Using a User-Level Memory Thread for Correlation Prefetching*

Yan Solihin † **Jaejin Lee** ‡ **Josep Torrellas** †

† University of Illinois at Urbana-Champaign
‡ Michigan State University
http://iacoma.cs.uiuc.edu
http://www.cse.msu.edu/∼jlee

Abstract

This paper introduces the idea of using a User-Level Memory Thread (ULMT) for correlation prefetching. In this approach, a user thread runs on a general-purpose processor in main memory, either in the memory controller chip or in a DRAM chip. The thread performs correlation prefetching in software, sending the prefetched data into the L2 cache of the main processor. This approach requires minimal hardware beyond the memory processor: the correlation table is a software data structure that resides in main memory, while the main processor only needs a few modifications to its L2 cache so that it can accept incoming prefetches. In addition, the approach has wide usability, as it can effectively prefetch even for irregular applications. Finally, it is very flexible, as the prefetching algorithm can be customized by the user on an application basis. Our simulation results show that, through a new design of the correlation table and prefetching algorithm, our scheme delivers good results. Specifically, nine mostly-irregular applications show an average speedup of 1.32. Furthermore, our scheme works well in combination with a conventional processor-side sequential prefetcher, in which case the average speedup increases to 1.46. Finally, by exploiting the customization of the prefetching algorithm, we increase the average speedup to 1.53.

1. Introduction

Data prefetching is a popular technique to tolerate long memory access latencies. Most of the past work on data prefetching has focused on processor-side prefetching [6, 7, 8, 12, 13, 14, 15, 19, 20, 23, 25, 26, 28, 29]. In this approach, the processor or an engine in its cache hierarchy issues the prefetch requests. An interesting alternative is memory-side prefetching, where the engine that prefetches data for the processor is in the main memory system [1, 4, 9, 11, 22, 28].

Memory-side prefetching is attractive for several reasons. First, it eliminates the overheads and state bookkeeping that prefetch requests introduce in the paths between the main processor and its caches. Second, it can be supported with a few modifications to the controller of the L2 cache and no modification to the main processor. Third, the prefetcher can exploit its proximity to the memory to its advantage, for example by storing its state in memory. Finally, 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 [27]. Some chipsets like NVIDIA's nForce even integrate a powerful processor in the North Bridge chip [22]. Simpler engines can be provided for prefetching, or existing graphics processors can be augmented with prefetching capabilities. Moreover, there are proposals to integrate processing logic in DRAM chips, such as IRAM [16].

Unfortunately, existing proposals for memory-side prefetching engines have a narrow scope [1, 9, 11, 22, 28]. Indeed, some designs are hardware controllers that perform simple and specific operations [1, 9, 22]. Other designs are specialized engines that are custom-designed to prefetch linked data structures [11, 28]. Instead, we would like an engine that is usable in a wide variety of workloads and that offers flexibility of use to the programmer.

While memory-side prefetching can support a variety of prefetching algorithms, one type that is particularly suited to it is Correlation prefetching [1, 6, 12, 18, 26]. Correlation prefetching uses past sequences of reference or miss addresses to predict and prefetch future misses. Since no program knowledge is needed, correlation prefetching can be easily moved to the memory side.

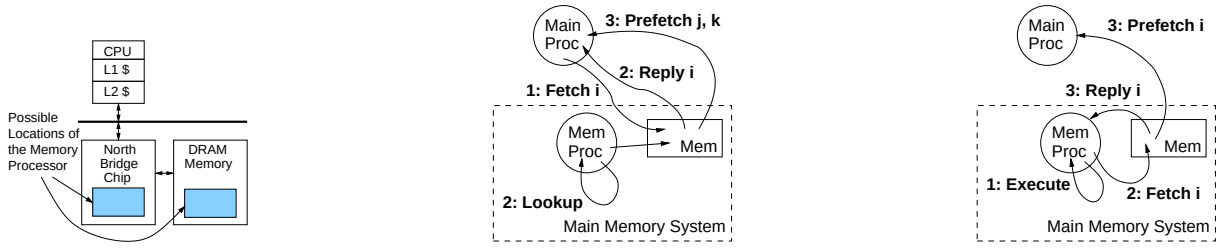
In the past, correlation prefetching has been supported by hardware controllers that typically require a large hardware table to keep the correlations [1, 6, 12, 18]. In all cases but one, these controllers are placed between the L1 and L2 caches, or between the processor and the L1. While effective, this approach has a high hardware cost. Furthermore, it is often unable to prefetch far ahead enough and deliver good prefetch coverage.

In this paper, we present a new scheme where correlation prefetching is performed by a User-Level Memory Thread (ULMT) running on a simple general-purpose processor in memory. Such a processor is either in the memory controller chip or in a DRAM chip, and prefetches lines to the L2 cache of the main processor. The scheme requires minimal hardware support beyond the memory processor: the correlation table is a software data structure that resides in main memory, while the main processor only needs a few modifications to its L2 cache controller so that it can accept incoming prefetches. Moreover, our scheme has wide usability, as it can effectively prefetch even for irregular applications. Finally, it is very flexible, as the prefetching algorithm executed by the ULMT can be customized by the programmer on an application basis.

Using a new design of the correlation table and correlation prefetching algorithm, our scheme delivers an average speedup of 1.32 for nine mostly-irregular applications. Furthermore, our scheme works well in combination with a conventional processor-side sequential prefetcher, in which case the average speedup increases to 1.46. Finally, by exploiting the customization of the prefetching algorithm, we increase the average speedup to 1.53.

This paper is organized as follows: Section 2 discusses memory-side and correlation prefetching; Section 3 presents ULMT for correla-

*This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; by Michigan State University; and by gifts from IBM, Intel, and Hewlett-Packard.



(a) (b) (c)

Figure 1. Memory-side prefetching: some locations where the memory processor can be placed (a), and actions under push passive (b) and push active (c) prefetching.

tion prefetching; Section 4 discusses our evaluation setup; Section 5 evaluates our design; Section 6 discusses related work; and Section 7 concludes.

2. Memory-Side and Correlation Prefetching

2.1. Memory-Side Prefetching

Memory-Side prefetching occurs when prefetching is initiated by an engine that resides either close to the main memory (beyond any memory bus) or inside of it [1, 4, 9, 11, 22, 28]. Some manufacturers have built such engines. Typically, they are simple hardwired controllers that probably recognize only simple stride-based sequences and prefetch data into local buffers. Some examples are NVIDIA's DASP engine in the North Bridge chip [22] and Intel's prefetch cache in the i860 chipset.

In this paper, we propose to support memory-side prefetching with a user-level thread running on a general-purpose core. The core can be very simple and does not need to support floating point. For illustration purposes, Figure 1-(a) shows the memory system of a PC. The core can be placed in different places, such as in the North Bridge (memory controller) chip or in the DRAM chips. Placing it in the North Bridge simplifies the design because the DRAM is not modified. Moreover, some existing systems already include a core in the North Bridge for graphics processing [22], which could potentially be reused for prefetching. Placing the core in a DRAM chip complicates the design, but the resulting highly-integrated system has lower memory access latency and higher memory bandwidth. In this paper, we examine the performance potential of both designs.

Memory- and processor-side prefetching are not the same as *Push* and *Pull* (or *On-Demand*) prefetching [28], respectively. Push prefetching occurs when prefetched data is sent to a cache or processor that has not requested it, while pull prefetching is the opposite. Clearly, a memory prefetcher can act as a pull prefetcher by simply buffering the prefetched data locally and supplying it to the processor on demand [1, 22]. 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 the memory access latency.

Memory-side prefetching can also be classified into *Passive* and *Active*. In passive prefetching, 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 data for the main processor that it expects the latter to need in the future (Figure 1-(b)).

In active prefetching, the memory processor runs an abridged version of the code that is running on the main processor. The execution of the code induces the memory processor to fetch data that the main processor will need later. The data fetched by these requests is also sent to the main processor (Figure 1-(c)).

In this paper, we concentrate on passive push memory-side prefetching into the L2 cache of the main processor. The memory processor aims to eliminate only L2 cache misses, since they are the only ones that it sees. Typically, L2 cache miss time is an important contributor to the processor stall due to memory accesses, and is usually the hardest to hide with out-of-order execution.

This approach to prefetching is inexpensive to support. The main processor core does not need to be modified at all. Its L2 cache needs to have the following supports. First, as in other systems [11, 15, 28], the L2 cache has to accept lines from the memory that it has not requested. To do so, the L2 uses free Miss Status Handling Registers (MSHRs) in such events. Secondly, if the L2 has a pending request and a prefetched line with the same address 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 cache is trying to write it back to memory, all MSHRs are busy, or all the lines in the set where the prefetched line wants to go are in transaction-pending state.

2.2. Correlation Prefetching

Correlation Prefetching uses past sequences of reference or miss addresses to predict and prefetch future misses [1, 6, 12, 18, 26]. Two popular correlation schemes are *Stride-Based* and *Pair-Based* schemes. Stride-based schemes find stride patterns in the address sequences and prefetch all the addresses that will be accessed if the patterns continue in the future. Pair-based schemes identify a correlation between pairs or groups of addresses, for example between a miss and a sequence of successor misses. A typical implementation of pair-based schemes uses a *Correlation Table* to record the addresses that are correlated. Later, when a miss is observed, all the addresses that are correlated with its address are prefetched.

Pair-based schemes are attractive because they have general applicability: they work for any miss patterns as long as miss address sequences repeat. Such behavior is common in both regular and irregular applications, including those with sparse matrices or linked data structures. Furthermore, pair-based schemes, like all correlation schemes, need neither compiler support nor changes in the application binary.

Pair-based correlation prefetching has only been studied using hardware-based implementations [1, 6, 12, 18, 26], typically by placing a custom prefetch engine and a hardware correlation table between the processor and L1 cache, or between the L1 and L2 caches. The typical correlation table, as used in [6, 12, 26], is organized as

follows. Each row stores the tag of an address that missed, and the addresses of a set of *immediate* successor misses. These are misses that have been seen to *immediately* follow the first one at different points in the application. The parameters of the table are the maximum number of immediate successors per miss (*NumSucc*), the maximum number of misses that the table can store predictions for (*NumRows*), and the associativity of the table (*Assoc*). According to [12], for best performance, the entries in a row should replace each other with a LRU policy.

Figure 4-(a) illustrates how the algorithm works. We call the algorithm *Base*. The figure shows two snapshots of the table at different points in the miss stream ((i) and (ii)). Within a row, successors are listed in MRU order from left to right. At any time, the hardware keeps a pointer to the row of the last miss observed. When a miss occurs, the table learns by placing the miss address as one of the immediate successors of the last miss, and a new row is allocated for the new miss unless it already exists. When the table is used to prefetch ((iii)), it reacts to an observed miss by finding the corresponding row and prefetching all *NumSucc* successors, starting from the MRU one.

The designs in [1, 18] work slightly differently. They are discussed in Section 6.

Overall, past work has demonstrated the applicability of pair-based correlation prefetching for many applications. However, it has also revealed the shortcomings of the approach. One critical problem is that, to be effective, this approach needs a large table. Proposed schemes typically need a 1-2 Mbyte on-chip SRAM table [12, 18], while some applications with large footprints even need a 7.6 Mbyte off-chip SRAM table [18].

Furthermore, the popular schemes that prefetch several potential *immediate* successors for each miss [6, 12, 26] have two limitations: they do not prefetch very far ahead and, intuitively, they need to observe one miss to eliminate another miss (its immediate successor). As a result, they tend to have low coverage. *Coverage* is the number of useful prefetches over the original number of misses [12].

3. ULMT for Correlation Prefetching

We propose to use a ULMT to eliminate the shortcomings of pair-based correlation prefetching while enhancing its advantages. In the following, we discuss the main concept (Section 3.1), the architecture of the system (Section 3.2), modified correlation prefetching algorithms (Section 3.3), and related operating system issues (Section 3.4).

3.1. Main Concept

A ULMT running on a general-purpose core in memory performs two conceptually distinct operations: *learning* and *prefetching*. Learning involves observing the misses on the main processor's L2 cache and recording them in a correlation table one miss at a time. The prefetching operation involves reacting to one such miss by looking up the correlation table and triggering the prefetching of several memory lines for the L2 cache of the main processor. No action is taken on a write-back to memory.

In practice, in agreement with past work [12], we find that combining both learning and prefetching works best: the correlation table continuously learns new patterns, while uninterrupted prefetching delivers higher performance. Consequently, the ULMT executes the infinite loop shown in Figure 2. Initially, the thread waits for a miss to be observed. When it observes one, it looks up the table and generates the addresses of the lines to prefetch (*Prefetching Step*). Then, it updates the table with the address of the observed miss (*Learning Step*). It then resumes waiting.

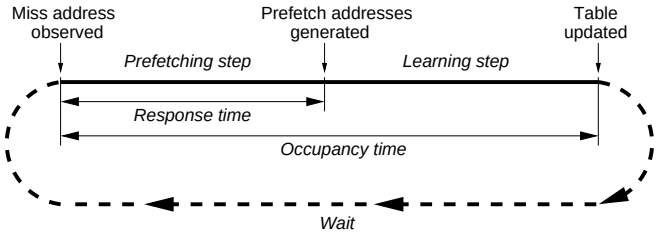


Figure 2. Infinite loop executed by the ULMT.

Any prefetch algorithm executed by the ULMT is characterized by its *Response* and *Occupancy* times. The response time is the time from when the ULMT observes a miss address until it generates the addresses to prefetch. For best performance, the response time should be as small as possible. This is why we always execute the Prefetching step before the Learning one. Moreover, we shift as much computation as possible from the Prefetching to the Learning step, retaining only the most critical operations in the Prefetching step.

The occupancy time is the time when the ULMT is busy processing a single observed miss. For the ULMT implementation of the prefetcher to be viable, the occupancy time has to be smaller than the time between two consecutive L2 misses most of the times.

The correlation table that the ULMT reads and writes is simply a *software* data structure in memory. Consequently, our scheme eliminates the costly hardware table required by current implementations of correlation prefetching [12, 18]. Moreover, accesses to the software table are inexpensive because the memory processor transparently caches the table in its cache. Finally, our new scheme enables the redesign of the correlation table and prefetching algorithms (Section 3.3) to address the low-coverage and short-distance prefetching limitations of current implementations.

3.2. Architecture of the System

Figures 3-(a) and (b) show the architecture of a system that integrates the memory processor in the North Bridge chip or in a DRAM chip, respectively. The first design requires no modification to the DRAM or its interface, and is largely compatible with conventional memory systems. The second design needs changes to the DRAM chips and their interface, and needs special support to work in typical memory systems, which have multiple DRAM chips. However, since our goal is to examine the performance potential of the two designs, we abstract away some of the implementation complexity of the second design by assuming a single-chip main memory. In the following, we outline how the systems work. In our discussion, we only consider memory accesses resulting from misses; we ignore write-backs for simplicity and because they do not affect our algorithms.

In Figure 3-(a), the key communication occurs through queues *1*, *2*, and *3*. Miss requests from the main processor are deposited in queues *1* and *2* simultaneously. The ULMT uses the entries in queue *2* to build its table and, based on it, generate the addresses to prefetch. The latter are deposited in queue *3*. Queues *1* and *3* compete to access memory, although queue *3* has a lower priority than *1*.

When the address of a line to prefetch is deposited in queue *3*, the hardware compares it against all the entries in queue *2*. If a match for address address *X* is detected, *X* is removed from both queues. We remove *X* from queue *3* because it is redundant: a higher-priority

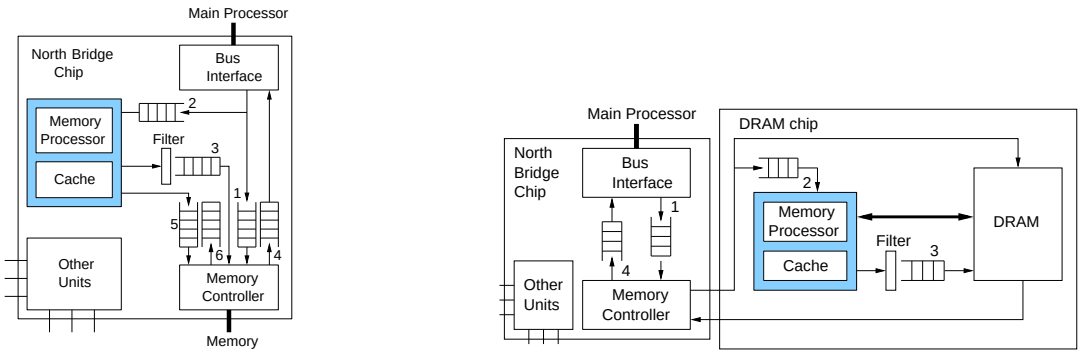


Figure 3. Architecture of a system that integrates the memory processor in the North Bridge chip (a) or in a DRAM chip (b).

request for *X* is already in queue *1*. *X* is removed from queue *2* to save computation in the ULMT. Note that it is unclear whether we lost the opportunity to prefetch *X*'s successors by not processing *X*. The reason is that our algorithms prefetch several levels of successor misses (Section 3.3) and, as a result, some of *X*'s successors may already be in queue *3*. Processing *X* may help improve the state in the correlation table. However, minimizing the total occupancy of the ULMT is crucial in our scheme.

Similarly, when a main-processor miss is about to be deposited in queues *1* and *2*, the hardware compares its address against those in queue *3*. If there is a match, the request is put only in queue *1* and the matching entry in queue *3* is removed.

It is possible that requests from the main processor arrive too fast for the ULMT to consume them and queue *2* overflows. In this case, the memory processor simply drops these requests.

Figure 3-(a) also shows the *Filter* module associated with queue *3*. This module improves the performance of correlation prefetching, which may sometimes try to prefetch the same address several times in a short time. The Filter module drops prefetch requests directed to any address that has been recently issued another prefetch requests. The module is a fixed-sized FIFO list that records the addresses of all the recently-issued requests. Before a request is issued to queue *3*, the hardware checks the Filter list. If it finds its address, the request is dropped and the list is left unmodified. Otherwise, the address is added to the tail of the list. With this support, some unnecessary prefetch requests are eliminated.

For completeness, the figure shows other queues. Replies from memory to the main processor go through queue *4*. In addition, the ULMT needs to access the software correlation table in main memory. Recall that the table is transparently cached by the memory processor. Logical queues *5* and *6* provide the necessary paths for the memory processor to access main memory. In practice, queues *5* and *6* are merged with the others.

If the memory processor is in the DRAM chip (Figure 3-(b)), the system works slightly differently. Miss requests from the main processor are deposited first in queue *1* and then in queue *2*. The ULMT in the memory processor accesses the correlation table from its cache and, on a miss, directly from the DRAM. The addresses to prefetch are passed through the Filter module and placed in queue *3*. As in Figure 3-(a), entries in queues *2* and *3* are checked against each other, and the common entries are dropped. The replies to both prefetches and main-processor requests are returned to the memory controller. As they reach the memory controller, their addresses are compared to the processor miss requests in queue *1*. If a memory-prefetched line matches a miss request from the main processor, the former is considered to be the reply of the latter, and the latter is not sent to the memory chip.

Finally, in machines that include a form of processor-side prefetching, we envision our architecture to operate in two modes: *Verbose* and *Non-Verbose*. In Verbose mode, queue *2* in Figures 3-(a) and (b) receives both main-processor misses and main-processor prefetch requests. In Non-Verbose mode, queue *2* only receives main-processor misses. This mode assumes that main-processor prefetch requests are distinguishable from other requests, for example with a tag as in the MIPS R10000 [21].

The Non-Verbose mode is useful to reduce the total occupancy of the ULMT. In this case, the processor-side prefetcher can focus on the easy-to-predict sequential or regular miss patterns, while the ULMT can focus on the hard-to-predict irregular ones. The Verbose mode is also useful: the ULMT can implement a prefetch algorithm that enhances the effectiveness of the processor-side prefetcher. We present an example of this case in Section 5.2.

3.3. Correlation Prefetching Algorithms

Simply taking the current pair-based correlation table and algorithm and implementing them in software is not good enough. Indeed, as indicated in Section 2.2, the *Base* algorithm has two limitations: it does not prefetch very far ahead and, intuitively, it needs to observe one miss to eliminate another miss (its immediate successor). As a result, it tends to have low coverage.

To increase coverage, three things need to occur. First, we need to eliminate these two limitations by storing in the table (and prefetching) *several levels* of successor misses per miss: immediate successors, successors of immediate successors, and so on for several levels. Second, these prefetches have to be highly accurate. Finally, the prefetcher has to take decisions early enough so that the prefetched lines reach the main processor before they are needed.

These conditions are easier to support and ensure when the correlation algorithm is implemented as a ULMT. There are two reasons for it. The first one is that storage is now cheap and, therefore, the correlation table can be inexpensively expanded to hold multiple levels of successor misses per miss, even if that means replicating information. The second reason is the *Customizability* provided by a software implementation of the prefetching algorithm.

In the rest of this section, we describe how a ULMT implementation of correlation prefetching can deliver high coverage. We describe three approaches: using a conventional table organization, using a table re-organized for ULMT, and exploiting customizability.

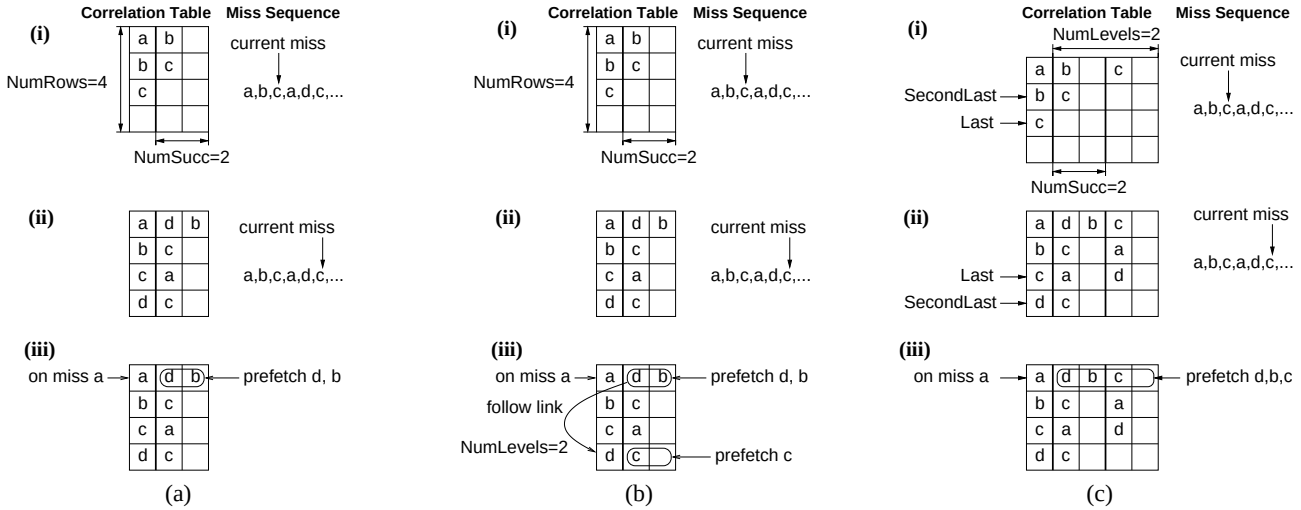


Figure 4. Pair-based correlation algorithms: *Base* (a), *Chain* (b), and *Replicated* (c).

3.3.1. Using a Conventional Table Organization

As a first step, we attempt to improve coverage without specifically exploiting the low-cost storage or customizability advantages of ULMT. We simply take the conventional table organization of Section 2.2 and force the ULMT to prefetch multiple levels of successors for every miss. The resulting algorithm we call *Chain*. *Chain* takes the same parameters as *Base* plus *NumLevels*, which is the number of levels of successors prefetched. The algorithm is illustrated in Figure 4-(b).

Chain updates the table like *Base* ((i) and (ii)) but prefetches differently ((iii)). Specifically, after prefetching the row of immediate successors, it takes the MRU one among them and accesses the correlation table again with its address. If the entry is found, it prefetches all *NumSucc* successors there. Then, it takes the MRU successor in that row and repeats the process. This is done *NumLevels*-1 times. As an example, suppose that a miss on *a* occurs ((iii)). The ULMT first prefetches *d* and *b*. Then, it takes the MRU entry *d*, looks-up the table, and prefetches *d*'s successor, *c*.

Chain addresses the two limitations of *Base*, namely not prefetching very far ahead, and needing one miss to eliminate a second one. However, *Chain* may not deliver high coverage for two reasons: the prefetches may not be highly accurate and the ULMT may have a high response time to issue all the prefetches.

The prefetches may be inaccurate because *Chain* does not prefetch the *true MRU* successors in each level of successors. Instead, it only prefetches successors found along the MRU path. For example, consider a sequence of misses that alternates between *a,b,c* and *b,e,b,f*: *a,b,c,...,b,e,b,f,...,a,b,c,...*. When miss *a* is encountered, *Chain* prefetches its immediate successors (*b*), and then accesses the entry for *b* to prefetch *e* and *f*. Note that *c* is not prefetched.

The high response time of *Chain* to a miss comes from having to make *NumLevels* accesses to different rows in the table. Each access involves an associative search because the table is associative and, potentially, one or more cache misses.

3.3.2. Using a Table Re-Organized for ULMT

We now attempt to improve coverage by exploiting the low cost of storage in ULMT solutions. Specifically, we expand the table to allow replicated information. Each row of the table stores the tag of the miss address, and *NumLevels* levels of successors. Each level contains *NumSucc* addresses that use LRU for replacement. Using this table, we propose an algorithm called *Replicated* (Figure 4-(c)). *Replicated* takes the same parameters as *Chain*.

As shown in Figure 4-(c), *Replicated* keeps *NumLevels* pointers to the table. These pointers point to the entries for the address of the last miss, second last, and so on, and are used for efficient table access. When a miss occurs, these pointers are used to access the entries of the last few misses, and insert the new address as the MRU successor of the correct level ((i) and (ii)). In the figure, the *NumSucc* entries at each level are MRU ordered. Finally, prefetching in *Replicated* is simple: when a miss is seen, all the entries in the corresponding row are prefetched ((iii)).

Note that *Replicated* eliminates the two problems of *Chain*. First, prefetches are accurate because they contain the *true MRU* successors at each level. This is the result of grouping together all the successors from a given level, irrespective of the path taken. In the sequence shown above *a,b,c,...,b,e,b,f,...,a,b,c,...*, on a miss on *a*, *Replicated* prefetches *b* and *c*.

Second, the response time of *Replicated* is much smaller than *Chain*. Indeed, *Replicated* prefetches several levels of successors with a single row access, and maybe even with a single cache miss. *Replicated* effectively shifts some computation from the Prefetching step to the Learning one: prefetching needs a single table access, while learning a miss needs multiple table updates. This is a good trade-off because the Prefetching step is the critical one. Furthermore, these multiple learning updates are inexpensive: the use of the pointers eliminates the need to do any associative searches on the table, and the rows to be updated are most likely still in the cache of the memory processor (since they were updated most recently).

3.3.3. Exploiting the Customizability of ULMT

We can also improve coverage by exploiting the second advantage of ULMT solutions: customizability. The programmer or system can choose to run a different algorithm in the ULMT for each application. The chosen algorithm can be highly customized to the application's needs.

One approach to customization is to use the table organizations and prefetching algorithms described above but to tune their parameters on an application basis. For example, in applications where the miss sequences are highly predictable, we can set the number of levels of successors to prefetch (*NumLevels*) to a high value. As a result,

Characteristics	*Base*	*Chain*	*Replicated*
Levels of successors prefetched	1	*NumLevels*	*NumLevels*
True MRU ordering for each level?	Yes	No	Yes
Number of row accesses in the Prefetching step (Requires SEARCH)	1	*NumLevels*	1
Number of row accesses in the Learning step (Requires 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 different pair-based correlation prefetching algorithms running on a ULMT.

we will prefetch more levels of successors with high accuracy. In applications with unpredictable sequences, we can do the opposite. We can also tune the number of rows in the table (*NumRows*). In applications that have large footprints, we can set *NumRows* to a high value to hold more information in the table. In small applications, we can do the opposite to save space.

A second approach to customization is to use a different prefetching algorithm. For example, we can add support for sequential prefetching to all the algorithms described above. The resulting algorithms will have low response time for sequential miss patterns.

Another approach is to adaptively decide the algorithm on-the-fly, as the application executes. In fact, this approach can also be used to execute different algorithms in different parts of one application. Such intra-application customizability may be useful in complex applications.

Finally, the ULMT can also be used for profiling purposes. It can monitor the misses of an application and infer higher-level information such as cache performance, application access patterns, or page conflicts.

3.3.4. Comparing the Algorithms

Table 1 compares the *Base*, *Chain*, and *Replicated* algorithms executing on a ULMT. *Replicated* has the highest potential for high coverage: it supports far-ahead prefetching by prefetching several levels of successors, its prefetches have high accuracy because they prefetch the true MRU successors at each level, and it has a low response time, in part because it only needs to access a single table row in the Prefetching step. Accessing a single row minimizes the associative searches and the cache misses. The only shortcoming of *Replicated* is the larger space that it requires for the correlation table. However, this is a minor issue since the table is a software structure allocated in main memory. Note that all these algorithms can also be implemented in hardware. However, *Replicated* is more suitable for an ULMT implementation because providing the larger space required in hardware is expensive.

3.4. Operating System Issues

There are some operating system issues that are related to ULMT operation. We outline them here.

Protection. The ULMT has its own separate address space with its instructions, the correlation table, and a few other data structures. The ULMT shares neither instructions nor data with any application. The ULMT can observe the physical addresses of the application misses. It can also issue prefetches for these addresses on behalf of the main processor. However, it can neither read from nor write to these addresses. Therefore, protection is guaranteed.

Multiprogrammed Environment. It is a poor approach to have all the applications share a single table: the table is likely to suffer a lot of interference. A better approach is to associate a different ULMT, with its own table, to each application. This eliminates interference in the tables. In addition, it enables the customization of each ULMT to its own application. If we conservatively assume a 4-Mbyte table on average per application, 8 applications require 32 Mbytes, which is only a modest fraction of today's typical main memory. If this requirement is excessive, we can save space by dynamically sizing the tables. In this case, if an application does not use the space, its table shrinks.

Scheduling. The scheduler knows the ULMT associated with each application. Consequently, the scheduler schedules and preempts both application and ULMT as a group. Furthermore, the operating system provides an interface for the application to control its ULMT.

Page Re-mapping. Sometimes, a page gets re-mapped. Since ULMTs operate on physical addresses, such events can cause some table entries to become stale. We can choose to take no action and let the table update itself automatically through learning. Alternatively, the operating system can inform the corresponding ULMT when a re-mapping occurs, passing the old and new physical page number. Then, the ULMT indexes its table for each line of the old page. If the entry is found, the ULMT relocates it and updates both the tag and any applicable successors in the row. Given current page sizes, we estimate the table update to take a few microseconds. Such overhead may be overlapped with the execution of the operating system page mapping handler in the main processor. Note that some other entries in the table may still keep stale successor information. Such information may cause a few useless prefetches, but the table will quickly update itself automatically.

4. Evaluation Environment

Applications. To evaluate the ULMT approach, we use nine mostly-irregular, memory-intensive applications. Irregular applications are hardly amenable to compiler-based prefetching. Consequently, they are the obvious target for ULMT correlation prefetching. The exception is CG, which is a regular application. Table 2 describes the applications. The last four columns of the table will be explained later.

Simulation Environment. The evaluation is done using an execution-driven simulation environment that supports a dynamic superscalar processor model [17]. We model a PC architecture with a simple memory processor that is integrated in either the North Bridge chip or in a DRAM chip, following the micro-architecture of Figure 3. Table 3 shows the parameters used for each component of the architecture. All cycles are 1.6 GHz cycles. The architecture is modeled cycle by cycle.

We model only a uni-programmed environment with a single application and a single ULMT that execute concurrently. We model all the contention in the system, including the contention of the application thread and the ULMT on shared resources such as the memory controller, DRAM channels, and DRAM banks.

Processor-Side Prefetching. The main processor optionally includes a hardware prefetcher that can prefetch multiple streams of stride 1 or -1 into the L1 cache. The prefetcher monitors L1 cache misses and can identify and prefetch up to *NumSeq* sequen-

Appl	Suite	Problem	Input	Correlation Table			
				NumRows (K)	Size (Mbytes)		
					Base	*Chain*	*Repl*
CG	NAS	Conjugate gradient	Class S	64	1.3	0.8	1.8
Equake	SpecFP2000	Seismic wave propagation simulation	*Test*	128	2.5	1.5	3.5
FT	NAS	3D Fourier transform	Class S	256	5.0	3.0	7.0
Gap	SpecInt2000	Group theory solver	Rako (subset of *test*)	128	2.5	1.5	3.5
Mcf	SpecInt2000	Combinatorial optimization	*Test*	32	0.6	0.4	0.9
MST	Olden	Finding minimum spanning tree	1024 nodes	256	5.0	3.0	7.0
Parser	SpecInt2000	Word processing	Subset of *train*	128	2.5	1.5	3.5
Sparse	SparseBench[10]	GMRES with compressed row storage	32^3	256	5.0	3.0	7.0
Tree	Univ. of Hawaii[3]	Barnes-Hut N-body problem	2048 bodies	8	0.2	0.1	0.2
Average	–	–	–	140	2.7	1.6	3.8

Table 2. Applications used.

PROCESSOR
Main Processor:
6-issue dynamic. 1.6 GHz. Int, fp, ld/st FUs: 4, 4, 2
Pending ld, st: 8, 16. Branch penalty: 12 cycles
Memory Processor:
2-issue dynamic. 800 MHz. Int, fp, ld/st FUs: 2, 0, 1
Pending ld, st: 4, 4. Branch penalty: 6 cycles
MEMORY
Main Processor's Memory Hierarchy:
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)
Memory bus: split-transaction, 8 B, 400 MHz, 3.2 GB/sec peak
Memory Processor's Memory Hierarchy:
L1 data: write-back, 32 KB, 2 way, 32-B line, 4-cycle hit RT
In North Bridge: RT mem latency: 100 cycles (row miss), 65 cycles (row hit)
Latency of a prefetch request to reach DRAM: 25 cycles
In DRAM: RT mem latency: 56 cycles (row miss), 21 cycles (row hit)
Internal DRAM data bus: 32-B wide, 800 MHz, 25.6 GB/sec peak
DRAM Parameters (applicable to all procs):
Dual channel. Each channel: 2 B, 800 MHz. Total: 3.2 GB/sec peak
Random access time (*tRAC*): 45 ns
Time from memory controller (*tSystem*): 60 ns
OTHER
Depth of queues *1* through *6*: 16
Filter module: 32 entries, FIFO

Table 3. 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.

tial streams concurrently. It works as follows. When the third miss in a sequence is observed, the prefetcher recognizes a stream. Then, it prefetches the next *NumPref* lines in the stream into the L1 cache. Furthermore, it stores the stride and the next address expected in the stream in a special register. If the processor later misses on the address in the register, the prefetcher prefetches the next *NumPref* lines in the stream and updates the register. The prefetcher contains *NumSeq* such registers. As we can see, while this scheme works somewhat like stream buffers [13], the prefetched lines go to L1. We choose this approach to minimize hardware complexity. A shortcoming is that the L1 cache may get polluted. For completeness, we resimulated the system with the prefetches going into separate buffers rather than into L1. We found that the performance changes very little, in part because checking the buffers on L1 misses introduces delay.

Algorithm Parameters. Table 4 lists the prefetching algorithms that we evaluate and the default parameters that we use. The sequential prefetching supported in hardware by the main processor is called *Conven4* for conventional. It can also be implemented in software by a ULMT. We evaluate two such software implementations (*Seq1* and *Seq4*). In this case, the prefetcher in memory observes L2 misses rather than L1.

Unless otherwise indicated, the processor-side prefetcher is off and, if it is on, the ULMT algorithms operate in Non-Verbose mode (Section 3.2). For the *Base* algorithm, we choose the parameter values used by Joseph and Grunwald [12] so that we can compare the work. The last four columns of Table 2 give a conservative value for the size of the correlation table for each application. The table is two-way set-associative. We have sized the number of rows in the table (*NumRows*) to be the lowest power of two such that, with a trivial hashing function that simply takes the lower bits of the line address, less than 5% of the insertions replace an existing entry. This is a very generous allocation. A more sophisticated hash function can reduce *NumRows* significantly without increasing conflicts much. In any case, knowing that each row in *Base*, *Chain*, and *Repl* takes 20, 12, and 28 bytes, respectively, in a 32-bit machine, we can compute the total table size. Overall, while some applications need more space than others, the average value is tolerable: 2.7, 1.6, and 3.8 Mbytes for *Base*, *Chain*, and *Repl*, respectively.

ULMT Implementation. We wrote all ULMTs in C and hand-optimized them for minimal response and occupancy time. One major performance bottleneck of the implementation is frequent branches. We remove branches by unrolling loops and hardwiring all algorithm parameters. We also perform optimizations to increase the spatial locality and to reduce instruction count. None of the algorithms uses floating-point operations.

5. Evaluation

5.1. Characterizing Application Behavior

Predictability of the Miss Sequences. We start by characterizing how well our ULMT algorithms can predict the miss sequences of the applications. For that, we run each ULMT algorithm simply observing all L2 cache miss addresses without performing prefetching. We record the fraction of L2 cache misses that are correctly predicted. For a sequential prefetcher, this means that the upcoming miss address matches the next address predicted by one of the streams identified; for a pair-based prefetcher, the upcoming address matches one of the successors predicted for that level.

Figure 5 shows the results of prediction for up to three levels of successors. Given a miss, the *Level 1* chart shows the predictability of the immediate successor, while *Level 2* shows the predictability of the next successor, and *Level 3* the successor after that one. The experiments for the pair-base schemes use large tables to ensure that practically no prediction is missed due to conflicts in the table: *NumRows* is 256 K, *Assoc* is 4, and *NumSucc* is 4. Under these condi-

Prefetching Algorithm	Implementation	Name	Parameter Values
Base	Software in memory as ULMT	*Base*	*NumSucc* = 4, *Assoc* = 4
Chain		*Chain*	*NumSucc* = 2, *Assoc* = 2, *NumLevels* = 3
Replicated		*Repl*	*NumSucc* = 2, *Assoc* = 2, *NumLevels* = 3
Sequential 1-Stream		*Seq1*	*NumSeq* = 1, *NumPref* = 6
Sequential 4-Streams		*Seq4*	*NumSeq* = 4, *NumPref* = 6
Sequential 4-Streams	Hardware in L1 of main processor	*Conven4*	*NumSeq* = 4, *NumPref* = 6

Table 4. Parameter values used for the different algorithms.

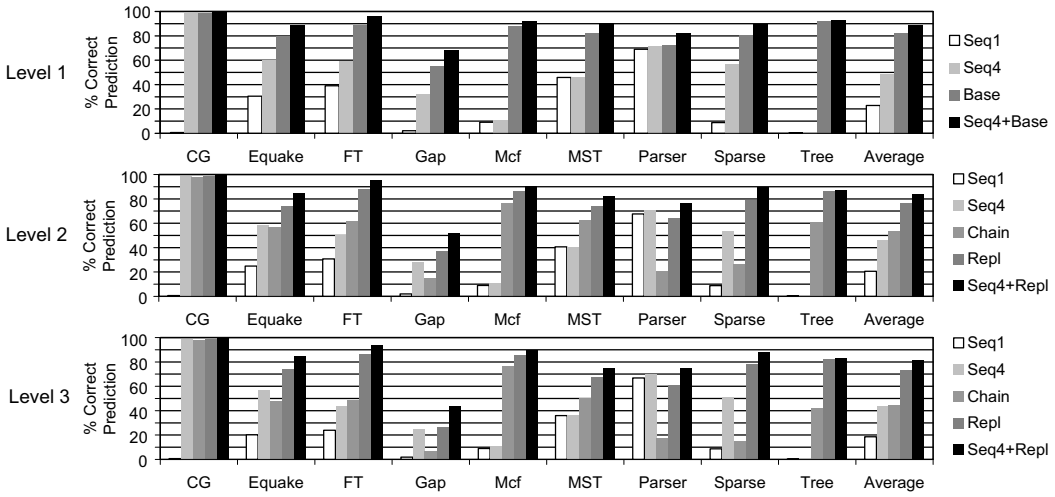


Figure 5. Fraction of L2 cache misses that are correctly predicted by different algorithms for different levels of successors.

tions, for level 1, *Chain* and *Repl* are equivalent to *Base*. For levels 2 and 3, *Base* is not applicable. The figure also shows the effect of combining algorithms.

Figure 5 shows that our ULMT algorithms can effectively predict the miss streams of the applications. For example, at level 1, *Seq4* and *Base* correctly predict on average 49% and 82% of the misses, respectively. Moreover, the best algorithms keep predicting correctly across several levels of successors. For example, *Repl* correctly predicts on average 77% and 73% of the misses for levels 2 and 3, respectively. Therefore, these algorithms have good potential.

The figure also shows that different applications have different miss behavior. For instance, applications such as Mcf and Tree do not have sequential patterns and, therefore, only pair-based algorithms can predict misses. In other applications such as CG, instead, sequential patterns dominate. As a result, sequential prefetching can predict practically all L2 misses. Most applications have a mix of both patterns.

Among pair-based algorithms, *Repl* almost always outperforms *Chain* by a wide margin. This is because *Chain* does not maintain the true MRU successors at each level. However, while *Repl* is effective under all patterns, it is better when combined with multi-stream sequential prefetching (*Seq4+Repl*).

Time Between L2 Misses. Another important issue is the time between L2 misses. Figure 6 classifies L2 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) cycles, [80,200) cycles, etc. The unit is 1.6 GHz processor cycles.

The most significant bin is [200,280), which contributes with 60% of all miss distances on average. These misses are critical beyond their numbers because their latencies are hard to hide with out-of-order execution. Indeed, since the round-trip latency to memory is 208-243 cycles, dependent misses are likely to fall in this bin. They contribute more to processor stall than the figure suggests because dependent misses cannot be overlapped with each other. Consequently, we want the ULMT to prefetch them. To make sure that the ULMT is fast enough to learn these misses, its occupancy should be less than 200 cycles.

The misses in the other bins are fewer and less critical. Those in [280,∞) are too far apart to put pressure on the ULMT's timing. Those in [0,80) may not give enough time to the ULMT to respond. Fortunately, these misses are more likely to be overlapped with each other and with computation.

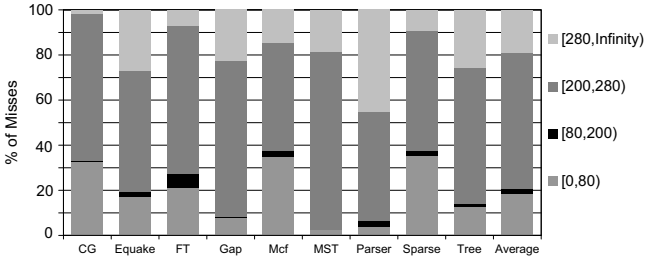


Figure 6. Characterizing the time between L2 misses.

5.2. Comparing the Different Algorithms

Figure 7 compares the execution time of the applications under different cases: no prefetching (*NoPref*), processor-side prefetching as listed in Table 4 (*Conven4*), different ULMT schemes listed in Table 4 (*Base*, *Chain*, and *Repl*), the combination of *Conven4* and *Repl* (*Conven4+Repl*), and some customized algorithms (*Custom*). The results are for the case where the memory processor is integrated in the DRAM. For each application and the average, the bars are normalized to *NoPref*. The bars show the memory-induced processor stall time that is caused by requests between the processor and the L2 cache (*UptoL2*), and by requests beyond the L2 cache (*Be-*

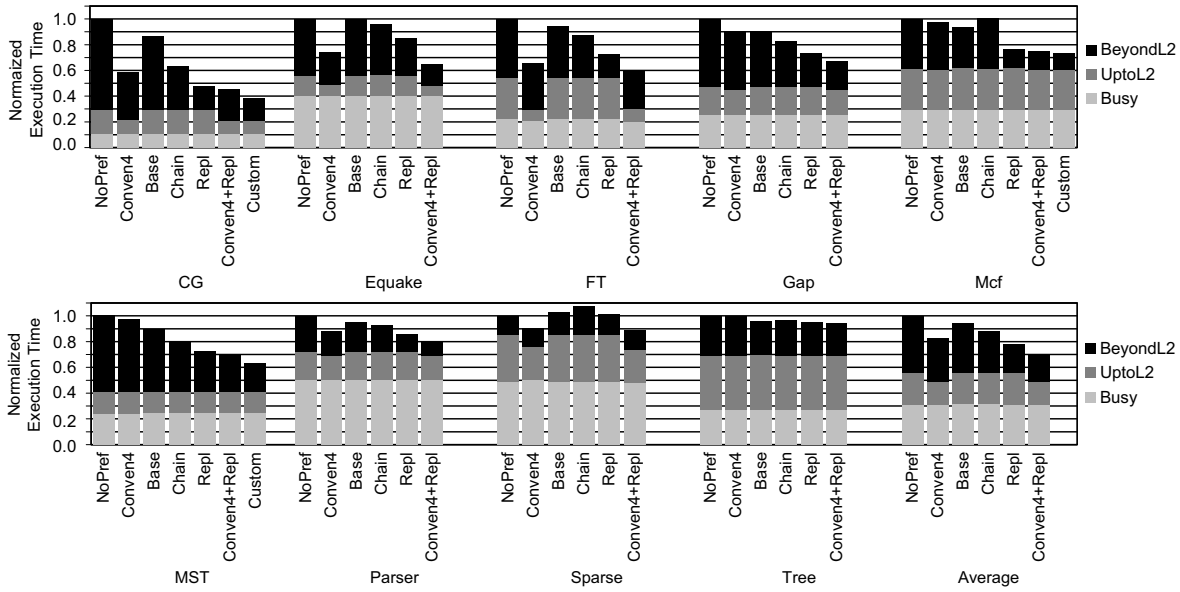


Figure 7. Execution time of the applications with different prefetching algorithms.

yondL2). The remaining time (*Busy*) includes processor computation plus other pipeline stalls. A system with a perfect L2 cache would only have the *Busy* and *UptoL2* times.

On average, *BeyondL2* is the most significant component of the execution time under *NoPref*. It accounts for 44% of the time. Thus, although our ULMT schemes only target L2 cache misses, they target the main contributor to the execution time.

Conven4 performs very well on CG because sequential patterns dominate. However, it is ineffective in applications such as Mcf and Tree that have purely irregular patterns. On average, *Conven4* reduces the execution time by 17%.

The pair-based schemes show mixed performance. *Base* shows limited speedups, mostly because it does not prefetch far enough. On average, it reduces *NoPref*'s execution time by 6%. *Chain* performs a little better, but it is limited by inaccuracy (Figure 5) and high response time (Section 3.3.1). On average, it reduces *NoPref*'s execution time by 12%.

Repl is able to reduce the execution time significantly. It performs well in almost all applications. It outperforms both *Base* and *Chain* in all cases. Its impact comes from the nice properties of the *Replicated* algorithm, as discussed in Section 3.3.4. The average of the application speedups of *Repl* over *NoPref* is 1.32.

Finally, *Conven4+Repl* performs the best. On average, it removes over half of the *BeyondL2* stall time, and delivers an average application speedup of 1.46 over *NoPref*. If we compare the impact of processor-side prefetching only (*Conven4*) and memory-side prefetching only (*Repl*), we see that they have a constructive effect in *Conven4+Repl*. The reason is that the two schemes help each other. Specifically, the processor-side prefetcher prefetches and eliminates the sequential misses. The memory-side prefetcher works in Non-Verbose mode (Section 3.2) and, therefore, does not see the prefetch requests. Therefore, it can fully focus on the irregular miss patterns. With the resulting reduced load, the ULMT is more effective.

Algorithm Customization. In this first paper on ULMT prefetching, we have attempted only very simple customization for a few applications. Table 5 shows the changes. For CG, we run *Seq1+Repl* in Verbose mode. For MST and Mcf, we run *Repl* with a higher *NumLevels*. In all cases, *Conven4* is on. The results are shown in Figure 7 as the *Custom* bar in the three applications.

Application	Customized ULMT Algorithm
CG	*Seq1+Repl* in Verbose mode
MST, Mcf	*Repl* with *NumLevels* = 4

Table 5. Customizations performed. *Conven4* is also on.

The customization in CG tries to further exploit positive interaction between processor- and memory-side prefetching. While CG only has sequential miss patterns (Figure 5), its multiple streams overwhelm the conventional prefetcher. Indeed, although processor-side prefetches are very accurate (99.8% of the prefetched lines are referenced), they are not timely enough (only 64% are timely) because some of them miss in the L2 cache. In our customization, we turn on the Verbose mode so that processor-side prefetch requests are seen by the ULMT. Furthermore, the ULMT is extended with a single-stream sequential prefetch algorithm (*Seq1*) before executing *Repl*. In this environment, the positive interaction between the two prefetchers increases. Specifically, while the application references the different streams in an interleaved manner, the processor-side prefetcher "unscrambles" the miss sequence into chunks of same-stream prefetch requests. The *Seq1* prefetcher in the ULMT then easily identifies each stream and, very efficiently, prefetches ahead. As a result, 81% of the processor-side prefetches arrive in a timely manner. With this customization, the speedup of CG improves from 2.19 (with *Conven4+Repl*) to 2.59. This case demonstrates that even regular applications that are amenable to sequential processor-side prefetching can benefit from ULMT prefetching.

The customization in MST and Mcf tries to exploit predictability beyond the third level of successor misses by setting *NumLevels* to 4 in *Repl*. As shown in Figure 7, this approach is successful for MST, but it produces marginal gains in Mcf.

Overall, this initial attempt at customization shows promising results. After applying customization on three applications, the average execution speedup of the nine applications relative to *NoPref* becomes 1.53.

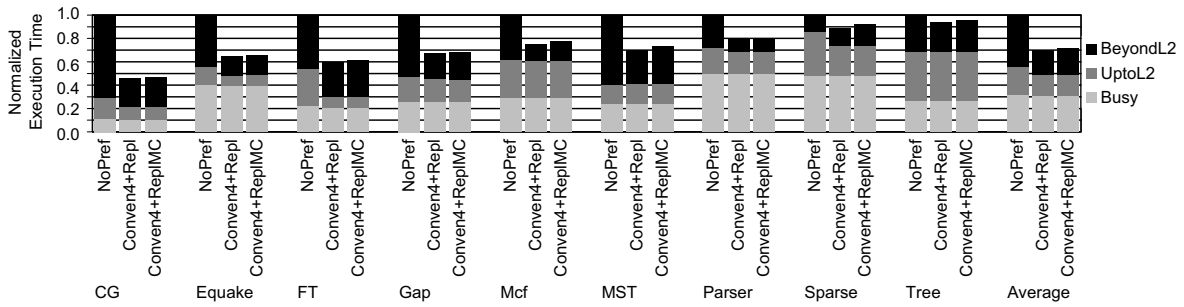


Figure 8. Execution time for different locations of the memory processor.

Location of Memory Processor. Figure 8 examines the impact of where we place the memory processor (Figure 3). The first two bars for each application are taken from Figure 7: *NoPref* and *Conven4+Repl*. The last bar for each application corresponds to the *Conven4+Repl* algorithm with the memory processor placed in the memory controller (North Bridge) chip (*Conven4+ReplMC*). With the processor in the North Bridge chip, we have twice the memory access latency (100 cycles vs. 56 cycles), eight times lower memory bandwidth (3.2 GB/sec vs. 25.6 GB/sec), and an additional 25-cycle delay seen by the prefetch requests before they reach the DRAM[1]. However, Figure 8 shows that the impact on the execution time is very small. It results in a small decrease in average speedups from 1.46 to 1.41. The impact is small thanks to the ability of *Repl* to accurately prefetch far ahead. Only the timeliness of the immediate successor prefetches is affected, while the prefetching of further levels of successors is still timely. Overall, given these results and the hardware cost of the two designs, we conclude that putting the memory processor in the North Bridge chip is the most cost-effective design of the two.

Prefetching Effectiveness. To gain further insight into these prefetching schemes, Figure 9 examines the effectiveness of the lines prefetched into the L2 cache by the ULMT. These lines are called *prefetches*. The figure shows data for Sparse, Tree, and the average of the other seven applications. The figure combines both L2 misses and prefetches, and breaks them down into 5 categories: prefetches that eliminate an L2 miss (*Hits*), prefetches that eliminate part of the latency of an L2 miss because they arrive a bit late (*DelayedHits*), L2 misses that pay the full latency (*NonPrefMisses*), and useless prefetches. Useless prefetches are further broken down into prefetches that are brought into the L2 but that are not referenced by the time they are replaced (*Replaced*), and prefetches that are dropped on arrival to L2 because the same line is already in the cache (*Redundant*). Since *Coverage* is the fraction of the original L2 misses that are fully or partially eliminated, it is represented by the sum of *Hits* and *DelayedHits* as shown in Figure 9. *NonPrefMisses* in Figure 9 is the number of L2 misses left after prefetching, relative to the original number of L2 misses. Note that *NonPrefMisses* can be higher than 1.0 for some algorithms. $1.0 - NonPrefMisses$ is the number of L2 misses eliminated relative to the original number of L2 misses. *NonPrefMisses* can be broken down into two groups: those misses below the 1.0 line in Figure 9 ($1.0 - Hits - DelayedHits$) come from the original misses, while those above the 1.0 line ($Hits + DelayedHits + NonPrefMisses - 1.0$) are the new L2 conflict misses caused by prefetches.

Looking at the average of the seven applications, we see why *Base* and *Chain* are not effective: their coverage is small. *Base* is hurt by its inability to prefetch far ahead, while *Chain* is hampered by its high response time and limited accuracy. The figure also shows that *Repl* has a high coverage (0.74). However, this comes at the cost of useless prefetches (*Replaced* plus *Redundant* are equivalent to 50% of the original misses) and additional misses due to conflicts with prefetches (20% of the original misses). We can see, therefore, that advanced pair-based schemes need additional bandwidth.

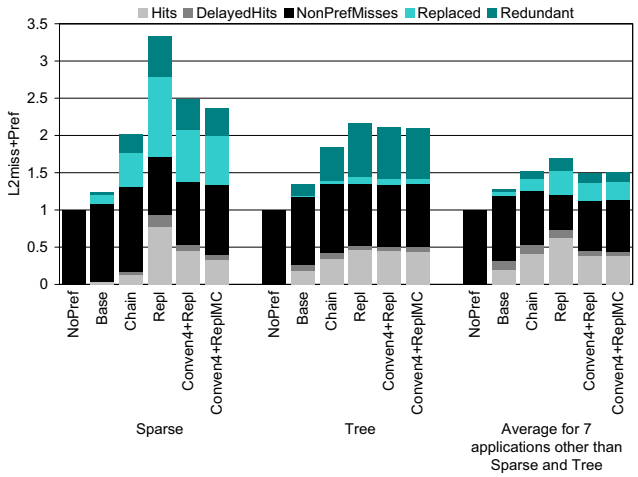


Figure 9. Breakdown of the L2 misses and lines prefetched by the ULMT (prefetches). The original misses are normalized to 1.

Conven4+Repl seems to have low coverage, despite its high performance in Figure 7. The reason is that the prefetch requests issued by the processor-side prefetcher, while effective in eliminating L2 misses, are lumped into the *NonPrefMisses* category in the figure if they reach memory. Since the ULMT prefetcher is in Non-Verbose mode, it does not see these requests. Consequently, the ULMT prefetcher only focuses on the irregular miss patterns. ULMT prefetches that eliminate irregular misses appear as *Hits+DelayedHits*.

Finally, Figure 9 also shows why *Sparse* and *Tree* showed limited speedups in Figure 7. They have too many conflicts in the cache, which results in many remaining *NonPrefMisses*. Furthermore, their prefetches are not very accurate, which results in large *Replaced* and *Redundant* categories.

Work Load of the ULMT. Figure 10 shows the average response time and occupancy time (Section 3.1) for each of the ULMT algorithms, averaged over all applications. The times are measured in 1.6 GHz cycles. 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 ULMT. The IPC is calculated as the number

[1] All these cycle counts are in main-processor cycles.

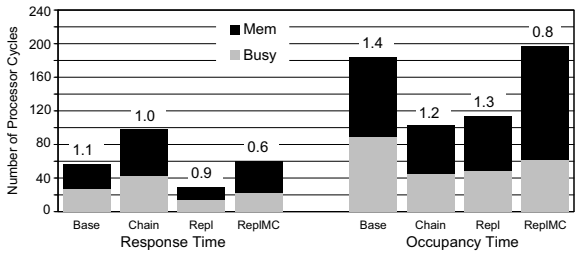


Figure 10. Average response and occupancy time of different ULMT algorithms in main-processor cycles.

of instructions divided by the number of *memory processor* cycles.

The figure shows that, in all the algorithms, the occupancy time is less than 200 cycles. Consequently, the ULMT is fast enough to process most of the L2 misses (Figure 6). Memory stall time is roughly half of the ULMT execution time when the processor is in the DRAM, and more when the processor is in the North Bridge chip (*ReplMC*). *Chain* and *Repl* have the lowest occupancy time. Note that *Repl*'s occupancy is not much higher than *Chain*'s, despite the higher number of table updates performed by *Repl*. The reasons are the fewer associative searches and the better cache line reuse in *Repl*.

The response time is most important for prefetching effectiveness. The figure shows that *Repl* has the lowest response time, at around 30 cycles. The response time of *ReplMC* is about twice as much. Fortunately, the *Replicated* algorithm is able to prefetch far ahead accurately and, therefore, the effectiveness of prefetching is not very sensitive to a modest increase in the response time.

Main Memory Bus Utilization. Finally, Figure 11 shows the utilization of the main memory bus for various algorithms, averaged over all applications. The increase in bus utilization induced by the advanced algorithms is divided into two parts: increase caused naturally by the reduced execution time, and additional increase caused by the prefetching traffic. Overall, the figure shows that the increase in bus utilization is tolerable. The utilization increases from the original 20% to only 36% in the worst case (*Conven4+Repl*). Moreover, most of the increase comes from the faster execution; only a 6% utilization is directly attributable to the prefetches. In general, the fact that memory-side prefetching only adds one-way traffic to the main memory bus, limits its bandwidth needs.

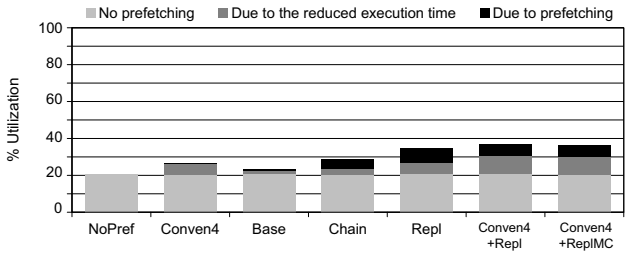


Figure 11. Main memory bus utilization.

6. Related Work

Memory-Side Prefetching. Some memory-side prefetchers are simple hardware controllers. For example, the NVIDIA chipset includes the DASP controller in the North Bridge chip [22]. It seems that it is mostly targeted to stride recognition and buffers data locally. The i860 chipset from Intel is reported to have a prefetch cache, which may indicate the presence of a similar engine. Cooksey *et al.* [9] propose the Content-Based prefetcher, which is a hardware controller that monitors the data coming from memory. If an item appears to be an address, the engine prefetches it. Alexander and Kedem [1] propose a hardware controller that monitors requests at the main memory. If it observes repeatable patterns, it prefetches rows of data from the DRAM to an SRAM buffer inside the memory chip. Overall, our scheme is different in that we use a general-purpose processor running a prefetching algorithm as a user-level thread.

Other studies propose specialized programmable engines. For example, Hughes [11] and Yang and Lebeck [28] propose adding a specialized engine to prefetch linked data structures. While Hughes focuses on a multiprocessor processing-in-memory system, Yang and Lebeck focus on a uniprocessor and put the engine at every level of the cache hierarchy. The main processor downloads information on these engines about the linked structures and what prefetches to perform. Our scheme is different in that it has general applicability.

Another related system is Impulse, an intelligent memory controller capable of remapping physical addresses to improve the performance of irregular applications [4]. Impulse could prefetch data, but only implements next-line prefetching. Furthermore, it buffers data in the memory controller, rather than sending it to the processor.

Correlation Prefetching. Early work on correlation prefetching can be found in [2, 24]. More recently, several authors have made further contributions. Charney and Reeves study correlation prefetching and suggest combining a stride prefetcher with a general correlation prefetcher [6]. Joseph and Grunwald propose the basic correlation table organization and algorithm that we evaluate [12]. Alexander and Kedem use correlation prefetching slightly differently [1], as we indicate above. Sherwood *et al.* use it to help stream buffers prefetch irregular patterns [26]. Finally, Lai *et al.* design a slightly different correlation prefetcher [18]. Specifically, a prefetch is not triggered by a miss; instead, it is triggered by a dead-line predictor indicating that a line in the cache will not be used again and, therefore, a new line should be prefetched in. This scheme improves prefetching timeliness at the expense of tighter integration of the prefetcher with the processor, since the prefetcher needs to observe not only miss addresses, but also reference addresses and program counters.

We differ from the recent works in important ways. First, they propose hardware-only engines, which often require expensive hardware tables; we use a flexible user-level thread on a general-purpose core that stores the table as a software structure in memory. Second, except for Alexander and Kedem [1], they place their engines between the L1 and L2 caches, or between the processor and the L1; we place the prefetcher in memory and focus on L2 misses. Time intervals between L2 misses are large enough for a ULMT to be viable and effective. Finally, we propose a new table organization and prefetching algorithm that, by exploiting inexpensive memory space, increases far-ahead prefetching and prefetch coverage.

Prefetching Regular Structures. Several schemes have been proposed to prefetch sequential or strided patterns. They include the Reference Prediction table of Chen and Baer [7], and the Stream buffers of Jouppi [13], Palacharla and Kessler [23], and Sherwood *et al.* [26]. We base our processor-side prefetcher on these schemes.

Processor-Side Prefetching. There are many more proposals for processor-side prefetching, often for irregular applications. A tiny, non-exhaustive list includes Choi *et al.* [8], Karlsson *et al.* [14], Lipasti *et al.* [19], Luk and Mowry [20], Roth *et al.* [25], and Zhang and Torrellas [29]. Most of these schemes specifically target linked data structures. They tend to rely on program information that is available to the processor, like the addresses and sizes of data struc-

tures. Often, they need compiler support. Our scheme needs neither program information nor compiler support.

Other Related Work. Chappell *et al*. [5] use a subordinate thread in a multithreaded processor to improve branch prediction. They suggest using such a thread for prefetching and cache management. Finally, our work is also related to data forwarding in multiprocessors, where a processor pushes data into the cache hierarchy of another processor [15].

7. Conclusions

This paper introduced memory-side correlation prefetching using a User-Level Memory Thread (ULMT) running on a simple general-purpose processor in main memory. This scheme solves many of the problems in conventional correlation prefetching and provides several important additional features. Specifically, the scheme needs minimal hardware modifications beyond the memory processor, uses main memory to store the correlation table inexpensively, can exploit a new table organization to increase far-ahead prefetching and coverage, can effectively prefetch for applications with largely any miss pattern as long as it repeats, and supports customization of the prefetching algorithm by the programmer for individual applications. Our results showed that the scheme delivers an average speedup of 1.32 for nine mostly-irregular applications. Furthermore, our scheme works well in combination with a conventional processor-side sequential prefetcher, in which case the average speedup increases to 1.46. Finally, by exploiting the customization of the prefetching algorithm, we increased the average speedup to 1.53.

This work is being extended by designing effective techniques for ULMT customization. In particular, we are customizing for linked data structure prefetching, cache conflict detection and elimination, and general application profiling. Customization for cache conflict elimination should improve Sparse and Tree, the applications with the smallest speedups.

Acknowledgments

The authors thank the anonymous reviewers, Hidetaka Magoshi, Jose Martinez, Milos Prvulovic, Marc Snir, and James Tuck.

References

[1] T. Alexander and G. Kedem. Distributed Predictive Cache Design for High Performance Memory Systems. In *the Second International Symposium on High-Performance Computer Architecture*, pages 254–263, February 1996.

[2] J. L. Baer. Dynamic Improvements of Locality in Virtual Memory Systems. *IEEE Transactions on Software Engineering*, 2:54–62, March 1976.

[3] J. E. Barnes. Treecode. Institute for Astronomy, University of Hawaii. 1994. *ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode*.

[4] J. B. Carter et al. Impulse: Building a Smarter Memory Controller. In *the 5th International Symposium on High-Performance Computer Architecture*, pages 70–79, January 1999.

[5] R. S. Chappell, J. Stark, S. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *the 26th International Symposium on Computer Architecture*, pages 186–195, May 1999.

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

[7] T. F. Chen and J. L.Baer. Reducing Memory Latency via Non-Blocking and Prefetching Cache. In *the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.

[8] S. Choi, D. Kim, and D. Yeung. Multi-Chain Prefetching: Effective Exploitation of Inter-Chain Memory Parallelism for Pointer-Chasing Codes. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 51–61, September 2001.

[9] R. Cooksey, D. Colarelli, and D. Grunwald. Content-Based Prefetching: Initial Results. In *the 2nd Workshop on Intelligent Memory Systems*, pages 33–55, November 2000.

[10] J. Dongarra, V. Eijkhout, and H. van der Vorst. SparseBench: A Sparse Iterative Benchmark. *http://www.netlib.org/benchmark/sparsebench*.

[11] C. J. Hughes. Prefetching Linked Data Structures in Systems with Merged DRAM-Logic. Master's thesis, University of Illinois at Urbana-Champaign, May 2000. Technical Report UIUCDCS-R-2001-2221.

[12] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *the 24th International Symposium on Computer Architecture*, pages 252–263, June 1997.

[13] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

[14] M. Karlsson, F. Dahlgren, and P. Stenstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *the 6th International Symposium on High-Performance Computer Architecture*, pages 206–217, January 2000.

[15] D. Koufaty and J. Torrellas. Comparing Data Forwarding and Prefetching for Communication-Induced Misses in Shared-Memory MPs. In *International Conference on Supercomputing*, pages 53–60, July 1998.

[16] C. Kozyrakis et al. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, pages 75–78, September 1997.

[17] V. Krishnan and J. Torrellas. A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, October 1998.

[18] A. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *the 28th International Symposium on Computer Architecture*, pages 144–154, June 2001.

[19] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid: Software Prefetching in Pointer and Call Intensive Environments. In *the 28th International Symposium on Microarchitecture*, pages 231–236, November 1995.

[20] C. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[21] MIPS. MIPS R10000 Microprocessor User's Manual. *Version 2.0*, January 1997.

[22] NVIDIA. Technical Brief: NVIDIA nForce Integrated Graphics Processor (IGP) and Dynamic Adaptive Speculative Pre-Processor (DASP). *http://www.nvidia.com/*.

[23] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *the 21st International Symposium on Computer Architecture*, pages 24–33, April 1994.

[24] J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio. Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks. *U.S. Patent 4,807,110*, February 1989.

[25] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.

[26] T. Sherwood, S. Sair, and B. Calder. Predictor-Directed Stream Buffers. In *the 33rd International Symposium on Microarchitecture*, pages 42–53, December 2000.

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

[28] C.-L. Yang and A. R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *International Conference on Supercomputing*, pages 176–186, May 2000.

[29] Z. Zhang and J. Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *the 22nd International Symposium on Computer Architecture*, pages 188–199, June 1995.