Correlation Prefetching with a User-Level Memory Thread

Yan Solihin, *Member*, *IEEE*, Jaejin Lee, *Member*, *IEEE*, and Josep Torrellas, *Senior Member*, *IEEE*

Abstract—This paper proposes 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 applicability, 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.

Index Terms—Prefetching, correlation prefetching, memory-side prefetching, helper threads, intelligent memory architecture, processing-in-memory, heterogeneous system.

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], [15], [16], [17], [18], [23], [24], [28], [30], [32], [35], [36]. 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], [14], [27], [35].

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 [34]. Some chipsets like NVIDIA's nForce even integrate a powerful processor in the North Bridge chip [27]. 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 [19].

Unfortunately, existing proposals for memory-side prefetching engines have a narrow scope [1], [9], [14], [27], [35]. Indeed, some designs are hardware controllers that perform simple and specific operations [1], [9], [27]. Other designs are specialized engines that are custom-designed to prefetch linked data structures [14], [35]. 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], [15], [21], [32]. 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], [15], [21]. In all cases but one, these controllers are placed between the L1 and L2 caches, or between the processor and the L1. While effective in some cases, this approach has a high hardware cost. Furthermore, it is often unable to prefetch far ahead enough.

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

- *Y. Solihin is with the Department of Electrical and Computer Engineering, North Carolina University, Campus Box 7911, Raleigh, NC 27695-7911. E-mail: solihin@ncsu.edu.*
- *J. Lee is with the School of Computer Science and Engineering, Seoul National University, Seoul 151-742, Korea. E-mail: jlee@cse.snu.ac.kr.*
- *J. Torrellas is with the Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, IL 61801. E-mail: torrellas@cs.uiuc.edu.*

Manuscript received 12 July 2002; revised 3 Feb. 2003; accepted 18 Feb. 2003. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 116940.

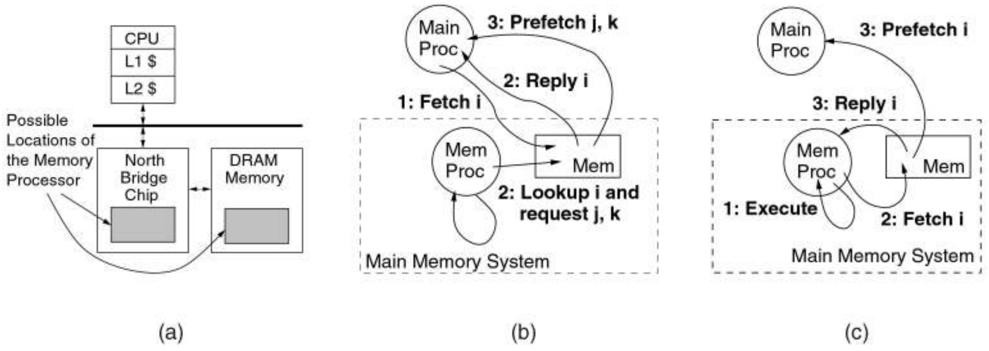


Fig. 1. Memory-side prefetching: some locations where the memory processor can be placed (a), and actions under push passive (b) and push active (c) prefetching.

wide applicability, 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 correlation 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

To provide background, this section describes memory-side and correlation prefetching. In this paper, we use the terms prefetching coverage and prefetching accuracy as follows: *Coverage* is the fraction of original misses that are eliminated (partially or totally) by the prefetched lines. *Accuracy* is the fraction of the prefetched lines that eliminate (partially or totally) original misses.

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], [14], [27], [35]. 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 [27] 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, Fig. 1a 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 chip for graphics processing [27], 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 [35], 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-side prefetcher can act as a pull prefetcher by simply buffering the prefetched data locally and supplying it to the processor on demand [1], [27]. 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 as *Passive* or *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 (Fig. 1b).

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 the memory processor is also sent to the main processor (Fig. 1c).

In this paper, we concentrate on push passive 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 support. First, as in other systems [14], [18], [35], the L2 cache has to accept a line from the memory that it has not requested. This line comes in a message that includes its address, as provided by the memory. When the line arrives at the L2 cache, the L2 cache can use a free Miss Status Handling Register (MSHR). Second, if the L2 has a pending request and a prefetched line with the same address arrives, the prefetched line simply steals the MSHR and updates the cache as if it were the reply. Finally, a 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 line wants to go to 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], [15], [21], [32]. 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], [15], [21], [32], 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], [15], [32], 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 [15], for best performance, the entries in a row should replace each other with a LRU policy.

Figs. 4a, 4b, and 4c illustrate how the algorithm works. We call the algorithm *Base*. Figs. 4a and 4b show two snapshots of the table at different points in the miss stream. Each table row keeps a miss and two of its immediate successor misses. Such 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 as in Fig. 4c, 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], [21] work slightly differently. They are discussed in Section 6.

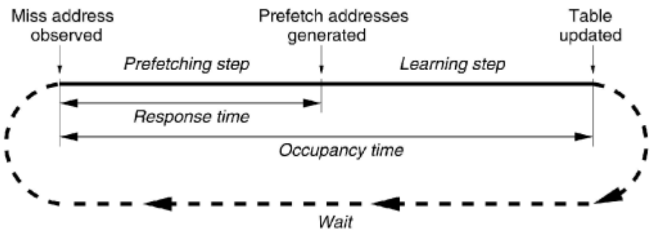


Fig. 2. Infinite loop executed by the ULMT.

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 [15], [21], while some applications with large footprints even need a 7.6 Mbyte off-chip SRAM table [21].

Furthermore, the popular schemes that prefetch several potential *immediate* successors for each miss [6], [15], [32] 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.

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 [15], 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 Fig. 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.

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 move as much housekeeping computation as possible from the Prefetching

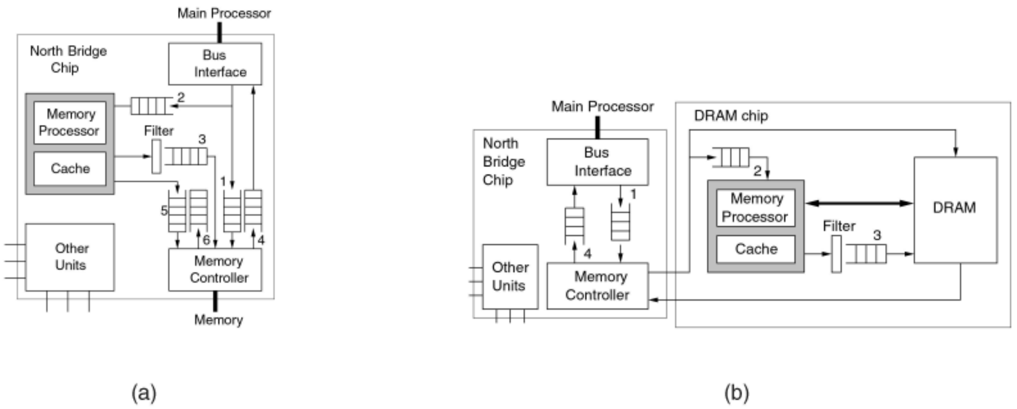


Fig. 3. (a) Architecture of a system that integrates the memory processor in the North Bridge chip or in (b) a DRAM chip.

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 time.

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 [15], [21]. 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

Figs. 3a and 3b 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 Fig. 3a, the key communication occurs through queues *1*, *2*, and *3*. Miss requests from the main processor are

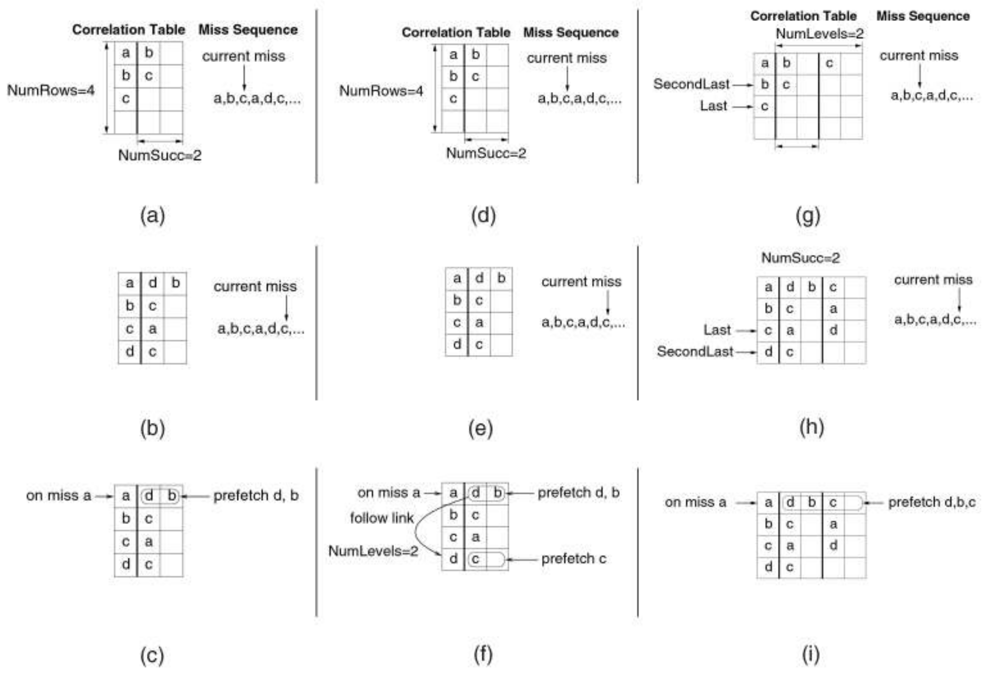


Fig. 4. Pair-based correlation algorithms: *Base* ((a), (b), and (c)), *Chain* ((d), (e), and (f)), and *Replicated* ((g), (h), and (i)).

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* try to access memory, although queue *3*'s messages have a lower priority: they wait until queue *1* is empty.

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 *X* is detected, *X* is removed from both queues. We remove *X* from queue *3* because it is redundant: a higher-priority 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.

Fig. 3a 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 prefetched by the memory processor. The module is a fixed-sized FIFO list that records the addresses of all the prefetches recently-issued by the memory processor. Before a prefetch 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 (Fig. 3b), 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 Fig. 3a, 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 *NonVerbose*. In Verbose mode, queue *2* in Figs. 3a and 3b receives both main-processor misses and main-processor prefetch requests. In NonVerbose 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 [26].

The NonVerbose 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 one other miss (its immediate successor). As a result, it tends to have low coverage.

Note that modifying the *Base* algorithm so that each row in the table stores, for example, only the second level successor misses (successors of the immediate succesors) would not help much. Indeed, while we would prefetch farther ahead, the coverage would still be low because we would still need one miss to eliminate one other miss. Moreover, the accuracy of the prefetches would decrease.

To increase coverage, three things need to occur. First, we need to eliminate the "one-level of successors per miss" limitation 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 target the misses accurately. Finally, the prefetcher has to make 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 reorganized for ULMT, and exploiting customizability.

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 Figs. 4d, 4e, and 4f.

Chain updates the table like *Base* (4d and 4e), but prefetches differently (Fig. 4f). Specifically, after prefetching the row of immediate successors, it takes the MRU successor 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 (Fig. 4f). 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*. An example of the possible inaccuracy is discussed in Section 3.3.4.

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 misses in the cache of the memory processor.

3.3.2 Using a Table Reorganized 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* (Figs. 4g, 4h, and 4i). *Replicated* takes the same parameters as *Chain*.

As shown in Fig. 4g, *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 (Figs. 4g and 4h). 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 (Fig. 4i).

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. An example of the increased accuracy is discussed in Section 3.3.4.

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 miss in the cache of the memory processor. *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, 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 intraapplication 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

We claimed in Sections 3.3.2 and 3.3.1 that, thanks to prefetching the true MRU successors at each level, *Replicated* is more accurate than *Chain*. Recall that the latter only prefetches successors *found along the MRU path*. We now show the higher accuracy of *Replicated* with an example: a repeating miss sequence that is interrupted by a one-time miss sequence that shares one of the misses with the repeating sequence.

Consider the miss sequence a, b, c, d, which repeats, but is interrupted by a one-time miss sequence b, e, f before resuming again: $a, b, c, d, \ \ldots, \ a, b, c, d, \ \ldots, a, b, c, d, \ \ldots,$ $b, e, f, \ldots, a$. Since a, b, c, d and b, e, f are distinct patterns, we do not want them to confuse the prefetcher. Thus, when the a miss after b, e, f is encountered, we want to prefetch b, c, d. However, *Chain* and *Replicated* behave differently in this case.

To see why, Figs. 5a, 5b, and 5c show the behavior of the *Chain* algorithm, and Figs. 5d, 5e, and 5f show the behavior

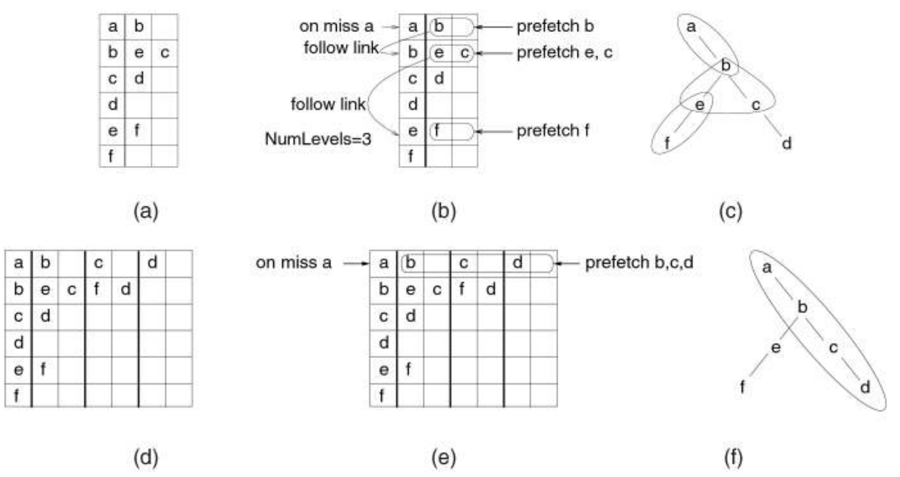


Fig. 5. ((a), (b), and (c)) Comparing the prefetching of several levels of successors in *Chain* and ((d), (e), and (f)) *Replicated*.

of the *Replicated* one. The tables correspond to *NumSucc* = 2 and *NumLevels* = 3. For each algorithm, Figs. 5a and 5d show the state of the tables immediately before we encounter the last a miss in our example.

When the last a miss is encountered in *Chain* (Fig. 5b), we prefetch its immediate successors (b), and then access the entry for b to prefetch e and c. Then, we follow the MRU successor of b, which is e, to prefetch f. The locations that are prefetched by *Chain* are then b, e, c, f. Fig. 5c shows the tree of misses following a and how *Chain* prefetches in three steps. We can see that the b, e, f sequence sidetracks the prefetching for the repeating a, b, c, d sequence. Indeed, while e, f are not successors of a, they are still prefetched. This is because *Chain* does not remember the "successors of miss a" as a unit; it simply follows the successors found along the MRU path. With *Replicated*, when the last a miss is encountered (Fig. 5e), we prefetch the successors of miss a as a unit. We use MRU at each single successor level if necessary, although this is not an issue in the example because there is a single miss per level. The locations prefetched are b, c, d, which are the ones that we want. Fig. 5f shows the tree of misses again, and how *Replicated* remembers and prefetches the true successors of a as a unit. This example illustrates that *Replicated* is more resistant to interfering miss sequences, and thus can prefetch far ahead more accurately than *Chain*.

Overall, 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 are more accurate 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 suboptimal to have all the applications share a single table: the table is likely to suffer 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 corresponding application.

It could be argued that the resulting tables may take so much memory space that they cause additional memory paging and, therefore, negate some of the expected performance gains. In practice, in our experiments, we find

TABLE 1
Comparing Different Pair-Based Correlation Prefetching Algorithms Running on a ULMT

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 2
Applications Used

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

that the average table size is less than 4 Mbytes. Consequently, eight applications require 32 Mbytes, which is 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. Finally, if the resulting space taken by the tables is still too high, we can swap out the tables of all nonexecuting applications. The resulting space requirements would then be only 4 Mbytes, which is certainly a tolerable overhead. In this case, as a new application is scheduled to run, its table can be regenerated.

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 Remapping. Sometimes, a page gets remapped. 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 remapping occurs, passing the old and new physical page numbers. 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

4.1 Applications

To evaluate the ULMT approach, we use nine mostly-irregular, memory-intensive applications. Irregular applications are hard to speed up with 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.

4.2 Simulation Environment

The evaluation is done using an execution-driven simulation environment that supports a dynamic superscalar processor model [20]. 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 microarchitecture of Fig. 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 uniprogrammed 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.

TABLE 3
Parameters of the Simulated Architecture

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

Latencies correspond to contention-free conditions. RT stands for round-trip from the processor. All cycles are 1.6 GHz cycles.

TABLE 4
Parameter Values Used for the Different Algorithms

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*	$NumSeq = 4$, $NumPref = 6$

4.3 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* sequential 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 [16], 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.

4.4 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 NonVerbose mode (Section 3.2). For the *Base* algorithm, we choose the parameter values used by Joseph and Grunwald [15] so that we can compare the work. The last four columns of Table 2 give the size of the correlation table that we use for each application. The table is two-way set-associative. We have sized the number of rows in the table (*NumRows*) as explained in Section 5.1.2. To map miss addresses into the table, we use a trivial hashing function that simply takes the lower bits of the line address. 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.

4.5 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 the instruction count. None of the algorithms uses floating-point operations.

5 Evaluation

5.1 Characterizing Application Behavior

5.1.1 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.

Fig. 6 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-based schemes use large tables to ensure that practically no prediction is missed due to conflicts in the table: *NumRows* is 256 K, *Assoc* is four, and *NumSucc* is four. Under these conditions, 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.

Fig. 6 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 percent and 82 percent of the misses, respectively. Moreover, the best algorithms keep predicting correctly across several levels of successors. For example, *Repl* correctly predicts on average 77 percent and 73 percent 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

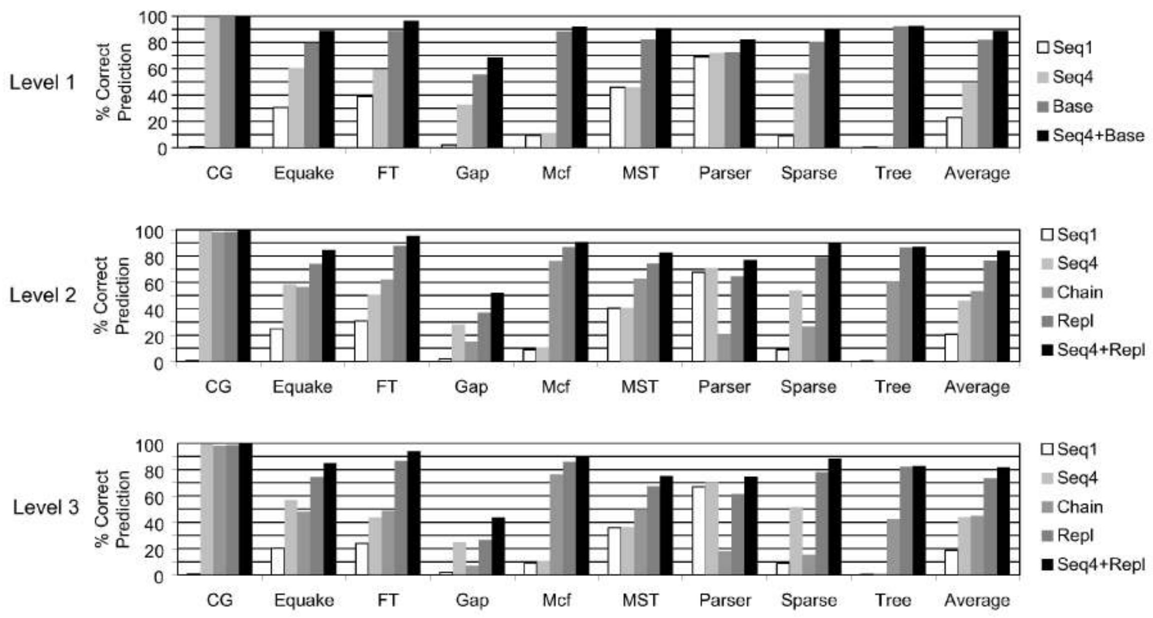


Fig. 6. Fraction of L2 cache misses that are correctly predicted by different algorithms for different levels of successors.

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 multistream sequential prefetching (*Seq4+Repl*).

5.1.2 Sensitivity to the Table Size

To decide how many entries (*NumRows*) to use in the correlation table, we can use different approaches. One approach is to base the decision on the predictability of *level 3* successor misses as defined in Section 5.1.1. We focus on level 3 successor misses because predicting such misses enables these algorithms to prefetch far ahead.

Fig. 7 shows the predictability of *level 3* successor misses for *Seq4 + Repl* for different values of *NumRows*. The figure is organized as Fig. 6, except that we vary *NumRows*. From the figure, we see that the prediction accuracy increases as *NumRows* increases. However, the accuracy stops increasing at a certain value that is application-dependent. We call this point the *Accuracy Threshold*. The accuracy threshold for each application is listed in the second column of Table 5. For example, while Equake, MST, and Sparse have an accuracy threshold of 128 K entries, CG and Parser have a threshold of only 8 K entries. The reason for the small threshold in CG is that, since CG is a fairly regular application, there are relatively few irregular cache miss addresses that need to be stored in the table for *Seq4 + Repl*. Overall, since each application has a different value for the accuracy threshold, we could save memory by setting the *NumRows* for each application to its accuracy threshold.

One problem of finding the accuracy threshold for each application is that it requires expensive profiling. Such profiling can be done offline or online. For example, an offline approach would involve performing multiple profiling runs per application, namely one run for each *NumRows* value that we want to evaluate, and then identifying the knee of the curve.

An alternative approach is to select the *NumRows* that result in a good utilization of the correlation table. This approach is different in that it neglects to consider any effects of conflicts in the table. To see how it is used, Fig. 8 shows the percentage of *NumRows* that are actually used during program execution (or *Occupancy* of the table), for different values of *NumRows*. We can see that the table occupancy for most of our applications is 100 percent at 8K *NumRows*. Then, at a certain *NumRows* value, the occupancy decreases quickly.

Under this approach, the profiling needed to identify a good value for *NumRows* is less expensive. We perform a single profiling run per application using a large correlation table. In our case, we use 256 K *NumRows*, which is the largest requirement in the application suite. We then observe what fraction of the table is used in the profiling run, as shown in the last column of each application in

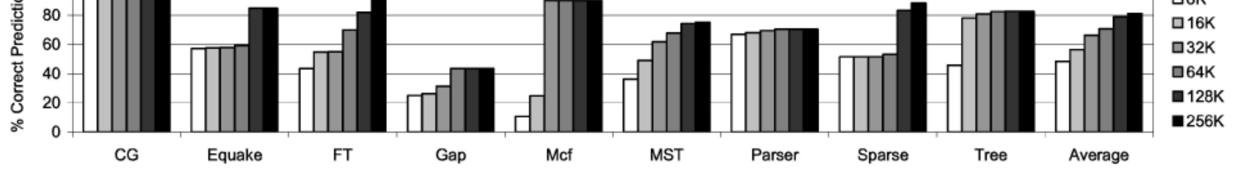


Fig. 7. Effect of the number of entries in the table (*NumRows*) on the accuracy of level 3 successor prediction for *Seq4 + Repl*.

TABLE 5
Determing the Size of the Correlation Table

Application	Accuracy Threshold *NumRows* (K)	Occupancy-Based *NumRows* (K)
CG	8	64
Equake	128	128
FT	256	256
Gap	64	128
Mcf	32	32
MST	128	256
Parser	8	128
Sparse	128	256
Tree	16	8
Average	85	140

Fig. 8. We multiply such a fraction by 256 K and round up the result to the nearest power of two in order to facilitate the simple hashing mechanism used in the table (Section 4.4). With a more advanced hashing function, we could avoid rounding up, and would save space for the table. We call this approach to select the table size as *Occupancy-Based*. The third column of Table 5 shows the resulting size of the table chosen with this approach for each application.

The table shows that *NumRows* in the occupancy-based approach is typically as high or higher than using the accuracy threshold. The difference between the two values is primarily due to infrequent misses, such as initialization misses. They contribute to the occupancy of the table, but they can be displaced from the table without significantly decreasing the accuracy of the miss address prediction. For *Parser*, these misses make the occupancy-based *NumRows* much higher than the accuracy threshold *NumRows*.

One exception is *Tree*, where the *NumRows* in the occupancy-based approach is lower than using the accuracy threshold. The reason is that the miss addresses in *Tree* suffer many conflicts, thereby keeping the address prediction accuracy low. To reach the desired accuracy threshold, we need a larger table. Such a table eliminates these conflicts, but ends up being sparsely populated.

Overall, while *NumRows* in the occupancy-based approach tends to be higher than using the accuracy threshold, we use the former approach. The reason is that it needs a single profiling run per application. Even with this approach, the average table size per application is a tolerable 140 K rows. In the rest of the paper, we use the occupancy-based *NumRows* for each application.

5.1.3 Time between L2 Misses

Another important issue is the time between L2 misses. We classify 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, 40) cycles, [40, 80) cycles, etc. We then cluster adjacent bins that have a similar weight. After that, the resulting bins are: [0, 80), [80, 200), [200, 280), and [280, Infinity), as shown in Fig. 9. In our analysis, the unit is 1.6 GHz processor cycles.

The most significant bin is [200, 280), which contributes with 60 percent 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, Infinity) 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.

5.2 Comparing the Different Algorithms

5.2.1 Execution Time Comparison

Fig. 10 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 (Fig. 10a) or in the memory controller chip (Fig. 10b). For the case in the memory controller chip, we use the *MC* suffix for the names of our memory-side prefetching algorithms. 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 (*BeyondL2*). 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.

We first consider the *NoPref* and *Conven4* bars, which are the same in both Figs. 10a and 10b. On average, *BeyondL2* is the most significant component of the execution time in *NoPref*. It accounts for 44 percent of the time. Thus,

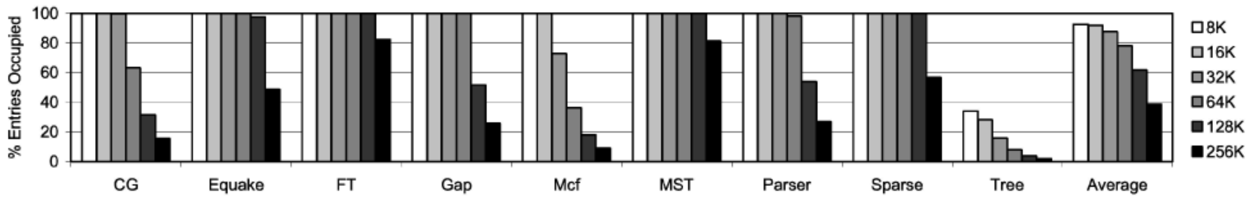


Fig. 8. Occupancy of the correlation table for different values of *NumRows*.

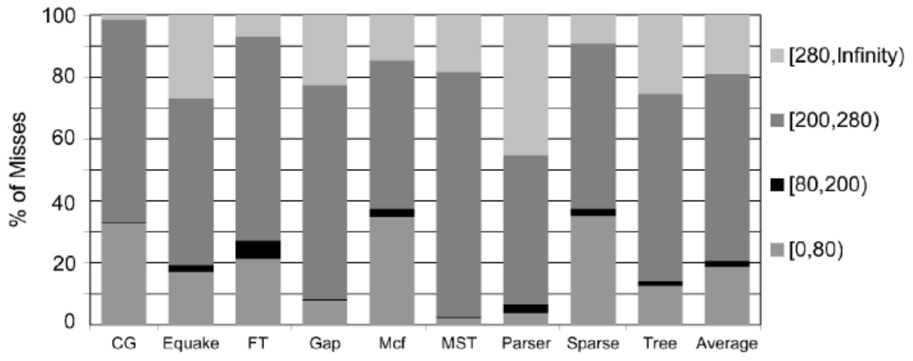


Fig. 9. Characterizing the time between L2 misses.

although our ULMT schemes only target L2 cache misses, they target the main contributor to the execution time.

Conven4 is an effective prefetching technique. On average, it reduces the execution time by 17 percent. However, its impact varies noticeably across applications. For example, *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.

We now consider the pair-based schemes in Fig. 10a. From the figure, we see that they 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 percent. *Chain* performs a little better, but it is limited by inaccuracy (Fig. 6) and high response time (Section 3.3.1). On average, it reduces *NoPref*'s execution time by 12 percent.

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.

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 NonVerbose 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.

Finally, we consider the pair-based schemes in Fig. 10b, which correspond to integrating the memory processor in the memory controller chip. Recall that, with the processor in the North Bridge chip, we have twice the memory access latency (100 cycles versus 56 cycles), eight times lower memory bandwidth (3.2 GB/sec versus 25.6 GB/sec), and an additional 25-cycle delay seen by the prefetch requests before they reach the DRAM.[1]

From the figure, we see that *BaseMC* is significantly slower than *Base*. The main reason for this is that, since the prefetching algorithm used does not prefetch beyond the immediate successors, it is important that the prefetched data be sent to the main processor as soon as possible. The extra latencies observed when the memory processor is in the memory controller greatly reduce the timeliness of the prefetched data.

ChainMC is also slower than *Chain*. In this case, the prefetching algorithm used has a high response time. The response time increases further when the memory processor is in the memory controller chip. This problem could be tolerable if the prefetching algorithm prefetched far ahead accurately. Unfortunately, we have seen that this prefetching algorithm is relatively inaccurate for successors beyond immediate ones.

Finally, if we compare *Repl* to *ReplMC* and *Conven4* + *Repl* to *Conven4* + *ReplMC*, we see that the differences are very small. The reason is that the memory-side prefetching algorithm used has the ability to prefetch far ahead accurately. As a result, by putting the memory processor in the memory controller chip, we may hurt the timeliness of the immediate successor prefetches. However, the prefetching of further levels of successors is still timely. Overall, therefore, putting the processor in the DRAM or in the memory controller chip makes little difference. Specifically, going from *Conven4* + *Repl* to *Conven4* + *ReplMC*, reduces the average speedups relative to *NoPref* from 1.46 to 1.41.

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, as long as the best algorithm (*Conven4* + *ReplMC*) is used.

5.2.2 Algorithm Customization

In this first paper on ULMT prefetching, we have tried only very simple ad hoc customizations for a few applications; we do not attempt any systematic analysis of possible customizations. Table 6 shows the customizations performed.

For CG, we observe that, while it only has sequential miss patterns (Fig. 6), its multiple streams may overwhelm the conventional prefetcher. Indeed, although processor-side prefetches are very accurate (99.8 percent of the prefetched lines are used), they are often not timely (only 64 percemt arrive to L2 before being needed). In our customization, we try to exploit positive interaction between processor and memory-side prefetching. To do so, we turn on the Verbose mode so that processor-side prefetch requests are seen by the ULMT. Furthermore, the

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

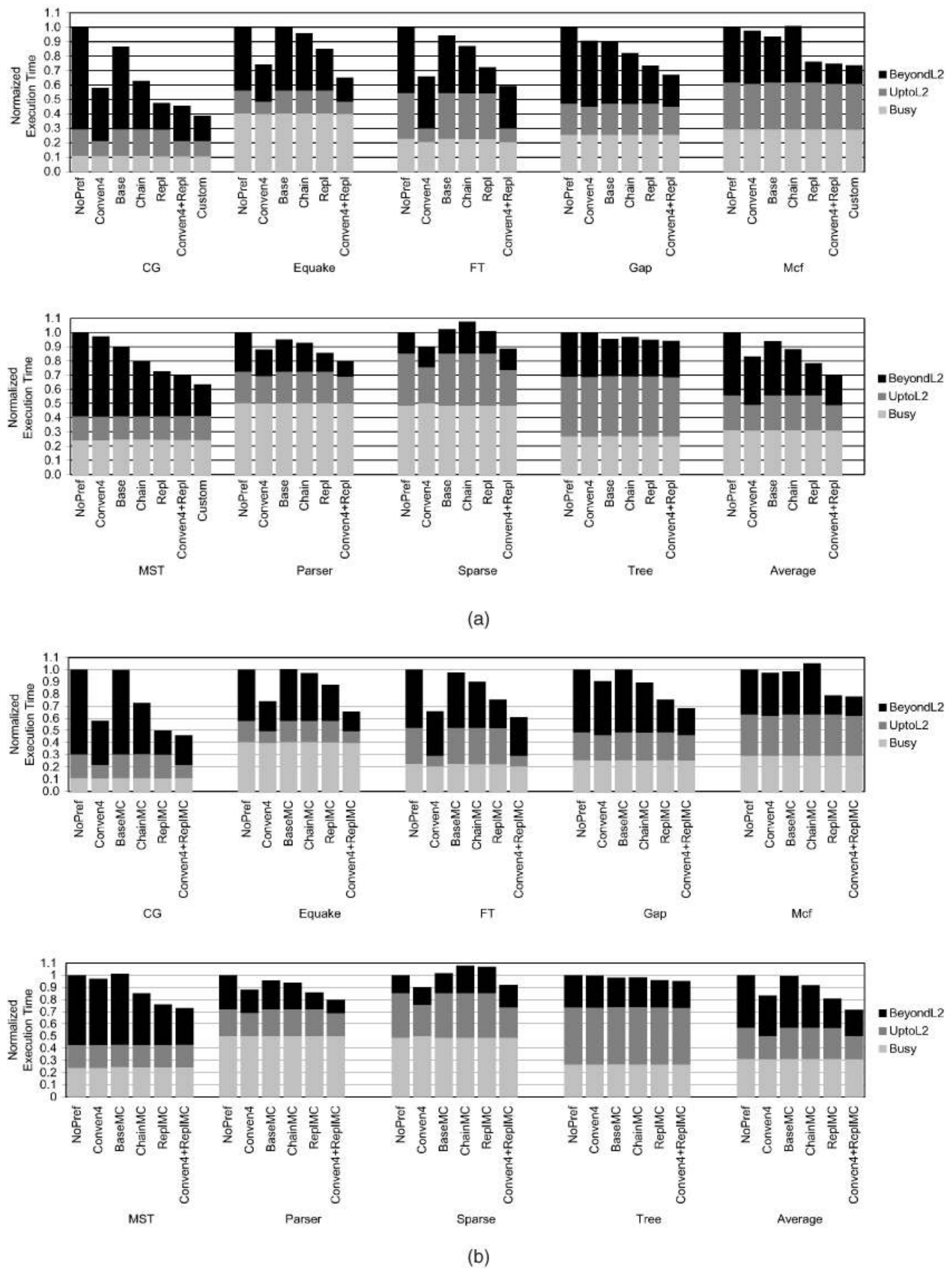


Fig. 10. Execution time of the applications with different prefetching algorithms. In (a), the memory processor is placed in the DRAM chip, while in (b), it is placed in the memory controller.

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, more processor-side prefetches arrive in a timely manner.

The result of this customization is shown in Fig. 10 as the *Custom* bar in CG. The speedup of CG improves from 2.19 (with *Conven4* + *Repl*) to 2.59. It can be shown that up to 81 percent of the processor-side prefetches arrive in a timely manner. This case demonstrates that even regular applications that are amenable to sequential processor-side prefetching can benefit from ULMT prefetching.

TABLE 6
Customizations Performed

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

In all cases, Conven4 is also on.

We have also taken some applications for which *Repl* ideally predicts well even the third level of successor misses (*Level 3* in Fig. 6) and examined prefetching a fourth level of successor misses. We choose the MST and Mcf applications. Our customization involves increasing *NumLevels* to 4 in *Repl*. As shown in the *Custom* bars in Fig. 10, this approach is successful for MST, but produces marginal gains in Mcf.

We have also examined other customization techniques, which provide only negligible gains. A systematic analysis of customization is left for future work. However, 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* improves from 1.46 (with *Conven4* + *Repl*) to 1.53.

5.2.3 Prefetching Effectiveness

To gain further insight into these prefetching schemes, we measure the misses and prefetch hits in the L2 cache (Fig. 11a), and the lines prefetched into the L2 cache by the ULMT (Fig. 11b). The latter are called *prefetches*. Both figures are shown in the *same scale*, namely normalized to the number of L2 misses in the *NoPref* environment. The figures show data for Sparse, Tree, and the average of the remaining seven applications. We show the data for Sparse and Tree separately because these applications are the hardest to prefetch successfully.

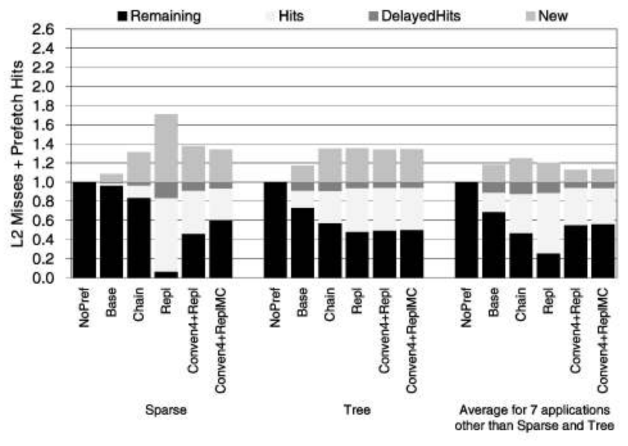
Fig. 11a combines several types of L2 misses and hits: original L2 misses that still remain after prefetching (*Remaining*), original L2 misses that are eliminated by the ULMT prefetches, either completely (*Hits*) or partially because the prefetches arrive a bit late (*DelayedHits*), and new L2 misses induced by the prefetches that displace useful L2 lines (*New*). By definition, the sum of *Remaining*, *Hits*, and *DelayedHits* is one. The sum of *Hits* and *DelayedHits* is the coverage of the ULMT prefetching algorithm. Due to the way we measure the data, *Remaining* also includes the original L2 misses that are successfully eliminated by the processor-side conventional prefetcher (in *Conven4* + *Repl* and *Conven4* + *ReplMC*).

Fig. 11b classifies the prefetches that arrive to L2: those that completely or partially eliminate a miss (*Useful*), and those that are useless, either because they are not referenced between the time they arrive to L2 and the time they are replaced (*Replaced*), or because they are dropped on arrival to L2 because the same line is already in the cache (*Redundant*). Note that *Useful* is equal to *Hits* plus *DelayedHits* in Fig. 11a. Moreover, *Useful* divided by all the sum of *Useful*, *Replaced*, and *Redundant*, is the accuracy of the prefetching algorithm.

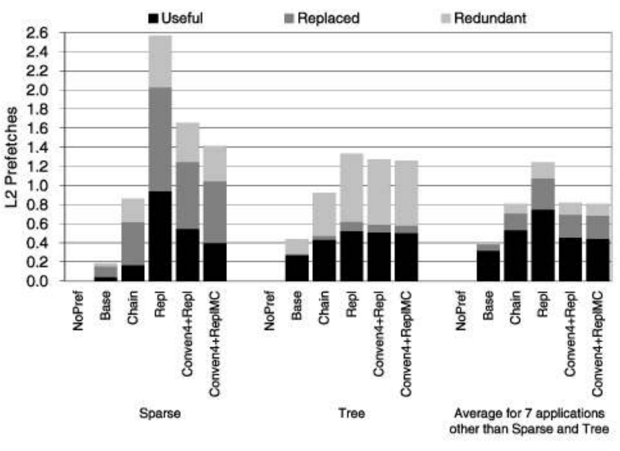
We consider the average of the seven applications first. Fig. 11a shows why *Base* and *Chain* deliver only moderate performance gains in Fig. 10: their coverage is modest and the prefetches generate *New* misses. *Base*'s coverage is hurt by its inability to prefetch far ahead, while *Chain*'s is hampered by its high response time and limited accuracy. The limited accuracy of *Chain* is seen in Fig. 11b.

Fig. 11a shows that *Repl* has a high coverage (0.74). However, this high coverage comes at the cost of also some *New* misses and limited accuracy. Fig. 11a shows that *New* misses are equivalent to 20 percent of the original misses. Moreover, Fig. 11b shows that the resulting accuracy is about 60 percent.

Conven4 + *Repl* delivers the best speedups in Fig. 10, thanks to a combination of three factors: few induced misses (*New* in Fig. 11a), a modest number of useless prefetches (Fig. 11b), and very good coverage. The total coverage is not shown in Fig. 11a. The figure only shows the coverage of ULMT prefetches. It does not show the additional coverage provided by the prefetch requests issued by *Conven4*. These prefetch requests target the regular miss patterns and eliminate some of the misses that Fig. 11a still shows as *Remaining* in the *Conven4* + *Repl* bar. The ULMT prefetcher does not see these prefetch requests in NonVerbose mode, and it only focuses on the irregular miss patterns. The resulting, total coverage of *Conven4* + *Repl* could be shown to be even higher than *Repl*.



(a)



(b)

Fig. 11. (a) Analysis of the misses and prefetch hits in the L2 cache and (b) the lines prefetched into the L2 cache by the ULMT. Both figures are shown in the same scale, normalized to the number of L2 misses in *NoPref*.

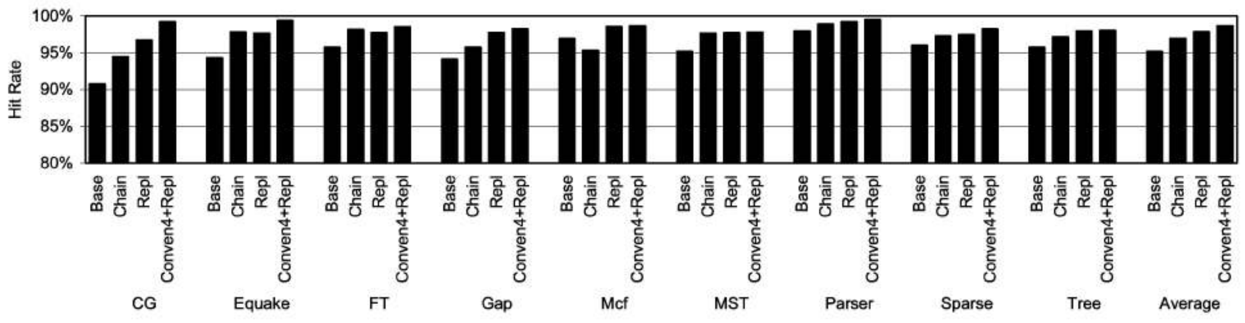


Fig. 12. Hit rate of the data cache of the memory processor. The data corresponds to when the memory processor is in the DRAM.

Conven4 + *ReplMC* has a behavior similar to *Conven4* + *Repl*.

Finally, the figures also show why *Sparse* and *Tree* had limited speedups in Fig. 10. These applications suffer many conflicts in the cache, which results in many *New* misses (Sparse) or a combination of many *New* and *Remaining* misses (Tree). Moreover, the prefetches are not very accurate, which results in large *Replaced* and *Redundant* categories.

5.2.4 Cache Performance of the ULMT

To understand the behavior of the ULMT algorithms better, Fig. 12 shows the hit rate of the data cache of the memory processor when the ULMT executes. We consider the case when the memory processor is in the DRAM, and examine the *Base*, *Chain*, *Repl*, and *Conven4* + *Repl* algorithms. We do not show the case when the memory processor is in the memory controller chip because the results are very similar.

The figure shows that the hit rate in *Repl* is higher than in *Chain*. This is because *Repl* has better cache line reuse than *Chain* (Section 3.3.2), despite the fact that *Repl* handles larger tables than *Chain*.

Note that the hit rate of *Repl* is higher when conventional hardware prefetching is also used in the main processor (*Conven4* + *Repl* bars). This is due to the filtering effect of the processor-side conventional prefetching. It reduces the number of cache misses seen by the ULMT, effectively reducing the working set of the ULMT and improving the hit rate.

5.2.5 Work Load of the ULMT

Further understanding of the behavior of the ULMT algorithms can be obtained from Fig. 13. The figure 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 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 (Fig. 9). 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.

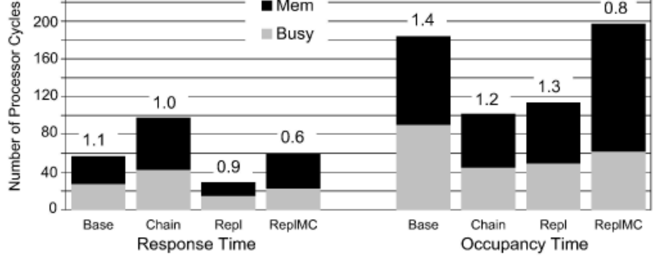


Fig. 13. Average response and occupancy time of different ULMT algorithms in main-processor cycles.

5.2.6 Main Memory Bus Utilization

Finally, Fig. 14 shows the utilization of the main memory bus for various algorithms, averaged over all applications. As usual, we consider both placing the memory processor in the DRAM and placing it in the memory controller chip. The increase in bus utilization induced by the prefetching 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 percent to only 36 percent in the worst case (*Conven4* + *Repl*). Moreover, most of the increase comes from the faster execution; only a 6 percent 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. As expected, bus utilization is slightly lower when the memory processor is in the memory controller chip. The reason is that the application takes a bit longer to completed execution. This effect can be seen for *BaseMC* and *ChainMC*.

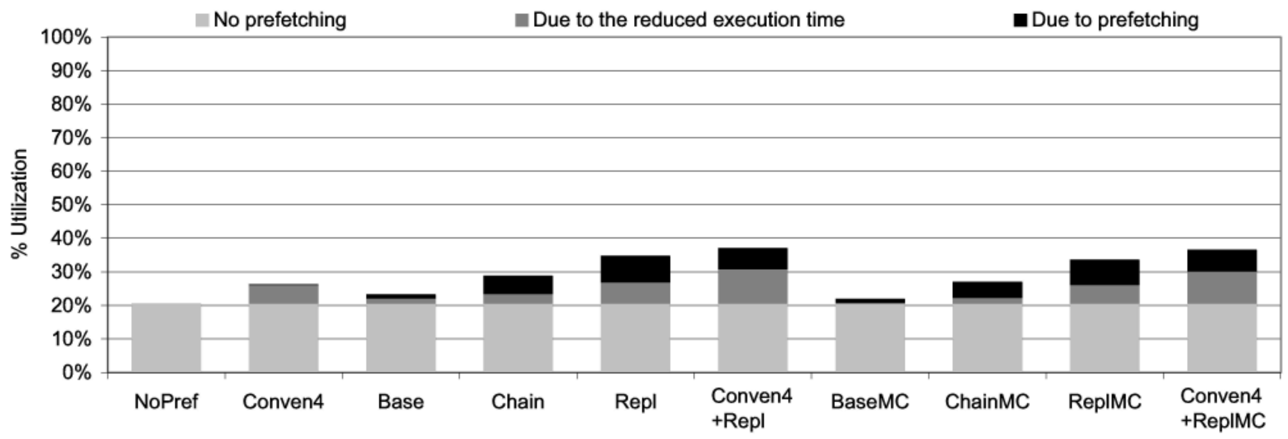


Fig. 14. Main memory bus utilization.

6 Related Work

6.1 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 [27]. 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 [14] and Yang and Lebeck [35] 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.

6.2 Correlation Prefetching

Early work on correlation prefetching can be found in [2], [29]. 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 [15]. 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 [32]. Lai et al. [21] and Hu et al. [12] design a slightly different correlation prefetcher. 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. Finally, Hu et al. [13] propose correlating cache tags instead of addresses.

We differ from most of 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] and Hu et al. [13], 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.

6.3 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 [16], Palacharla and Kessler [28], and Sherwood et al. [32]. We base our processor-side prefetcher on these schemes.

6.4 Processor-Side Prefetching

There are many more proposals for processor-side prefetching, often for irregular applications. A tiny, nonexhaustive list includes Choi et al. [8], Karlsson et al. [17], Lipasti et al. [23], Luk and Mowry [24], Mehrotra [25], Roth et al. [30], and Zhang and Torrellas [36]. Many of these schemes specifically target linked data structures. Many of them rely on program information that is available to the processor, like the addresses and sizes of data structures. Often, they need compiler support. Our scheme needs neither program information nor compiler support.

6.5 Other Related Work

There are other proposals for using an additional thread to perform some specific functions. We give two examples to illustrate.

One area of work is what is called "helper threads." Dubois and Song [11] propose fine-grain threads (nanothreads) that run on the same processor as the main program, and perform sequential and stride prefetching in software for the main program. 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. Preexecution and precomputation by Roth and Sohi [31] and many others use a thread that prefetches for the main program. In contrast, our ULMT is not tightly integrated with the main program and is not derived from the main program code. Moverover, it is customizable and runs in the main memory system, where it does not suffer the same memory access latency as the main processor.

Lee et al. [22] use a ULMT in a similar platform as in this paper to coexecute code sections that are memory intensive. A set of compiler and runtime algorithms partition the code into sections that have uniform computation and memory behavior called *modules*, schedule the modules to the most appropriate processor, and try to overlap execution of main and memory processor. Compared to that work, ULMT for correlation prefetching needs simpler support: applications do not need recompilation, the memory processor does not need to support floating point, and there is no need to support cache coherence between main and memory processor because the application thread and the ULMT share neither data nor instructions. However, coexecution can gain speedups from parallel execution of main and memory processor.

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.

Acknowledgments

An early version of this paper appeared in [33]. Much of this work was done while Yan Solihin was at the Department of Computer Science, University of Illinois at Urbana-Champaign, and while Jaejin Lee was at the Department of Computer Science and Engineering, Michigan State University. This work was supported in part by the US 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. This work was also supported by North Carolina State University, and in part by the Korean Ministry of Education under the BK21 program and by the Korean Ministry of Science and Technology under the National Research Laboratory Program.

References

[1] T. Alexander and G. Kedem, "Distributed Predictive Cache Design for High Performance Memory Systems," *Proc. Second Int'l Symp. High-Performance Computer Architecture,* pp. 254-263, Feb. 1996.

[2] J.L. Baer, "Dynamic Improvements of Locality in Virtual Memory Systems," *IEEE Trans. Software Eng.,* vol. 2, pp. 54-62, Mar. 1976.

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

[4] J.B. Carter, et al. "Impulse: Building a Smarter Memory Controller," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture,* pp. 70-79, Jan. 1999.

[5] R.S. Chappell, J. Stark, S. Kim, S.K. Reinhardt, and Y.N. Patt, "Simultaneous Subordinate Microthreading (SSMT)," *Proc. 26th Int'l Symp. Computer Architecture,* pp. 186-195, May 1999.

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

[7] T.F. Chen and J.L. Baer, "Reducing Memory Latency via Non-Blocking and Prefetching Cache," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 51-61, Oct. 1992.

[8] S. Choi, D. Kim, and D. Yeung, "Multi-Chain Prefetching: Effective Exploitation of Inter-Chain Memory Parallelism for Pointer-Chasing Codes," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 51-61, Sept. 2001.

[9] R. Cooksey, D. Colarelli, and D. Grunwald, "Content-Based Prefetching: Initial Results," *Proc. Second Workshop Intelligent Memory Systems,* pp. 33-55, Nov. 2000.

[10] J. Dongarra, V. Eijkhout, and H. van der Vorst, "SparseBench: A Sparse Iterative Benchmark," http://www.netlib.org/benchmark/sparsebench, 2003.

[11] M. Dubois and Y.H. Song, "Assisted Execution," CENG Technical Report 98-25, Dept. of Electrical Eng.-Systems, Univ. of Southern California, Oct. 1998.

[12] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior," *Proc. 29th Int'l Symp. Computer Architecture,* May 2002.

[13] Z. Hu, M. Martonosi, and S. Kaxiras, "Tag Correlating Prefetchers," *Proc. Ninth Int'l Symp. High-Performance Computer Architecture,* Feb. 2003.

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

[15] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 252-263, June 1997.

[16] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Int'l Symp. Computer Architecture,* pp. 364-373, May 1990.

[17] M. Karlsson, F. Dahlgren, and P. Stenstrom, "A Prefetching Technique for Irregular Accesses to Linked Data Structures," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture,* pp. 206-217, Jan. 2000.

[18] D. Koufaty and J. Torrellas, "Comparing Data Forwarding and Prefetching for Communication-Induced Misses in Shared-Memory MPs," *Proc. Int'l Conf. Supercomputing,* pp. 53-60, July 1998.

[19] C. Kozyrakis, et al. "Scalable Processors in the Billion-Transistor Era: IRAM," *IEEE Computer,* pp. 75-78, Sept. 1997.

[20] V. Krishnan and J. Torrellas, "A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 286-293, Oct. 1998.

[21] A. Lai, C. Fide, and B. Falsafi, "Dead-Block Prediction and Dead-Block Correlating Prefetchers," *Proc. 28th Int'l Symp. Computer Architecture,* pp. 144-154, June 2001.

[22] J. Lee, Y. Solihin, and J. Torrellas, "Automatically Mapping Code on an Intelligent Memory Architecture," *Proc. Seventh Int'l Symp. High Performance Computer Architecture,* Jan. 2001.

[23] M.H. Lipasti, W.J. Schmidt, S.R. Kunkel, and R.R. Roediger, "Spaid: Software Prefetching in Pointer and Call Intensive Environments," *Proc. 28th Int'l Symp. Microarchitecture,* pp. 231-236, Nov. 1995.

[24] C. Luk and T.C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 222-233, Oct. 1996.

[25] S. Mehrotra, "Data Prefetch Mechanisms for Accelerating Symbolic and Numeric Computation," PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1996.

[26] MIPS, MIPS R10000 Microprocessor User's Manual, Version 2.0, Jan. 1997.

[27] NVIDIA, technical brief: NVIDIA nForce Integrated Graphics Processor (IGP) and Dynamic Adaptive Speculative Pre-Processor (DASP), http://www.nvidia.com/, 2002.

[28] S. Palacharla and R. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," *Proc. 21st Int'l Symp. Computer Architecture,* pp. 24-33, Apr. 1994.

[29] J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio, "Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks," US Patent 4,807,110, Feb. 1989.

[30] A. Roth, A. Moshovos, and G. Sohi, "Dependence Based Prefetching for Linked Data Structures," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 115-126, Oct. 1998.

[31] A. Roth and G. Sohi, "Speculative Data Driven Multithreading," *Proc. Seventh Intl. Symp. High-Performance Computer Architecture,* Jan. 2001.

[32] T. Sherwood, S. Sair, and B. Calder, "Predictor-Directed Stream Buffers," *Proc. 33rd Int'l Symp. Microarchitecture,* pp. 42-53, Dec. 2000.

[33] Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," *Proc. 29th Int'l Symp. Computer Architecture,* May 2002.

[34] Sony Computer Entertainment Inc., http://www.sony.com, 2003.

[35] C.-L. Yang and A.R. Lebeck, "Push versus Pull: Data Movement for Linked Data Structures," *Proc. Int'l Conf. Supercomputing,* pp. 176-186, May 2000.

[36] Z. Zhang and J. Torrellas, "Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 188-199, June 1995.

Yan Solihin received the BS degree in computer science from Institut Teknologi Bandung, Indonesia, in 1995, the MASc degree in computer engineering from Nanyang Technological University, Singapore, in 1997, and the MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign in 1999 and 2002. He is currently an assistant professor at the Department of Electrical and Computer Engineering at the North Carolina State University. From 1999 to 2000, he was on an internship with the Parallel Architecture and Performance team at Los Alamos National Laboratory. His research interests include high-performance computer architectures, emerging memory systems, and performance modeling. He has authored/coauthored several papers in each area and released several software packages, including Scaltool 1.0, a predictive-diagnostic tool for parallel program scalability bottleneck, at the National Center for Supercomputing Applications (NCSA) in 1999. He was a recipient of the AT&T Leadership Award in 1997. More information can be found at http://www.cesr.ncsu.edu/solihin. He is a member of the IEEE.

Jaejin Lee received the BS degree in physics from Seoul National University in 1991, the MS degree in computer science from Stanford University in 1995, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1999. He is an assistant professor in the School of Computer Science and Engineering at Seoul National University, Korea, where he has been a faculty member since September 2002. Before joining Seoul National University, he was an assistant professor in the Computer Science and Engineering Department at Michigan State University. He was a recipient of an IBM Cooperative Fellowship and a fellowship from the Korea Foundation for Advanced Studies during his PhD study. His research interests include compilers, computer architectures, embedded computer systems, and systems in high-performance computing. He is a member of the IEEE and ACM. More information can be found at http://aces.snu.ac.kr/~jlee.

Josep Torrellas received the PhD in electrical engineering from Stanford University in 1992. He is a professor and Willett Faculty Scholar in the Computer Science Department of the University of Illinois at Urbana-Champaign. He is also vice-chairman of the IEEE Technical Committee on Computer Architecture (TCCA). In 1998, he was on sabbatical at the IBM TJ Watson Research Center. He has published more than 100 papers in computer architecture, which cover scalable and chip multiprocessor architectures, processing-in-memory architectures, and speculative multithreading. He received a 1994 US National Science Foundation Young Investigator Award and a 1997 IBM Partnership Award. He has been on the organizing committee of many international conferences and workshops. He is a senior member of IEEE. More information can be found at http://iacoma.cs.uiuc.edu/~torrellas.