L2 has a pending request for the same line when a prefetch arrives, the prefetch simply steals the MSHR and updates the cache as if it were the reply. Finally, a prefetched line arriving at L2 is dropped in the following cases: the L2 cache already has a copy of the line, the write back queue has a copy of the line because the L2 is trying to write it back to memory, all MSHRs are full, or all the lines in the set where the prefetch line wants to go are in pending state.

2.2 Correlation Prefetching

Correlation Prefetching uses the current state of the reference or miss stream to predict and prefetch future misses. Two popular correlation schemes are the *Stride-Based* and *Pair-Based* schemes. The former tries to find a stride pattern in the miss stream and prefetch all the locations that would be accessed if the pattern continues in the future. The latter tries to identify a correlation between pairs of misses, for example between a miss and its immediate successor. It basically records a sequence of miss addresses in a table, and later when it encounters the head of the sequence, it looks up the table and prefetches the rest of the sequence. What makes pair-based schemes attractive is their general applicability, i.e. they work for any miss sequences that repeat. This is true for regular applications and for a wide range of irregular applications such as those that operate on sparse matrices and linked data structures. Furthermore, the schemes can be employed without any compiler support or changes in the application binaries.

Pair-based correlation prefetching has only been studied using a hardware implementation of prefetch engines [1, 3, 5, 11, 17], usually by placing the engine between the L1 and L2 cache [3, 5, 11, 17]. These studies have demonstrated the applicability of pair-based correlation prefetching on a wide variety of applications. However, they also reveal shortcomings of the approach. One critical problem is that to be effective, it needs large storage space to match the footprints of the applications. One and two Megabytes of dedicated on-chip SRAM tables have been proposed [5, 11], while some applications with larger footprints even need a 7.6 MB off-chip SRAM table [11]. Furthermore, it does not prefetch far enough and has low coverage (unless it is tightly coupled to the main processor and uses more fine grain information [11]). For example, for each miss, Joseph and Grunwald only store *immediate* successors [5]. The coverage is low because it needs one miss to trigger the prefetcher to prefetch the successor of the miss. At best only half of the misses can be eliminated. This scheme uses a wide table that stores many successors per miss and continuously rebuilds the table to increase the coverage. However, it causes excessive useless prefetches.

3 Proposed Scheme

Pair-based correlation prefetching is suitable for our memory-side prefetching system to support because it has general applicability and can be supported inexpensively. We show that shortcomings of the current correlation prefetching schemes can be eliminated by improving the correlation algorithms and implementing them in software. The algorithms described are implemented in a prefetching thread running on the memory processor. The code for the prefetching thread is written in C and hand-optimized for minimal prefetch response and occupancy time.

In the following sections, we discuss the concepts (Section 3.1), the architecture (Section 3.2), pair-based correlation prefetching algorithms (Section 3.3), and conventional processor-side prefetching (Section 3.4).

3.1 Concepts

Prefetching algorithms are implemented as a user-level helper thread that we call *prefetching thread*. The actions of the memory processor are determined by the behavior of the prefetching thread that we implement. The operation of the prefetching thread can be conceptually divided into two phases: *learning* and *prefetching*. In the learning phase, the prefetching thread records the L2 read and write miss patterns that it observes in a correlation table, one miss at a time. In the prefetching phase, every time that the prefetching thread sees a miss, it looks up the correlation table and prefetches several memory lines to the L2 cache of the main processor. No action is taken on a write-back memory access. In practice, as in [5], we found that combining the learning and prefetching phases enables the correlation table to quickly learn new patterns and provides the best performance in most cases (Figure 2).

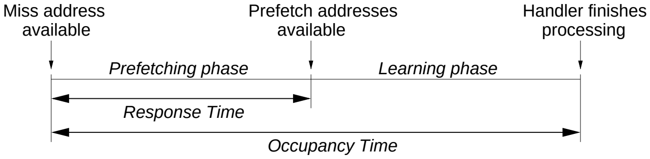


Figure 2: Timing of the prefetching thread.

The prefetching algorithm can be characterized by its *response time* and *occupancy time* (Figure 2). The response time is defined as the time beginning when the prefetching thread obtains a miss address until the prefetching thread produces the prefetch addresses. The occupancy time is the time the prefetching thread is busy and cannot process another miss address. As can be seen in the figure, the prefetching phase is always executed before the learning phase to minimize the response time. For the software implementation to be viable, the occupancy time has to be smaller than the average time between two consecutive L2 cache misses. Also, for best performance, the response time needs to be as small as possible.

By using a prefetching thread that stores the correlation table in the main memory, we eliminate the high hardware cost required by the table in the traditional implementation. We further address the inadequacies of traditional correlation prefetching, namely low prefetching coverage, and not prefetching far enough, by improving the correlation algorithms (Section 3.3).

3.2 Architecture of the System

When we integrate the memory processor in the DRAM chips, the DRAM chips and possibly the DRAM interface need to be modified. Extra complexities in handling multiple DRAM chips must also be addressed. Our goal in this paper is to study the performance potential of this case. Consequently, we abstract away the implementation complexity of integrating the processor in the DRAM by assuming a single chip main memory with a single memory processor in it (Figure 3).

The key communication occurs through queues *1*, *2*, and *3*. Miss requests from the main processor are deposited in queues *1* and then in *2*. In the learning phase, the memory processor uses the entries in queue *2* to build its state. In the prefetching phase, the memory processor uses the entries in queue *2* and its state to generate addresses to prefetch. The

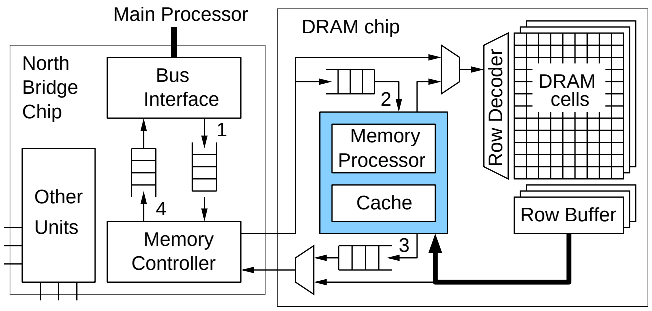


Figure 3: Microarchitecure a DRAM chip that includes a memory processor used for correlation prefetching.

lines prefetched are deposited in queue *3*. If the memory processor suffers a cache miss on its correlation table structure, it accesses the DRAM directly. Queue *4* is in the replying path from memory to the main processor.

3.3 Pair-Based Correlation Algorithms

We now discuss the pair-based correlation prefetching algorithms. We consider two different organizations for the correlation table: a basic one that does not allow data replication and a more advanced one that allows replication. Their use gives rise to different algorithms. We consider them in turn.

Pair-Based Algorithms with Basic Table Organization

Each row in this table stores the tag of the miss address, and the addresses of a set of *immediate* successor misses stored in MRU order. We consider two algorithms that use this basic organization: *Base* and *Chain*.

Base follows the scheme proposed by Joseph and Grunwald [5]. For any given miss, *Base* is only interested in prefetching *immediate* successor misses. The parameters of the algorithm are the number of immediate successors predicted (*NumSucc*), the number of misses that the correlation table can store predictions for (*NumRows*), and the associativity of the correlation table (*Assoc*).

Base is illustrated in Figure 4-(a). It shows two snapshots of the correlation table at the point that the corresponding miss trace has been consumed (i and ii). In the example, *NumSucc* is 2, *NumRows* is 4, and *Assoc* is 1. Within a row, successors are replaced using LRU replacement policy. As in Joseph and Grunwald's study [5], we find that LRU replacement policy for the successors in each row works best. The figures show the successors in MRU order from left to right. In the learning phase, the processor keeps a pointer to the row of the last miss observed. When a miss occurs, its address is placed as one of the immediate successors of the last miss, and a new row is allocated for the new miss unless an entry for the address already exists. In the prefetching phase (iii), when a miss is observed, the processor finds the corresponding row and prefetches all the *NumSucc* immediate successors, starting from the MRU one.

Since *Base* only prefetches immediate successors, its coverage and latency hiding capabilities are limited. To improve this, we propose the *Chain* algorithm, which for every miss prefetches multiple levels of successors. The algorithm takes one extra parameter called *NumLevels*, which is the number of levels of successors prefetched. The algorithm is illustrated in Figure 4-(b).

In the learning phase, *Chain* is identical to *Base* (i and ii). However, *Chain* does more work in the prefetching phase (iii). After prefetching the row of immediate successors, it takes the most recently-used successor among them and indexes the correlation table with its address. If the entry is found, it prefetches all *NumSucc* successors there. Then, it takes the most recently used successor in that row and repeats the process for *NumLevels-1* times. As an example, suppose that a miss on line *a* occurs (iii). The memory processor first prefetches *d* and *b*. Then, it takes the MRU entry *d*, looks-up the table, and prefetches *d*'s successor, *c*.

While improving the coverage and far-ahead prefetching capability over *Base*, *Chain* has two limitations. One limitation is that the response time of the algorithm is high. To issue prefetches in response to a miss, it needs to make *NumLevels* accesses to different rows in the table, each possibly involving a low-associative search and potentially causing a cache miss. The second limitation is that it does not prefetch the *correct MRU* successors of each level of successors. Instead, it only prefetches successors found along the MRU path.

Pair-Based Algorithms with Replicated Table Organization

Each row in this table stores the tag of the miss address, and *NumLevels* levels of successors. Each level contains *NumSucc* addresses, which are MRU-ordered.

We propose a a new algorithm called *Replicated* that exploits this table organization. Replicated takes the same parameters as *Chain*. In the learning phase, *NumLevels* pointers to the table are kept for efficient access, pointing to the rows for the address of the last miss, second last, and so on. When a miss occurs, its address is recorded in the correct position of MRU successors of the last few misses by using these pointers. Figures 4-(c) illustrates the algorithm. In the example, *NumSucc* is 2, *NumRows* is 4, *Assoc* is 1, and *NumLevels* is 2. The figure shows two snapshots of the correlation table in the learning phase at the point where the corresponding miss trace has been consumed (i and ii). The figure also shows the position of the two pointers, and the algorithm in prefetching phase (iii).

Note that this organization solves the two problems of *Chain*. First, the response time is much shorter. We can prefetch several levels of successors with a single row access, possibly with only one cache miss. In fact, we shift some computation from the prefetching phase, which is the critical phase, to the learning phase. Now the learning phase needs to update several rows in the table. However, the rows are most likely still in the cache and, since we keep the pointers to the entries of last few miss addresses, the associative search is avoided. Secondly, by grouping together all the successors from a given level, we can identify the correct MRU successors from that level, yielding higher accuracy.

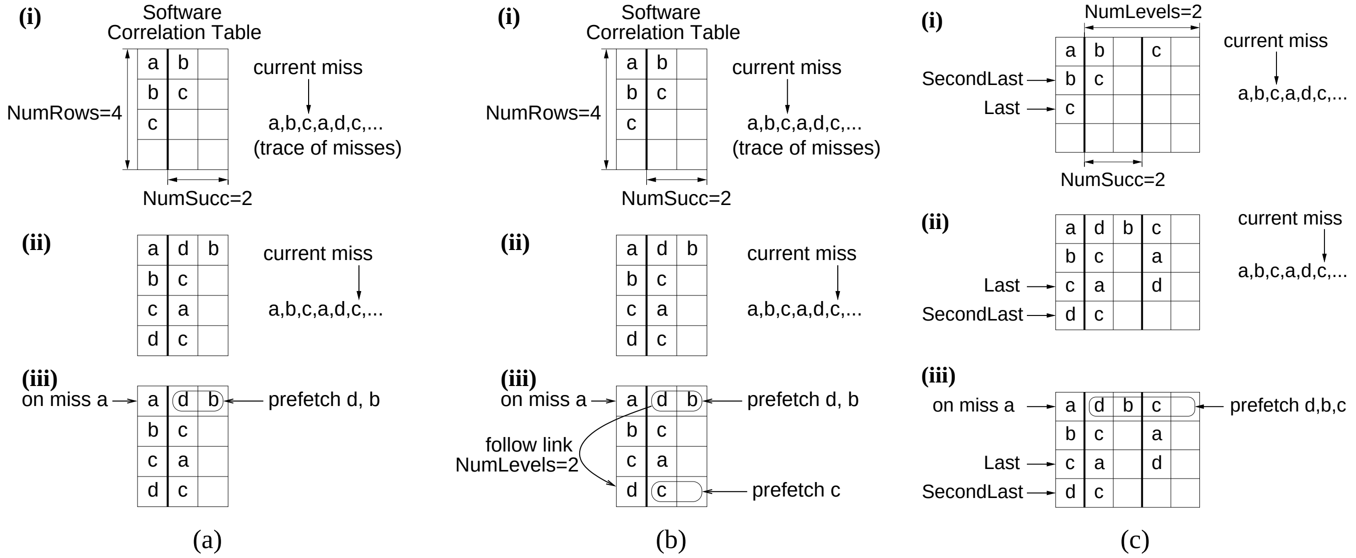


Figure 4: Pair-based correlation algorithms: *Base* (a), *Chain* (b), and *Replicated* (c).

Characteristics	*Base*	*Chain*	*Replicated*
Levels of successors prefetched	1	$NumLevels$	$NumLevels$
Full MRU ordering for each level?	Yes	No	Yes
Num. row accesses in the prefetching phase (SEARCH)	1	$NumLevels$	1
Num. row accesses in the learning phase (NO SEARCH)	1	1	$NumLevels$
Response Time	Low	High	Low
Space requirement (for constant number of prefetches)	x	x	$NumLevels \times x$

Table 1: Comparing the different pair-based algorithms.

Algorithm Comparison

Table 1 compares the three pair-based schemes. From the table, we see that *Replicated* algorithm tries to solve problems in current correlation prefetching algorithms: it looks far ahead by prefetching several levels of successors, thereby improving coverage, while keeping high accuracy by prefetching the correct MRU successors in each level. Its only shortcoming is its high space requirements for the correlation table. Fortunately, this is a minor issue, since the table is allocated in the main memory.

The response time is better with the *Replicated* algorithm than with the *Chain* algorithm. The handler in *Replicated* runs very efficiently because cache lines are well utilized. Note that all the correlation algorithms could be implemented in hardware. However, *Replicated* is very suitable for a software implementation because it has a low response time, far-ahead prefetching capability, and uses cache lines well.

3.4 Conventional Prefetching

Previous studies found that placing a stride-based prefetcher as a front end of a pair-based prefetcher makes pair-based prefetching more effective [3, 17]. We exploit this finding by including processor-side prefetching in the form of a hardware multi-stream sequential prefetcher at the L1 cache. The prefetcher has similar capabilities to stream buffers [6], except that the prefetch lines are put directly in the L1 cache.

In our system, we assume that the memory controller can distinguish the prefetches issued by the processor-side prefetcher from regular misses. The memory controller chooses not to pass such prefetches to the memory processor. As a result, in general, the processor-side prefetcher targets the regular misses while the memory-side prefetcher targets the irregular ones.

4 Evaluation Environment

Applications. To evaluate our prefetching scheme, we use three mostly irregular memory-intensive applications from the SPECInt2000 suite. Irregular applications are hardly amenable to compiler-based prefetching. Consequently, they are the obvious target for the type of prefetching that we propose. We choose *Gap*, *Mcf*, and *Parser*. *Gap* uses a subset of the test input set, *Mcf* uses the test input set, and *Parser* uses a subset of the train input set.

Simulation Environment. The evaluation is performed using execution-driven simulation. Our environment is based on an extension to MINT that supports dynamic superscalar processor models with register renaming, branch prediction, and non-blocking memory operations [9].

The architecture modeled is that of a high-end PC with a

Main Proc	6-issue dynamic, 1.6 GHz. Int, fp, ld/st FU: 4,4,2. Pending ld/st: 8/16. Branch penalty: 12 cycles. L1 data: write-back, 16 KB, 2 way, 32-B line, 3-cycle hit RT. L2 data: write-back, 512 KB, 4 way, 64-B line, 19-cycle hit RT. RT memory latency: 243 cycles (row miss), 208 cycles (row hit). Main memory bus: split-transaction, 8-B wide, 400 MHz, 3.2 GB/sec peak.
Mem Proc in DRAM	2-issue dynamic, 800 MHz. Int, fp, ld/st FU: 2,2,1. Pending ld/st: 4/4. Branch penalty: 6 cycles. L1 data: write-back, 32 KB, 2 way, 32-B line, 4-cycle hit RT. RT memory latency: 56 cycles (row miss), 21 cycles (row hit). Internal DRAM data bus: 32-B wide, 800 MHz, 25.6 GB/sec.
DRAM parameters	Dual channel; each channel 2-B wide, 800 MHz; total 3.2 GB/sec peak. Random access time (*tRAC*) 45 ns; from Mem Controller (*tSystem*) 60 ns.
Other	Depth of queues *1* through *4*: 16.

Table 2: Parameters of the simulated architecture. Latencies correspond to contention-free conditions. *RT* stands for round-trip *from the processor*. All cycles are 1.6 GHz cycles. 512-KB L2 cache is chosen for the main processor because we run small inputs for the applications.

memory processor that is integrated in the DRAM, following the microarchitecture of Figure 3. Table 2 shows the parameters used for each component of the architecture. The architecture is modeled cycle by cycle, including contention effects.

In the simulation, both the application thread and the prefetching thread are run simultaneously. We model the contention between the two threads on memory subsystems that are shared (memory controller, DRAM channels, DRAM banks, etc.). The simulation includes all overheads incurred by running the two threads on different processors.

Algorithm Parameters. Table 3 shows the default parameter values that we use for the algorithms described in Section 3.2. For the *Base* algorithm, we use the values similar to what Joseph and Grunwald use for their system [5] to make the comparison easier. For all the algorithms, we use *NumRows* = 64K, which results in a table of size 1.3 MBytes, 0.66 MBytes, and 1.8 MBytes for *Base*, *Chain*, and *Repl*, respectively. These sizes are very tolerable, since the table is a plain software data structure that is stored in main memory, is dynamically allocated, and is cached by the memory processor.

The conventional prefetching discussed in Section 3.4 takes two parameters: the number of streams it is able to prefetch simultaneously (*NumSeq*) and the number of prefetches that it issues per miss in a sequence observed (*NumPref*). We implement this algorithm in hardware in the L1 cache (*Conven4*) and also in software running on the memory processor (*Seq1* and *Seq4*).

Algorithm	Label	Parameter Values
Base	*Base*	*NumSucc* = 4, *Assoc* = 4
Chain	*Chain*	*NumSucc* = 2, *Assoc* = 2, *NumLevels* = 3
Replicated	*Repl*	*NumSucc* = 2, *Assoc* = 2, *NumLevels* = 3
Conventional 1-Stream	*Seq1*	*NumSeq* = 1, *NumPref* = 6
Conventional 4-Stream	*Seq4*	*NumSeq* = 4, *NumPref* = 6
Conventional 4-Stream	*Conven4*	*NumSeq* = 4, *NumPref* = 6

Table 3: Parameter values used in the algorithms.

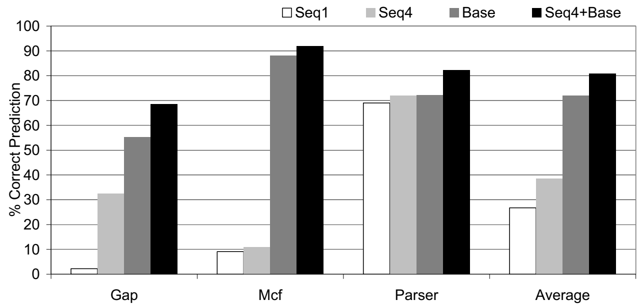


Figure 5: Characterizing the predictability of misses.

5 Evaluation

To evaluate our prefetching scheme, we first characterize the behavior of applications (Section 5.1) and then compare the performance of different algorithms (Section 5.2).

5.1 Characterizing Application Behavior

For memory-side correlation prefetching to be effective, the miss address streams have to be predictable. In this experiment, we record the fraction of L2 cache misses that are correctly predicted. For a sequential scheme, this means that the upcoming address exactly matches the one predicted, while for a pair-based scheme, the upcoming address matches one of the predicted successors. The thread does not perform prefetching here and it only observes the addresses of all L2 cache misses.

In our experiments, shown in Figure 5, we record the fraction of L2 cache misses that are correctly predicted. We try stride-based schemes that detect up to one stream (*Seq1*) and four streams (*Seq4*), the *Base* algorithm, and the combination.

The figure shows that the miss stream is largely predictable, with *Seq4*, *Base*, and *Seq4+Base* correctly predicting roughly 40%, 70%, and 80% of the misses on average, respectively. However, the predictability of each application differs. For example, *Mcf* does not have sequential patterns, while *Parser* has mostly sequential patterns, and *Gap* is mixed.

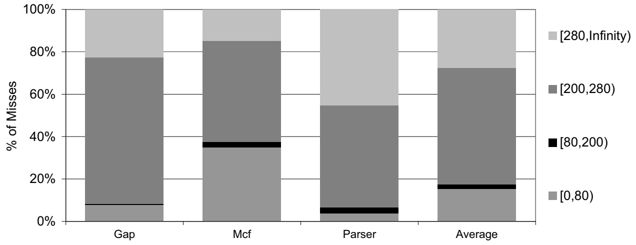


Figure 6: Characterizing the time between consecutive misses.

Seq4 always outperform *Seq1*, indicating that multiple

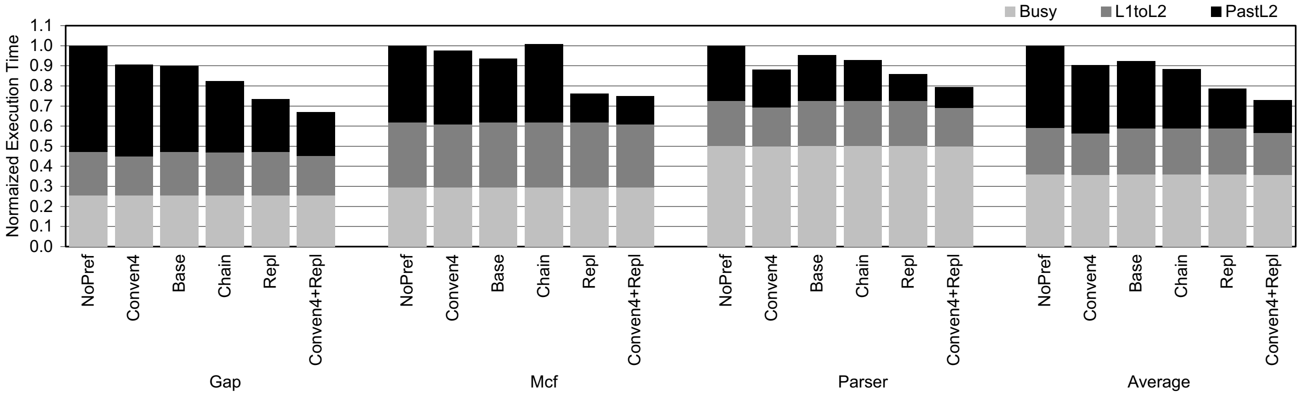


Figure 7: Execution time of the different algorithms.

stream support is necessary for a sequential scheme. The figure shows that in all applications, *Base* is almost as good as the combination *Seq4+Base*. This is because a correlation table is able to detect both sequential and irregular patterns, as long as the patterns repeat. Once the table learns a pattern, it can predict it effectively. However, it is still beneficial to have a multi-stream sequential prefetcher at the processor-side for several reasons: it does not need learning, it can be cheaply implemented, and it can hide the full memory latency if integrated with the L1 cache. Furthermore, it splits the misses into regular and irregular streams, and by tackling the regular one, it removes some load from the memory prefetcher.

We now consider the time between misses. Figure 6 classifies the misses according to the number of cycles between two consecutive misses arriving at the memory. The misses are grouped in bins corresponding to [0,80) 1.6 GHz processor cycles, [80,200), etc. The most significant bins in the figure are [200,280), [280,∞), and [0,80), which contribute on average to 54%, 28%, and 18% of all miss distances. The misses with distances between 200 and 280 are critical as they are both frequent and hard to hide even with out-of-order processors. Furthermore, since the round-trip memory latency is between 208 and 243 cycles, dependent misses are likely to fall in this bin. This characterization suggests that, to be on the safe side, occupancy time of the prefetching algorithm should be less than 200 cycles.

The [0,80) bin contains misses that may not give enough time for our prefetching thread to respond. Fortunately, these misses are not frequent and are likely to be overlapped with each other or with computation. Thus, they harm the performance much less than the bin size implies.

5.2 Comparing the Different Algorithms

Figure 7 compares the execution time of the applications in different cases: no prefetching (*NoPref*), hardware processor-side L1 prefetching as shown in Table 3 (*Conven4*), different software memory-side prefetching schemes as shown in Table 3 (*Base*, *Chain*, and *Repl*), and the combination of *Conven4* and *Repl* (*Conven4+Repl*). For each application and the average, the bars are normalized to *NoPref*. They are broken down into miss stall time past the L2 cache (*PastL2*), miss stall time between the L1 and L2 caches (*L1toL2*), and the remaining time (*Busy*) that represents processor computation plus various pipeline stalls.

On average, the *PastL2* time is the most significant component of the execution time, contributing about 40%, while *Busy* and *L1toL2* follow with 35% and 25%, respectively. Thus, although our software scheme can only target L2 cache misses, we are targeting the main performance bottleneck.

The conventional scheme (*Conven4*) performs well on applications with some sequential patterns, such as *Gap* and *Parser*, but is ineffective in the application that has purely irregular patterns (*Mcf*). On average, *Conven4* reduces the execution time by 10%.

The pair-based schemes show mixed performance. The *Base* scheme, modeled after Joseph and Grunwald's, shows limited speedups because it does not prefetch far enough. *Chain* performs slightly better than *Base*, but is limited by inaccuracy and high response time. *Repl* is able to reduce the execution time significantly. It outperforms both *Base* and *Chain* in all applications. Its impact comes from the nice properties of the *Replicated* algorithm, as discussed in Section 3.

The combined scheme (*Conven4+Repl*) performs the best. Its impact is significant: it removes on average 60% of *PastL2* stall time, providing an average speedup of 1.36. Compared to processor-side prefetching only (*Conven4*) with an average speedup of 1.11, and memory-side prefetching only (*Repl*) with an average speedup of 1.28, there is a clear synergistic effect in the combined scheme. Memory-side prefetching helps processor-side prefetching in irregular patterns, while processor-side prefetching helps in regular patterns.

Workload of the Prefetching Thread

We can gain further insight by examining the work load of the prefetching thread. Figure 8 shows the average response time and occupancy of the prefetching thread for each of the memory-side prefetching algorithm. The latencies are shown in 1.6 GHz cycles and correspond to the average of all applications. Each bar is broken down into computation time (*Busy*) and memory stall time (*Mem*). The numbers on top of each bar show the average IPC of the prefetching thread. The IPC is calculated as the number of instructions divided by the number of memory processor cycles.

The figure shows that for all the algorithms, the occupancy time is less than 200 cycles, showing the viability of the software implementation. *Chain* and *Repl* have the lowest occupancy time. Due to the fewer associative searches and the better cache use, *Repl* has only slightly higher occupancy time compared to *Chain*, despite performing more table updates.

The response time is very important for prefetching effectiveness. The figure shows that *Repl* has the lowest response time. its value is around 30 cycles.

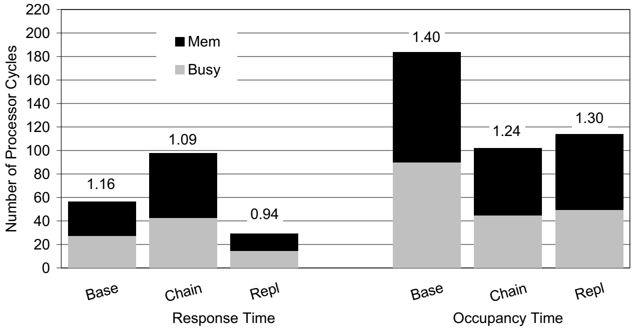


Figure 8: Response and occupancy time of the prefetching thread for each of the prefetching algorithm.

6 Conclusions

This paper introduced memory-side correlation-based prefetching implemented in a user-level thread. The scheme runs on a general-purpose processor in the main memory. The scheme can be supported with few modifications to the L2 cache and no modification to the main processor. We introduced a new organization of the correlation table and a new correlation prefetching algorithm that enable fast and accurate far-ahead prefetching with high coverage. Overall, our scheme effectively prefetched irregular applications, speeding up three SPECInt2000 applications by an average of 1.28. Furthermore, our scheme can work synergistically with a conventional processor-side prefetcher to deliver an average speedup of 1.36.

Acknowledgement

We thank James Tuck, Jose F. Martinez, Jose Renau, and Michael Huang for contributions to this work.

References

[1] T. Alexander and G. Kedem. Distributed Predictive Cache Design for High Performance Memory Systems. In *Proceedings of the 2nd HPCA*, Feb 1996.

[2] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, E.L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M.A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *Proceedings the 5th HPCA*, January 1999.

[3] M.J. Charney and A.P.Reeves. Generalized Correlation Based Hardware Prefetching. *Technical Report EE-CEG-95-1, Cornell University*, Feb 1995.

[4] C.J. Hughes. Prefetching Linked Data Structures in Systems with Merged DRAM-Logic. Master's thesis, University of Illinois at Urbana-Champaign, May 2000. URL: http://rsim.cs.uiuc.edu/∼cjhughes/cjhughesmsthesis.pdf.

[5] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *Proceedings of the 24th ISCA*, June 1997.

[6] 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 ISCA*, pages 388–397, 1990.

[7] D. Koufaty and J. Torrellas. Comparing Data Forwarding and Prefetching for Communication-Induced Misses in Shared-Memory MPs. In *Proceedings of the ICS*, July 1998.

[8] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, September 1997.

[9] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[10] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th ISCA*, pages 87–85, 1981.

[11] A.-C. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *Proceedings of the 28th ISCA*, 2001.

[12] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th ISCA*, 2001.

[13] NVIDIA. http://www.nvidia.com.

[14] R. Cooksey, D. Colarelli, and D. Grunwald. Content-based Prefetching: Initial Results. In *The 2nd Workshop on Intelligent Memory Systems*, Nov 2000.

[15] A. Roth and G.S. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the 7th HPCA*, pages 37–48, Jan 2001.

[16] Sony Computer Entertainment Inc. http://www.sony.com.

[17] T. Sherwood, S. Sair, and B. Calder. Predictor-Directed Stream Buffers. In *Proceedings of the 33th MICRO*, Dec 2000.

[18] C.-L. Yang and A.R.Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *Proceedings of the 2000 ICS*, May 2000.