Prefetching with Helper Threads for Loosely Coupled Multiprocessor Systems

Jaejin Lee, *Member*, *IEEE*, Changhee Jung, *Student Member*, *IEEE*,
Daeseob Lim, *Student Member*, *IEEE*, and Yan Solihin, *Senior Member*, *IEEE*

Abstract—This paper presents a helper thread prefetching scheme that is designed to work on loosely coupled processors, such as in a standard chip multiprocessor (CMP) system or an intelligent memory system. Loosely coupled processors have an advantage in that resources such as processor and L1 cache resources are not contended by the application and helper threads, hence preserving the speed of the application. However, interprocessor communication is expensive in such a system. We present techniques to alleviate this. Our approach exploits large loop-based code regions and is based on a new synchronization mechanism between the application and helper threads. This mechanism precisely controls how far ahead the execution of the helper thread can be with respect to the application thread. We found that this is important in ensuring prefetching timeliness and avoiding cache pollution. To demonstrate that prefetching in a loosely coupled system can be done effectively, we evaluate our prefetching by simulating a standard unmodified CMP system and an intelligent memory system where a simple processor in memory executes the helper thread. Evaluating our scheme with nine memory-intensive applications with the memory processor in DRAM achieves an average speedup of 1.25. Moreover, our scheme works well in combination with a conventional processor-side sequential L1 prefetcher, resulting in an average speedup of 1.31. In a standard CMP, the scheme achieves an average speedup of 1.33. Using a real CMP system with a shared L2 cache between two cores, our helper thread prefetching plus hardware L2 prefetching achieves an average speedup of 1.15 over the hardware L2 prefetching for the subset of applications with high L2 cache misses per cycle.

Index Terms—Helper thread, prefetching, chip multiprocessors, processing-in-memory system.

1 Introduction

Data prefetching tolerates long memory access latency by predicting which data in memory is needed in the future and fetches it to the cache before it is accessed. To deal with irregular access patterns that are hard to predict, one recent class of prefetching techniques that relies on a helper thread has been proposed [1], [2], [3], [4], [5], [6], [7], [8], [9]. The helper thread executes an abbreviated version of the application code ahead of the application execution, bringing data into the cache early to avoid the application's cache misses.

Prior studies of helper thread prefetching schemes have relied on a tightly coupled system where the application and the helper thread run on the same processor in a simultaneous multithreaded (SMT) system [2], [3], [4], [5], [6], [9], [10]. Using a tightly coupled system has a major drawback that the application and helper threads contend for fine-grain resources such as processor and L1 cache resources. Partitioning resources between the threads can remove the contention; however, it introduces hardware modifications into the critical path of the processor cores. Alternatively, the resource contention for the helper thread can be managed by imposing priorities among the threads [11]. For example, IBM Power5, a commercial SMT implementation, supports enhanced SMT features like dynamic resource balancing and software-controlled thread prioritization [12]. However, this requires modifications to the operating system and the processor's front end and reduces the effectiveness of the helper thread.

This paper presents a helper thread prefetching scheme that is designed for loosely coupled processors, such as in a standard chip multiprocessor (CMP) or an intelligent memory system. Loosely coupled processors have an advantage that threads do not contend for resources such as the processor and L1 cache. The lack of contention preserves the application thread's speed and avoids unnecessary hardware at the processor's critical path. However, high interprocessor communication latency in loosely coupled processors presents a challenge.

The main contributions of this paper are architecture and compiler techniques that enable effective helper thread prefetching for loosely coupled multiprocessors. The contributions include

- a loop-based helper thread extraction algorithm based on modules [13], [14], yielding very large prefetching regions (millions of instructions),
- a coarse-grain synchronization mechanism for precise control of the execution of the prefetching thread without high synchronization overheads or the requirement of custom synchronization hardware in CMP,

- *J. Lee and D. Lim are with the School of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea. E-mail: jlee@cse.snu.ac.kr, daeseob@aces.snu.ac.kr.*
- *C. Jung is with the School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332-0280. E-mail: chjung@gatech.edu.*
- *Y. Solihin is with the Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911. E-mail: solihin@eos.ncsu.edu.*

Manuscript received 4 Apr. 2008; revised 18 Sept. 2008; accepted 23 Sept. 2008; published online 3 Oct. 2008.
Recommended for acceptance by R. Bianchini.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-04-0129.
Digital Object Identifier no. 10.1109/TPDS.2008.224.

- a mechanism for the prefetching thread to instantly catch up to the application execution,
- characterization and analysis of the effects of nonmemory operations on the helper thread's performance, and
- architecture mechanisms to support intelligent memory prefetching: cache coherence extension and synchronization registers.

To demonstrate the effectiveness of our approach, we test it on two platforms that differ on how loosely coupled the processors are. The first platform is a standard CMP system where there are two identical processor cores in a single chip. Each core has its own primary cache, and all cores share a secondary (L2) cache. In this system, the application and the helper thread run on separate cores. We use both a simulated CMP architecture and a real CMP architecture such as Intel Core Duo for our evaluation.

The second platform tested is an intelligent memory system where a simple general-purpose core is embedded in the memory system. The core can be placed in different locations, such as in the *Double Inline Memory Module* (DIMM) or in the DRAM chips. The intelligent memory architecture has a heterogeneous mix of processors: the main processor, which is more powerful and backed up by a deep cache hierarchy due to its high memory access latency, and a memory processor, which is less powerful but has a smaller memory latency. The memory processor is loosely coupled to the main processor because it does not share any cache hierarchy with the main processor. In this system, the helper thread runs on the memory processor, while the application runs on the main processor.

In a high-throughput computing environment, it might be difficult to find an idle core on which a prefetching helper thread can run because all cores of the CMP are likely to be busy. In this case, the memory processor in an intelligent memory system would be a good target for running the prefetching helper thread. Also, it is possible to enable the helper thread prefetching for systems lacking a secondary core (or context) by replacing its memory subsystem with the intelligent memory subsystem.

Running a prefetching thread in an idle CMP core introduces contention at the shared lowest level cache and off-chip bandwidth, which is a very limited resource in a CMP. Moreover, in a high-throughput computing environment, it might be difficult to find an idle core on which a prefetching helper thread can run because all cores of the CMP are likely to be busy. In contrast, running a prefetching thread in memory does not introduce cache space contention or bandwidth contention with the main processor. The memory access latency of a memory processor is much less than that of a core in a CMP, since the memory processor is physically closer to the main memory. This can benefit applications with pointer-chasing behaviors. However, CMP is a mainstream architecture, while IM is not.

We present a helper thread extraction algorithm that can be automatically constructed by the compiler from the application program. The helper thread contains only computations that are essential for address calculation. In addition, because our design should work even when there is no shared cache between the application and the helper thread, the helper thread is augmented with explicit prefetch instructions that prefetch (and push) the data to the L2 cache.

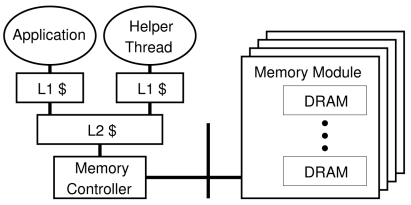


Fig. 1. The CMP architecture used in this paper.

In addition, we characterize the relationship between prefetching effectiveness and the relative speed of the helper thread to the application thread. Based on the observation, we propose a new busy-waiting synchronization mechanism that controls how far ahead the execution of the helper thread can be with respect to the application. The synchronization also detects the case where the helper thread execution lags behind the application and allows the helper thread to instantly catch up to the application execution.

To support our prefetching scheme, simple hardware modifications are made to the intelligent memory, while no hardware modifications are made to the CMP. The intelligent memory modifications include three synchronization registers to facilitate fast synchronization, the main processor's L2 cache modification to enable accepting incoming prefetches initiated by the memory processor, and a per-line stale bit in the memory to enforce coherence automatically between the main and memory processors.

Despite the simple hardware support, we show that our scheme is effective. It delivers an average speedup of 1.25 for nine memory-intensive applications in an intelligent memory where the memory processor is integrated in the DRAM. Furthermore, our scheme works well in combination with a conventional sequential hardware L1 prefetcher, achieving an average speedup of 1.31. When the memory processor is integrated in the DIMM, the average speedup is 1.26. For a two-way unmodified CMP architecture, our scheme delivers an average speedup of 1.33. Finally, our scheme plus hardware L2 prefetching achieves an average speedup of 1.15 over the hardware L2 prefetching on a CMP system with an Intel Core Duo processor for the subset of applications with high L2 cache misses per cycle.

The rest of the paper is organized as follows: Section 2 describes the architectures and hardware support used in this paper. Section 3 presents helper thread construction and synchronization mechanisms between the application and helper threads. Section 4 describes our evaluation environment. Section 5 discusses the relationship between the speed of the helper thread and prefetching effectiveness. Sections 6 and 7 present the evaluation results both with simulation and with a real CMP system. Section 8 discusses related work, and Section 9 concludes the paper.

2 Architectures and Hardware Support

2.1 The CMP Architecture

For the CMP architecture, the application and helper threads are run on different processors in the same chip, as shown in Fig. 1. We assume that the L1 caches are write-through, and an invalidation-based coherence protocol between L1 and L2 is supported (Section 2.3).

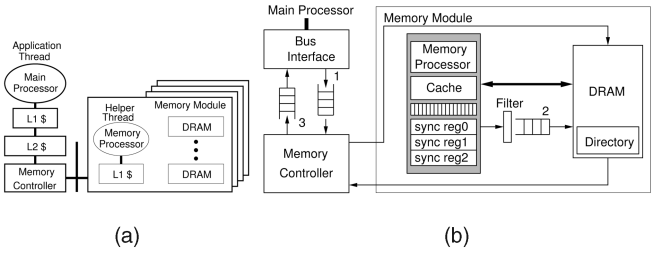


Fig. 2. The intelligent memory architecture used in this paper. (a) The location of the memory processor. (b) The hardware support for prefetching.

2.2 The Intelligent Memory Architecture

For intelligent memory, we need the extra hardware support shown in Fig. 2. Fig. 2a shows the architecture of a system that integrates the memory processor in the DRAM chips or in the memory module (e.g., DIMM). Embedding processors in the DRAM chips allows very low latency and high bandwidth access to the DRAM; however, it requires modifying the DRAM chips and DRAM interface and adding inter-DRAM chip communication support. Embedding a simple processor in the DIMM requires fewer modifications. First, commodity DRAM chips can be used because they do not need to be modified. Second, fewer processors are needed because typically, several DRAM chips share a module. Finally, since all transactions go through the module, the memory processor can readily snoop existing transactions or generate new prefetch transactions. In the following, we will describe the architecture support for the intelligent memory architecture in detail.

As shown in Fig. 2b, a miss request from the main processor is deposited in *queue1*. When the memory processor executes a prefetch instruction, it generates a prefetch request that not only places the data in its own cache but also pushes the data to the L2 cache of the main processor. The request is placed in *queue2* after being filtered by a *Filter* module. The Filter module is a fixed-size FIFO list that keeps most recent prefetch addresses. When there is a new prefetch request, it compares the requested address with the addresses in the list. If there is a match, the request is dropped. Otherwise, the request is placed in *queue2*, and the address is added to the tail of the Filter's list. The Filter module drops unnecessary prefetches and improves prefetching performance when there are prefetch requests to the same block address (but not necessarily to the same words) issued in a short time period.

Replies from memory (for both miss and prefetch requests) to the main processor are placed in *queue3* and then sent to the L2 cache of the main processor via a bus interface. When the replies reach the memory controller, their addresses are compared to the main processor's miss requests in *queue1*. If the fetched line matches a miss request from the main processor, it is considered to be the reply of the miss request in *queue1*. This miss request is not sent to the memory.

The main processor's L2 cache in the intelligent memory architecture needs to be modified so that it can accept prefetched lines from the memory that it has not requested. The L2 cache uses free Miss Status Handling Registers (MSHRs) for this. If the L2 cache has a pending request and a prefetched line with the same block address arrives, the line simply steals the MSHR and updates the cache as if it were the reply. If it does not have an existing pending request, a request that matches the prefetched line is created. Finally, a prefetched line arriving at L2 is dropped in the following cases: the L2 cache already has a copy of the line, the write-back queue has a copy of the line because the L2 cache is trying to write it back to memory, all MSHRs are busy, or all the lines in the set where the prefetched line wants to go are in transaction-pending state.

Fig. 2b shows that the memory processor has three special synchronization registers. Two of them are used to record the relative progress of the main thread and the helper thread. The other is used to transfer the value of the synchronization variable from the application thread to the helper thread. The main processor accesses these registers as coprocessor registers. The details of the synchronization mechanisms using these registers are explained in Section 3.3.

2.3 Cache Coherence

Since our helper thread and the application thread share data, we need to guarantee data coherence in the caches by guaranteeing that a read operation gets the value from the latest write. Since the helper thread construction algorithm removes writes to shared data, we only need to consider writes by the main processor and reads by the memory processor. In addition, the degree of sharing data between the application and helper threads is not that high because of the way we construct the helper thread (i.e., privatization in Section 3.1). Thus, the overhead of cache coherence does not significantly affect prefetching performance.

Coherence for CMP. We assume that the L2 cache enforces invalidation-based coherence between the L1 caches of the processors. When a processor suffers an L1 read miss, the L2 cache supplies the data. Since both L1 caches are write-through, the L2 cache always has the latest copy of a cache line. When a processor suffers an L1 write miss, the L2 cache supplies the data and invalidates the copy of the line in the other L1 cache if the line is cached there. When a processor writes to a line in the L1 cache, the value is written to the L2 cache, and the L2 cache invalidates the copy of the line in the other L1 cache if the line is cached there.

Coherence for intelligent memory. To track the main processor's writes, the memory processor maintains a *stale bit* for each line in the main memory. The directory is stored in the main memory and cached by the memory processor in the directory cache. Fig. 2b shows that the memory processor caches some of the stale bits in its chip.

If the stale bit is set, it indicates that the block is cached by the main processor and its state is modified. If the stale bit is not set, the block is either uncached by the main processor or cached in the clean state.

Initially, the stale bit for each block is clear. If the main processor suffers a read miss, the block is returned to the main processor. If the main processor suffers a write miss, it sends a *stale bit update* request in addition to the miss request to the memory. The stale bit update request sets the stale bit (to one) in the directory with the miss request, and the block is returned to the processor. This includes the case

when the main processor writes to a line that is in a clean state. If, on the other hand, the write is to a line that is already in the modified state, no request is generated to the memory processor. When a modified line is written back to the memory, the stale bit is cleared.

When the memory processor performs a read, it checks the stale bit. If the stale bit is clear, it accesses the data from its cache or from the main memory. If the stale bit is set, it has to retrieve the correct version from the main processor. To do that, it simply asks for the main processor to write back the line. The write-back clears the stale bit, and the memory processor can snoop the data being written back for its use without accessing the main memory.

The modification of the memory controller includes implementing queue 1, queue 3, and the comparison logic. In addition, a small directory is added to the cache coherence protocol, with one state bit per memory block. While this introduces extra hardware support, it is simpler than a typical directory-based cache coherence protocol for multiprocessors.

3 Active Prefetching

In this section, we present our helper thread construction algorithm (Section 3.1) and the synchronization mechanisms for CMP (Section 3.2) and intelligent memory (Section 3.3). Section 3.4 describes the synchronization mechanism for regular loop nests.

3.1 Constructing a Helper Thread

The goal of the helper thread construction algorithm is to make the helper thread as short as possible but still contain the instructions necessary to generate correct prefetches to target loads/stores that will likely miss in the L2 cache. To achieve that, we first extract *prefetching sections*, the code sections that will be targeted for prefetching. These are chosen as regions of code that have a high concentration of L2 cache misses. These code sections can be identified by profiling or by using a static code partitioning technique such as the one described in [13] and [14]. In the static code partitioning, the code is partitioned into sections that have a homogeneous memory and computing behavior. The resulting section is a maximal loop nest where each nesting level has only one loop and possibly several statements. Such a loop nest may span several subroutine levels. By allowing only one loop per level, we increase the chances of finding a homogeneous memory behavior.

The prefetching sections typically consists of several millions of dynamic instructions. Reads and writes that likely miss in a prefetching section then will be converted into prefetch instructions in the helper thread. The helper thread is spawned only once at the beginning of the application. It synchronizes with the application thread at the start of each prefetching section and after a multiple of iterations inside a prefetching section. Using a large prefetching section, one-time thread spawning, and infrequent synchronization allows our scheme to work well for loosely coupled systems. We do not have any thread-spawn latency that occurs in the helper thread prefetching for SMT processors due to context initialization [3], [4], [5].

The following is the algorithm we use to extract the prefetching helper thread for the identified loops:

1. An inline function calls in the loop.
2. Identify target reads and writes to array elements or structures' fields that likely miss in the L2 cache.
3. Starting from these read and writes, identify address computations by following the use-def chain while preserving the original control flow.
4. Privatize the locations that are written in the address chain:
   - If the location is an array element, substitute the reads/writes of the array in the address chain with reads/writes to a new privatized array that belongs to the helper thread.
   - Privatize scalar variables that are assigned in the address chain with a new temporary variable.
5. Replace the target reads and writes with prefetch instructions that access the same addresses.
6. Remove unnecessary computations, branches, and redundant prefetch instructions. Combine the two branches of the if statement if they do not affect the original control flow of the address computation.

Our helper thread construction algorithm privatizes the variables that are written in the address computation chain. Unnecessary writes that are not involved in the address computation are also removed along with the unnecessary computation, branches, etc. Consequently, the helper thread does not have any writes to shared data.

Fig. 3a shows the original `refresh_potential()` subroutine in *mcf* of Spec2000, which is selected as a prefetching section. In the constructed helper thread (Fig. 3b), `node` and `tmp` are privatized to allow the helper thread to compute independently, the original `if` statement on line `06` is combined in the helper thread, and reads/writes are converted into prefetch instructions (lines `06-08` in the helper thread). In addition, unnecessary computation is removed, such as accesses to field members that likely fall within the same cache line (`node->potential` and `node->basic_arc->cost` are in the same cache line as `node`). Note that it is possible that the heap objects accessed in one iteration (`node`, `node->pred`, and `node->basic_arc`) happen to be in the same cache line. If that is the case, only the first prefetch will go to the memory, while the prefetches to other heap objects will either be queued in the MSHRs in the CMP or be filtered out by the Filter module in the intelligent memory.

3.2 Synchronization for CMP

As we will show in Section 5, the speed difference between the helper and application threads greatly affects the prefetching performance. When the helper thread runs too far ahead of the application thread, prefetched lines may pollute the cache and may be evicted from it before they are accessed by the application thread. On the other hand, if the helper thread runs behind the application thread, it no longer fetches useful data but instead causes cache pollution and degrades the performance of the application thread. In most cases, since we remove many instructions from the original code to construct the helper thread, it usually runs ahead of the application thread. However, in some cases, the helper thread may suffer long latency

```
01  long refresh_potential(network_t * net) {
02    node_t * node, * tmp;
03    ··· // some computation
04    while (node != root) {
05      while (node) {
06        if(node->orientation == UP) {
07          node->potential = node->basic_arc->cost
08                  + node->pred->potential;
09        } else {
10          node->potential = node->pred->potential
11                  - node->basic_arc->cost;
12          checksum++;
13        }
14        tmp = node;
15        node = node->child;
16      }
17      node = tmp;
18      while (node->pred) {
19        tmp = node->sibling;
20        if (tmp) {
21          node = tmp;
22          break;
23        } else {
24          node = node->pred;
25        }
26      }
27    }
28  }
```

(a)

```
01  long refresh_potential(network_t * net) {
02    node_t * node, * tmp;
03
04    while (node != root) {
05      while (node) {
06        pref(node);
07        pref(node->pred);
08        pref(node->basic_arc);
09        tmp = node
10        node = node->child;
11      }
12      node = tmp;
13      while (node->pred) {
14        tmp = node->sibling;
15        if (tmp) {
16          node = tmp;
17          break;
18        } else {
19          node = node->pred;
20        }
21      }
22    }
23  }
```

(b)

Fig. 3. Constructing a prefetching helper thread. (a) The original application thread. (b) The constructed helper thread. The prefetching section is the `refresh_potential()` subroutine in *mcf*.

events (e.g., cache misses or page faults). In this case, when the helper thread resumes, it may slow down the application thread by polluting the cache and be unable to catch up to the application thread within a reasonable time period.

We present a new synchronization mechanism that targets both problems without requiring any hardware modifications. It prevents the helper thread from running too far ahead by controlling the maximum distance between the helper thread and the application thread. It also prevents the helper thread from running behind the application thread by allowing it to catch up quickly to the application thread. To achieve both goals, we use lightweight general semaphores. In a semaphore, the $P(s)$ operation is a *wait* operation that prevents the thread that executes it from going past the synchronization until $s > 0$, after which it decrements s by 1. $V(s)$ is a *signal* operation that increments s by 1. We assume that the P and V operations are atomic.

Fig. 4a shows the original prefetching section of the application thread, augmented with synchronization that controls the helper thread's execution distance, while Fig. 4b shows the helper thread code with the synchronization code inserted. Two semaphores are used: `sem_helper_start` controls when the helper thread should start its prefetching code, while `sem_loop_sync` controls the execution distance of the helper thread. The helper thread is spawned at the beginning of the application. When it executes `sema_P(&sem_helper_start)` on line `04` in Fig. 4b, it busy waits there until the application thread calls `sema_V(&sem_helper_start)` on line `05` in Fig. 4a. The helper thread will synchronize

```
01  long refresh_potential(network_t * net) {
02    node_t * node, * tmp;
03    int syncInterval = LOOP_SYNC_INTERVAL;
04    sema_init(&sem_loop_sync, MAX_DIST);
05    sema_V(&sem_helper_start);
06    ... // some computation
07    while (node != root) {
08      while (node) {
09        ...
10      }
11      ...
12      if (!(--syncInterval)) {
13        syncInterval = LOOP_SYNC_INTERVAL;
14        node_main = node;
15        sema_V(&sem_loop_sync);
16      }
17    }
18  }
```

(a)

```
01  long refresh_potential(network_t * net) {
02    node_t * node, * tmp;
03    int syncInterval = LOOP_SYNC_INTERVAL;
04    sema_P(&sem_helper_start);
05
06    while (node != root) {
07      while (node) {
08        pref(node);
09        pref(node->pred);
10        pref(node->basic_arc);
11        ...
12      }
13      ...
14      if (!(--syncInterval)) {
15        syncInterval = LOOP_SYNC_INTERVAL;
16        if (sem_loop_sync.count > MAX_DIST) {
17          sema_init(& sem_loop_sync, MAX_DIST);
18          node = node_main;  // catch up
19        } else {
20          sema_P(&sem_loop_sync);
21        }
22      }
23    }
24  }
```

(b)

Fig. 4. Synchronizing the main and helper threads in a CMP. (a) The main thread. (b) The helper thread.

```
01  long refresh_potential(network_t * net) {
02    node_t * node, * tmp;
03
04    SYNC_CLEAR_MAIN();
05    SYNC_SIGNAL();
06    ... // some computation
07    while (node != root) {
08        ...
09      SYNC_MAIN_INC();
10      SYNC_SET_ADDR(node);
11    }
12  }
```

(a)

```
01  long refresh_potential(network_t * net) {
02    node_t * node, * tmp;
03
04    SYNC_CLEAR_HELPER();
05    SYNC_WAIT();
06    while (node != root) {
07      ...
08      SYNC_HELPER_INC();
09      while (SYNC_TEST&SET(MAX_DIST,&node));
10    }
11  }
```

(b)

Fig. 5. Synchronizing the main and helper threads in an intelligent memory using special synchronization registers. (a) The application thread. (b) The helper thread.

every time after it executes LOOP_SYNC_INTERVAL iterations in the loop. Since sem_loop_sync is initialized to MAX_DIST by the application thread on line 04, the helper thread is allowed to run ahead by MAX_DIST * LOOP_SYNC_INTERVAL.

When the helper thread synchronizes, it first restores the value of syncInterval (line 15). Then, it checks the value of the sem_loop_sync semaphore. A value larger than MAX_DIST indicates that the helper thread runs behind the application thread since there are multiple signals that it has not consumed. In such a case, the helper thread skips the iterations that the application thread has already gone through by resetting the sem_loop_sync to MAX_DIST (line 17) and setting its current traversal node to the application thread's current node node_main (line 18). If it is not running behind, it simply consumes a signal and continues or waits until a signal is available (line 20).

3.3 Synchronization for Intelligent Memory

The synchronization mechanism for CMP in Section 3.2 cannot be directly applied to an intelligent memory. For it to work correctly, the semaphore variables must be placed in an uncacheable region in memory so that the updates by the application thread and the helper thread are seen by each other. If the semaphores were cacheable, the update by the helper thread would not be propagated to the main processor since our simple intelligent memory cache coherence mechanism in Section 2.3 assumes that the memory processor does not write to shared data. On the other hand, an uncacheable shared variable that is written only by the application thread and read only by the helper thread would suffice for synchronization between the two threads. However, when the semaphores or the shared synchronization variable is not cached, each thread's access to them incurs a high latency, noticeably degrading the prefetching performance. Thus, we propose three special registers in the memory processor to provide correct and efficient synchronization (Fig. 2b).

The first two registers (main_count and helper_count) store the number of iterations that the application and helper threads have executed, reflecting their respective progress. The distance between the two threads is obtained by subtracting these two register values. The distance is then used to control how far ahead the helper thread is allowed to run. Another register (sync_addr) stores the address of the synchronization variable that is transferred from the main processor and provides the mechanism for the helper thread to catch up to the application thread when it is running behind. We define APIs for synchronization using these special registers.

Figs. 5a and 5b show the application's and the helper thread's code section using the synchronization APIs, respectively. The helper thread is spawned at the beginning of the application and busy waits when it executes SYNC_WAIT() on line 05 in Fig. 5b until the application thread calls SYNC_SIGNAL() on line 05 in Fig. 5a. SYNC_CLEAR_MAIN() and SYNC_CLEAR_HELPER() initialize main_count and helper_count to 0. At the end of each iteration, the application (or helper) thread increments the main_count (or helper_count) register by calling SYNC_MAIN_INC() (or SYNC_HELPER_INC()). In addition, the application thread calls SYNC_SET_ADDR(node) to set the sync_addr register to the address of the node on which it is currently working. Finally, the helper thread executes while (SYNC_TEST&SET(MAX_DIST,&node)) on line 09 in Fig. 5b. If the function returns one, indicating that the helper thread is ahead by MAX_DIST iterations, it will busy wait in the function until the application thread progresses. When it is not yet too far ahead, it will exit the function and execute more iterations. However, when it lags behind, it catches up to the application thread by copying the application thread's current node to its own node variable.

3.4 Regular Loop Nests

Our synchronization mechanism can also be applied to some regular loop nests found in scientific/numerical applications. Fig. 6 shows an example where the number of iterations is used to prevent the helper thread from running too far ahead for intelligent memory. The number of iterations is used to prevent the helper thread from running too far ahead for intelligent memory. The case of CMP is similar to the case of intelligent memory, and it uses semaphores. The main_count register contains the number of iterations that the application thread has performed. In the helper thread, the helper_count register contains the number of iterations that the helper thread has performed. The difference between main_count and helper_count indicates how far the helper thread is behind or ahead. The iteration index variable i is stored in sync_addr, allowing the helper thread to copy its value and catch up when it discovers that it is running behind.

3.5 Synchronization with Multiple Live-In Variables

There may be more complex applications to which our *catch-up* mechanism cannot be directly applied. For example, consider a pointer-chasing loop that requires multiple

```
01  void smvp(int nodes, double ***A, int *Acol, int *Aindex
02            double **v, double **w) {
03
04    int i, Anext, Alast, col;
05    double sum0, sum1, sum2;
05
06    SYNC_CLEAR_MAIN();
07    SYNC_SIGNAL();
08
09    for (i=0; i<nodes; i++) {
10      Anext = Aindex[i];
11      Alast = Aindex[i+1];
12      ...
13      Anext++;
14      while (Anext < Alast) {
15        sum0 += A[Anext][0][0]*v[col][0]
16          +A[Anext][0][1]*v[col][1] + A[Anext][0][2]*v[col][2];
17        sum1 += A[Anext][1][0]*v[col][0]
18          +A[Anext][1][1]*v[col][1] + A[Anext][1][2]*v[col][2];
19        sum2 += A[Anext][2][0]*v[col][0]
20          +A[Anext][2][1]*v[col][1] + A[Anext][2][2]*v[col][2];
21        ...
22        Anext++;
23      }
24      ...
25      SYNC_MAIN_INC();
26      SYNC_SET_ADDR(i);
27    }
28  }
```

(a)

```
01  void smvp(int nodes, double ***A,
02            int *Acol, int *Aindex
03            double **v, double **w) {
04
05    SYNC_CLEAR_HELPER();
06    SYNC_WAIT();
07
08    for (i=0; i<nodes; i++) {
09      Anext = Aindex[i];
10      Alast = Aindex[i+1];
11      Anext++;
12      while (Anext < Alast) {
13        col = Acol[Anext];
14        prefetch(&A[Anext][0][0]);
15        prefetch(&A[Anext][1][0]);
16        prefetch(&A[Anext][2][0]);
17        prefetch(&v[col][0]);
18        Anext++;
19      }
20      SYNC_HELPER_INC();
21      while (SYNC_TEST&SET(MAX_DIST,&i));
22    }
23  }
```

(b)

Fig. 6. Synchronizing the application and helper threads in the context of a regular loop nest. (a) The application thread. (b) The helper thread. The prefetching section is from *equake* of Spec2000.

live-in variables to compute the value of the pointer. With our current catch-up mechanism, the helper thread cannot compute the correct value of the next pointer after receiving the current pointer value from the application thread because it does not have up-to-date live-in values for computing the next pointer. To solve this problem, the application thread has to pass all the live-in variables used in the address computation to the helper thread for every synchronization point; the more such variables, the more overhead incurred in the *catch-up* mechanism. Particularly, additional hardware cost can be charged to the intelligent memory architecture because it needs more synchronization registers for the live-in variables. However, we do not encounter such a complicated case in the prefetching sections of the benchmarks that we evaluate. One live-in register was sufficient for them.

4 Evaluation Environment

Applications. To evaluate our helper threading strategies, we use nine memory-intensive applications, shown in Table 1. We only choose SPEC2K applications with many cache misses and supplement them with cache-miss-intensive applications from Olden and NAS.

The first four columns show the name, source, description, and input set of the applications. The fifth column shows the number of target loops selected as the prefetching sections. The last column shows the percentage of the original execution time covered by the target loops.

Each benchmark application is profiled with the same input that is used in the evaluation. Profiling is done with the SESC execution-driven simulator [15]. Based on the profiling information, we identify memory read and write instructions that cause heavy cache miss stalls. Then, the loops that contain the identified read/write instructions are selected as our targets for prefetching.

We apply the helper thread construction algorithm by hand. However, this can be automatically done by a compiler. As mentioned before, prefetching sections are determined based on the profiling information. This profile run can be done automatically or manually, provided that a performance monitoring unit is available in the target machine. After automatically or manually identifying the target sections, the helper thread code and the main thread code can be automatically extracted by a compiler from the target sections using our helper thread construction algorithm. The major compiler techniques are based on program slicing [16], privatization [17], and control and data dependence analyses [18].

Simulation environment. The evaluation is performed using a cycle-accurate execution-driven simulator that

TABLE 1
Applications Used

Appl	Suite	Problem	Input	# of target loops	% exec. time
bzip2	SpecInt2K	Compression and Decompression	Reference	1	29.51%
cg	NAS	Conjugate gradient	Class W	1	91.63%
em3d	Olden	Electro-magnetic wave propagation in 3D	200,000	1	47.55%
equake	SpecFP2K	Seismic wave propagation simulation	Reference	1	65.16%
mcf	SpecInt2K	Combinatorial optimization	Subset of Reference	1	78.03%
mg	NAS	Multigrid solver	Class A	1	47.56%
mst	Olden	Finding minimum spanning tree	1,200 nodes	1	69.11%
parser	SpecInt2K	Word processing	Subset of Reference	1	19.48%
swim	SpecFP2K	Shallow water modeling	Train	3	99.21%

TABLE 2
Parameters of the Simulated Architecture

MAIN PROCESSOR	6-issue dynamic. 3.2 GHz. FUs: Int-ALU/Mul/Div: 2/1/1, Fp-Add/Mul/Div: 2/1/1, Ld/St: 1/1 Branch penalty: 12 cycles L1 data: write-through, 8 KB, 4 way, 64B line, 2-cycle hit RT, 16 outstanding misses L2 data: write-back, 1MB, 8 way, 128B line, 22-cycle hit RT, 32 outstanding misses RT memory latency: 381 cycles (row miss), 333 cycles (row hit) Memory bus: split-transaction, 8B, 800 MHz, 6.4 GB/sec peak
MEMORY PROCESSOR	2-issue dynamic. 800 MHz. FUs: Int-ALU/Mul/Div: 2/1/0, no FP units, Ld/St: 1/1, Pending ld/st: 16/16 Branch penalty: 6 cycles L1 data: write-back, 64 KB, 2 way, 64B line, 4-cycle hit RT, 16 outstanding misses **In DIMM** RT mem latency: 155 cycles (row miss), 107 cycles (row hit) Latency of a prefetch request to reach DRAM: 71 cycles **In DRAM** RT mem latency: 95 cycles (row miss), 47 cycles (row hit) Internal DRAM data bus: 32B wide, 800 MHz, 25.8 GB/sec peak
CONFIGURATIONS	**CMP:** 2 main procs, private L1 caches, shared L2 cache and lower memory hierarchy **Intelligent Memory:** 1 main proc + 1 memory proc, shared main memory
PREFETCHING	Filter module: 32 entries, FIFO Main proc prefetching: hardware, 8-stream, sequential

*Latencies correspond to contention-free conditions. *RT* stands for round-trip *from the processor*. All cycles are 3.2 GHz

models dynamic superscalar processor cores, CMP, and intelligent memory [15]. The MIPS instruction set is used for simulation. We run the benchmark applications in Table 1 to completion. We model a PC architecture with a simple memory processor integrated in either a DRAM chip or the DIMM, following the microarchitecture described in Section 2. Table 2 shows the parameters used for each component of the architecture. All cycles are 3.2-GHz cycles. We model a uniprogrammed environment where the application and the helper thread execute concurrently without context switches. We model contention in the system between the application and helper threads on shared resources such as the L2 cache and the system bus in the CMP configuration plus the memory controller and the DRAM resources (banks and row buffers) in all configurations. Especially, for the L2 cache, a single L2 tag array and its contention are simulated.

For the synchronization overhead of intelligent memory, we used the delays below for our simulation. Both latencies are on top of the time needed for the main processor to access the L2 cache:

- *Latency for transferring an address from the application to a synchronization register in the intelligent memory (SYNC_SET_ADDR).* Bus data delay (64)+ Memory controller delay $(4) = 68$ cycles. This figure assumes that an entire cache block containing the address is written back over the data bus.
- *Latency for transferring a signal from the application to the intelligent memory (SYNC_MAIN_INC).* Bus command delay (3) + Memory controller delay $(4) = 7$ cycles.

Synchronization APIs. The new synchronization APIs (in Section 3.3) are only needed by the intelligent memory architecture and not by CMP. For CMP, the communication is achieved through semaphore variables in shared memory: communication latency is equivalent to cache hit/miss latency to those variables.

Main processor's hardware prefetching. The main processor optionally includes a hardware prefetcher at the L1 cache level that can prefetch eight streams of strided accesses to consecutive cache lines. The prefetched data is placed in the L1 cache. It uses the double delta scheme, in which it waits until it identifies three consecutive lines being accessed before it prefetches six lines in the identified stream. The prefetcher is somewhat similar to stream buffers [19], but the prefetched lines are placed in the L1 cache.

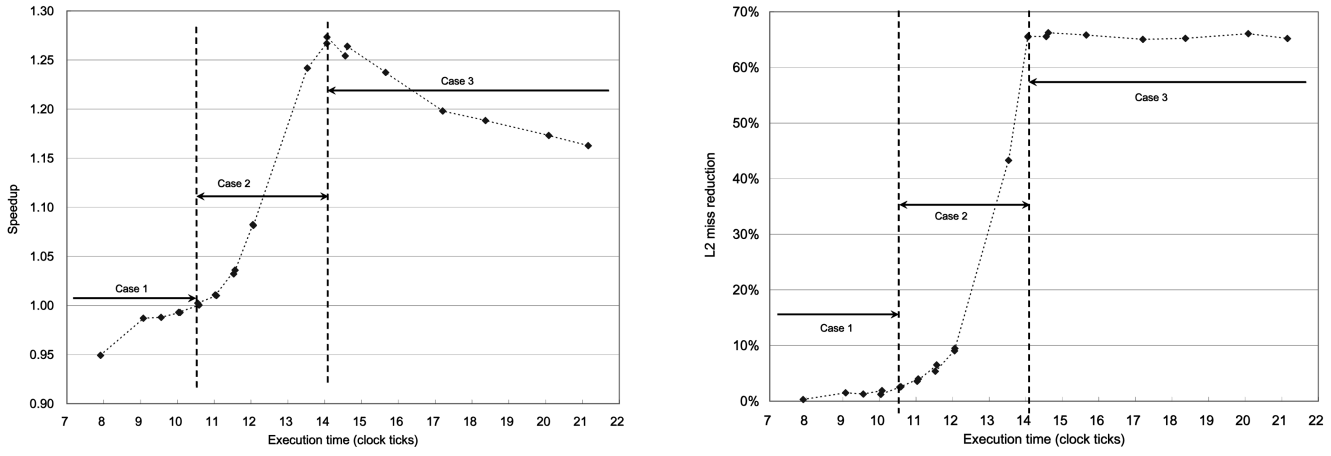
5 Performance Characterization

In this section, we evaluate and analyze the effects of nonmemory operations on the prefetching performance. A nonmemory operation in this section refers to the operation that is not involved in any memory access and its address computation. Intuitively, the more nonmemory operations a prefetching section has, the shorter the helper thread can be, allowing the helper thread to run sufficiently ahead of the application thread. To obtain a more precise idea of the extent and nature of this problem, we evaluate the speedup resulting from prefetching when the amount of nonmemory operations is varied.

To achieve that, we create a synthetic benchmark that performs a linked-list traversal on a fixed number of nodes and vary the amount of nontraversal operations in each iteration. Figs. 7a and 7b show the speedup and the fraction of eliminated L2 cache misses in the application thread due to the helper thread prefetching, respectively. The x-axis represents the execution time of the target loops in the application thread for the entire traversal, including the synchronization overhead. When nonmemory operations are added into the code, the execution time increases along the x-axis. The figure shows that there are three cases. The helper thread runs on the memory processor in DRAM in the intelligent memory configuration. The case of CMP has the same trend. Even though real programs are not written in the way this synthetic microbenchmark is written, this benchmark explains well the nature of the problem.

In **Case 1**, when there is a small fraction of L2 cache misses eliminated, there is a slight slowdown in the application. This is because the helper thread code is not much shorter than the application code. When this helper thread runs on a helper processor, it is either slower or not much faster than the application thread that runs on the main processor. As a result, instead of eliminating L2 cache misses, it causes extra prefetching traffic and pollutes the L2 cache, even though much of the effect is mitigated by the ability of the synchronization to help the helper thread catch up.

In **Case 2**, the helper thread is sufficiently faster than the main thread, resulting in timely and effective prefetches. The actual speedup depends on the speed of the helper



(a) (b)

Fig. 7. The prefetching performance for a fixed number of nodes in a linked-list traversal. (a) Speedup. (b) The fraction of L2 cache misses that are eliminated. The unit of execution time is one million clock ticks.

thread relative to the application thread. The speedup saturates at some point (the border of Case 2 and Case 3) after the helper thread has prefetched all the L2 cache misses that it is able to prefetch. Assuming that each L2 cache miss contributes to the memory stall time equally, the execution time of the application after prefetching (T_{new}) can be modeled as

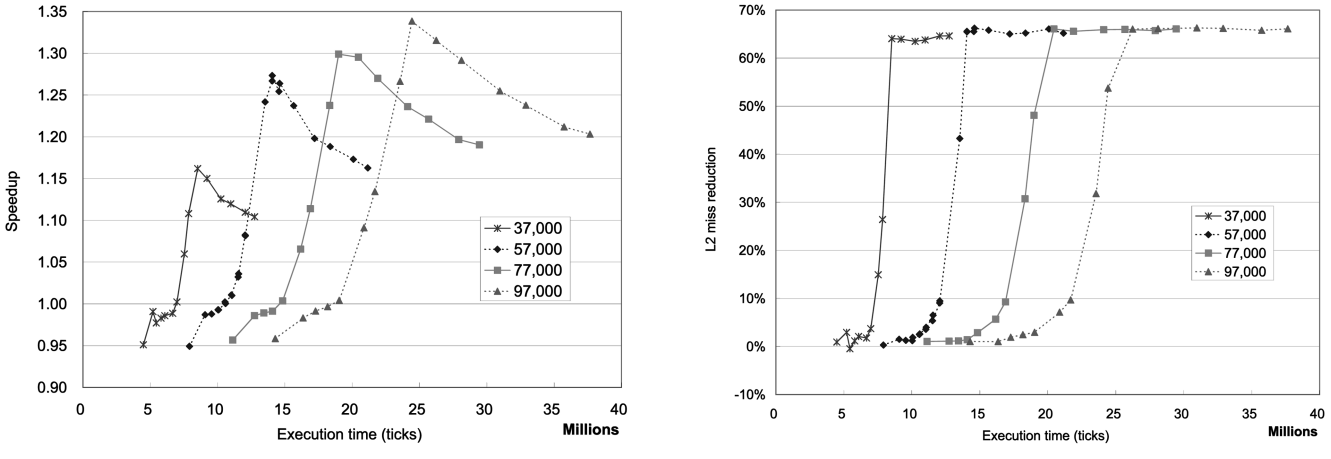
$$T_{new} = T_{orig} - T_{orig\_mem} \cdot \left(1 - \frac{Miss_{new}}{Miss_{orig}}\right),$$

where T_{orig} denotes the original execution time of the loop running on the main processor, and T_{orig_mem} denotes the memory stall time portion of T_{orig} due to L2 cache misses in the loop. $Miss_{orig}$ and $Miss_{new}$ are the number of L2 misses in the loop before and after prefetching, respectively. Note that the fraction of eliminated L2 cache misses in Fig. 7b corresponds to $1 - \frac{Miss_{new}}{Miss_{orig}}$ in the formula. Since T_{orig_mem} is constant in the experiment, the formula states that the application's speedup by prefetching is proportional to the fraction of the eliminated cache misses that is experienced by the main thread. This explains why the shape of the figure in Case 2 in Fig. 7a follows that in Fig. 7b. As the difference of the execution time between the main thread and the helper thread increases, the reduction of L2 cache misses becomes larger (smaller $Miss_{new}$), resulting in better speedups (smaller T_{new}).

In **Case 3**, the helper thread cannot eliminate any more L2 cache misses (Fig. 7b). However, the speedup in Fig. 7a decreases since the nonmemory-stall fraction of the execution time also grows, hence reducing the speedup.

Fig. 8, similar to Fig. 7, shows the speedup and L2 miss reduction due to helper thread prefetching. However, we vary the number of nodes in the linked list, and as a result, the number of L2 cache misses in the original code is also varied. We see that the overall trends are the same for different numbers of nodes in the linked list (i.e., for different T_{orig_mem}s).

Overall, we see that the relative execution time of the helper thread to the application thread is a critical factor that determines the prefetching effectiveness of the helper thread.



(a) (b)

Fig. 8. The prefetching performance when the number of nodes in the linked-list traversal is varied. (a) Speedup. (b) The fraction of L2 cache misses that are eliminated.

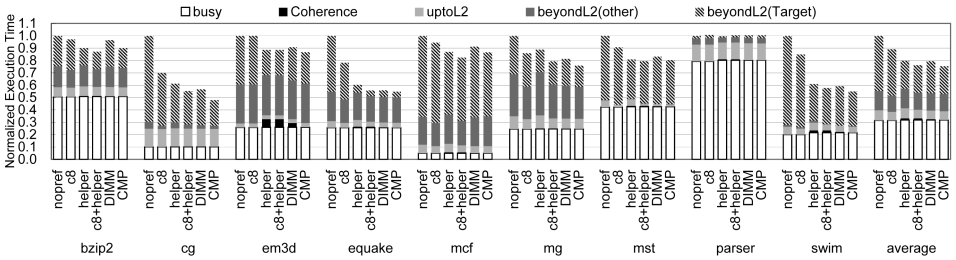


Fig. 9. Execution time of the applications with different prefetching schemes and different architectures.

6 Evaluation with Simulation

6.1 Prefetching Performance

Fig. 9 compares the execution time of the entire applications (not just the targeted loops) for the following cases:

1. There is no prefetching (*nopref*).
2. There is only hardware-processor-side prefetching (*c8*).
3. Our helper thread prefetching is running on a memory processor in the DRAM chip (*helper*).
4. The hardware prefetching is backed up by the helper thread prefetching in the memory processor in DRAM (*c8+helper*).
5. The hardware prefetching is backed up by the helper thread prefetching in the memory processor in the DIMM (*DIMM*).
6. The hardware prefetching is backed up by the helper thread prefetching in the memory processor in a separate CMP processor (*CMP*).

For all applications and their average, the bars are normalized to *nopref*. Each bar shows the memory stall time due to L2 cache misses in the target loops (*beyondL2(Target)*) or other parts of the application (*beyondL2(Other)*), memory stall time due to L1 or L2 cache hits (*uptoL2*), hardware cache coherence overhead (*coherence*), and the remaining time (*busy*).

On the average, *beyondL2(Target)* is the most significant component of the processor's stall time in *nopref*, accounting for 61 percent of the total execution time. It is also significantly larger than *beyondL2(Other)*, indicating that the L2 cache misses are quite concentrated in the targeted loops. *Em3d* and *swim* have noticeable coherence traffic, but it is not zero for other applications. Since it is very small for other applications, it cannot be seen in Fig. 9.

C8 performs relatively well on the applications with sequential access patterns such as *cg*, *equake*, *mg*, and *swim*. However, it is relatively ineffective for applications that have mostly irregular access patterns such as *bzip2*, *em3d*, *mcf*, *mst*, and *parser*. On the average, *c8* reduces the execution time by 11 percent.

Helper reduces the execution time significantly for almost all applications except for parser. This is partly due to the small memory stall time of the target loops (*beyondL2(Target)*) in parser. In addition, *Helper* can reduce only the *beyondL2(Target)* time because it only performs prefetches in the target loops, whereas *c8* prefetches for all parts of the application and hides the L1 miss latency. Because of that, *c8* outperforms *helper* in *mg*.

C8+helper performs better than other cases on the average, except for *CMP*. It removes more than 50 percent of the *beyondL2(Target)* time and reduces the total execution time by 24 percent, resulting in a speedup of 1.31. We can see that in most cases, when both prefetching schemes are combined, *c8+helper* achieves better performance than either one of them can. This is because 1) the helper thread prefetching is able to target irregular access patterns that are difficult to prefetch using a conventional prefetcher, 2) conventional prefetching contributes improvements from outside the targeted loops, and 3) conventional prefetching provides additional L1 miss latency hiding.

Finally, *DIMM* and *CMP* also achieve very good speedups. Despite the memory processor suffering from higher memory access latency in the DIMM, *DIMM* achieves an average speedup of 1.26. In *CMP*, the very high memory access latency in the CMP processor is partially offset by the speed of the processor. As a result, *CMP* is able to deliver an average speedup of 1.33. This is a significant result considering that *CMP* does not require any hardware modifications. Therefore, CMP is an attractive architecture to run the helper thread.

6.2 Prefetching Effectiveness

Fig. 10 gives further insights into the prefetching effectiveness of our schemes. It shows the number of prefetch requests normalized to the original number of L2 misses in *nopref*. The requests are classified into 1) those that completely/partially eliminate a cache miss (*Useful*) and

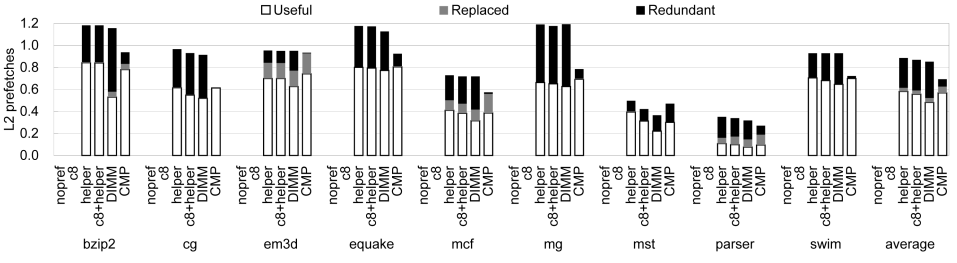


Fig. 10. Breakdown of the L2 misses and lines prefetched by the helper thread.

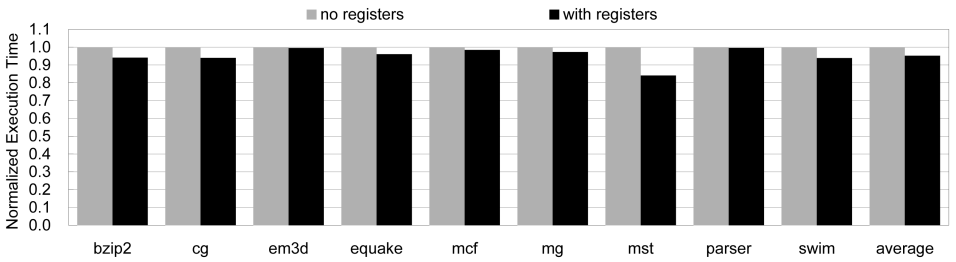


Fig. 11. The performance comparison of the cases with and without special synchronization registers.

2) those that are useless because either they are replaced from the L2 cache before they are accessed (*Replaced*) or they are dropped on their arrival at the L2 cache because the line is already in the L2 cache (*Redundant*). The figure shows that on the average, almost 60 percent of the L2 cache misses are prefetched by the *helper*. However, *helper* also generates 27 percent useless prefetches. *c8+helper* achieves lower *Useful* prefetching but higher performance due to the contribution from the conventional prefetching. Finally, *CMP* achieves much lower useless prefetching, especially *Redundant*. This is because in the CMP, a prefetch request is issued only for a line that is not already in the L2 cache. Overall, *CMP* is attractive due to its high *Useful* prefetching and, at the same time, low *Replaced* and *Redundant* prefetching.

6.3 Impact of Synchronization Hardware

Fig. 11 compares the execution time of the application thread using *c8+helper* prefetching when the synchronization uses uncacheable semaphores (*no registers*) versus when it uses synchronization registers (*with registers*), normalized to the semaphore case. The figure shows that the synchronization registers reduce the execution time by 5 percent on the average. The reduction is quite significant in *bzip2*, *cg*, *mst*, and *swim*. This is because a low-overhead synchronization allows us to control the distance between the helper thread and the application more finely, by synchronizing at the end of each iteration. With uncached semaphores, the large synchronization overhead forces us to use a larger `LOOP_SYNC_INTERVAL` value to reduce the synchronization frequency to every several iterations of the target loop (see Section 3). For *em3d*, its `LOOP_SYNC_INTERVAL` is already an optimal value when the semaphores are used. Thus, there is no performance variation when the synchronization registers are used.

6.4 Main Memory Bus Utilization

Fig. 12 shows the main memory bus utilization of the different schemes for the target loops. The additional bus utilization is divided into one that is caused by the prefetching requests from the helper thread and one that is caused by the reduction in the overall execution time due to prefetching. Overall, the figure shows that the majority of the increase in bus utilization is due to the reduction in execution time, with only 10 percent-15 percent extra bus utilization coming from extra prefetching requests. This is quite tolerable. Even in the worst case, the extra utilization due to prefetching traffic is only 29 percent in *c8+helper* for *bzip2*.

6.5 Synchronization Intervals

Our synchronization is not performed in every iteration. Instead, it is performed every LOOP_SYNC_INTERVAL iterations. This reduces the synchronization overhead significantly. The helper thread is allowed to run ahead by MAX_DIST*LOOP_SYNC_INTERVAL iterations, whose values are shown in the following table.

bzip2	150	cg	20	em3d	120
equake	30	mcf	15	mg	20
mst	100	parser	50	swim	3

If LOOP_SYNC_INTERVAL*MAX_DIST is 120 and LOOP_SYNC_INTERVAL is 40, then MAX_DIST is 3. Larger LOOP_SYNC_INTERVAL values have lower synchronization overheads and produce better performance. MAX_DIST must be at least 1 to detect whether the helper thread lags behind. We found the value of LOOP_SYNC_INTERVAL*MAX_DIST for each application through experiments. Each application has a different value of LOOP_SYNC_INTERVAL*MAX_DIST.

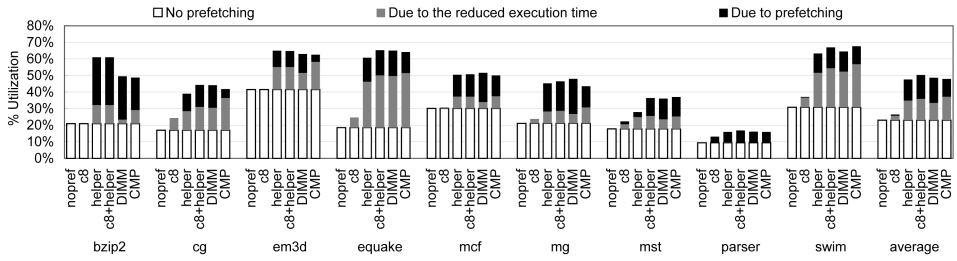


Fig. 12. Main memory bus utilization for the target loops.

TABLE 3
Specification of the Real CMP System

CPU: Intel Core Duo (Yonah) 1.66 GHz, dual core
L1 instruction cache: 2 X 32 KB, one per core, 8-way set-associative, 64B line size
L1 data cache: 2 X 32 KB, one per core, 8-way set-associative, 64B line size
L2 unified cache: 1 X 2 MB, shared between cores, 8-way set-associative, 64B line size
Memory bus: 667 MHz, 5.3 GB/sec peak
Total memory: 2 GB
Operating system: Windows XP professional
Compilers: Intel C++ and FORTRAN compiler v9.0 with -O2

6.6 Code Size and Memory Size

Since the helper thread was extracted for just one function for all the applications except *swim* (*swim* has three target loops), the helper thread's code size is very small compared to the application.

The extra dynamic memory (heap and stack) created by the helper thread is negligible because it just reads the dynamic memory location created by the application thread for prefetching. In addition, the helper thread does private writes only for address calculation, which, in most cases, are to scalar variables. This means that the dynamic memory size for the helper thread is much smaller than the application thread and equal to the application thread in the worst case.

7 Evaluation on a System with Intel Core Duo Processor

To evaluate our helper thread prefetching strategies on a real machine, we use a system with a single Intel Core Duo processor. Intel Core Duo is a dual-core CMP with a shared L2 cache. The specification of the machine is shown in Table 3. Note that this machine has a 2-Mbyte L2 cache shared between the two cores. It is much bigger than the shared 1-Mbyte L2 cache used in our evaluation with simulation in the previous section.

The Intel Core Duo processor provides an automatic hardware prefetching mechanism to the L2 cache in addition to software prefetch instructions [20]. Note that the CMP configuration in our evaluation with simulation has a stride prefetcher that prefetches data up to the L1 cache. The hardware prefetcher can handle multiple streams in the forward or backward directions with regular stride patterns and prefetches data into the L2 cache. Two successive cache misses in the L2 cache trigger the hardware prefetcher when the two cache misses satisfy the condition that the stride of the cache misses is less than the trigger distance of the hardware prefetcher. The hardware prefetcher in each core prefetches data independently, and it attempts to prefetch two cache lines ahead of the prefetch stream [20]. Since we cannot turn off this hardware prefetcher in the Intel Core Duo processor, our base case for comparison includes the performance improvement by the hardware prefetcher.

TABLE 4
Applications and Inputs Used on Intel Core Duo

Appl	Input	% execution time
bzip2	Reference	15.75%
cg	Class B	94.77%
em3d	600,000	57.18%
equake	Reference	59.59%
mcf	Reference	63.80%
mg	Class B	46.07%
mst	10,000 nodes	69.38%
parser	Reference	26.56%
swim	Reference	99.52%

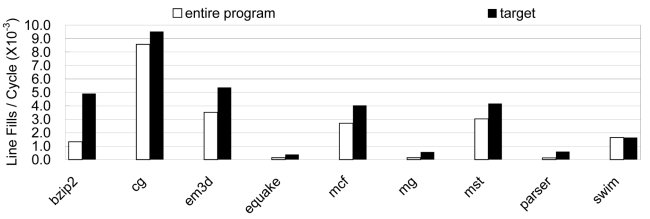


Fig. 13. *L2 cache line fills per cycle* of the entire application and target loop(s) on the Intel Core Duo processor.

We use the same applications and target loops described in Section 4. However, the size of the input is different. Table 4 shows input sets, and the percentage of the original execution time covered by the target loops on the Intel Core Duo. Executable images are generated using Intel C++ and Fortran compilers [21], [22] with -O2 option because it is most commonly used by the application developers. Software prefetch instructions [20] are used in the helper thread.

Intuitively, we can approximate the impact of L2 cache misses to the execution time with the value of *L2 cache line fills per cycle*, where the number of L2 cache line fills does not include the line fills due to the hardware prefetcher [23]. A high *L2 cache line fills per cycle* value tells us that the application suffers L2 cache misses too often. Thus, our helper thread prefetching will be effective for the applications with a high *L2 cache line fills per cycle* value. Fig. 13 shows the values for the entire application and the target loop(s). To measure the value of *L2 cache line fills per cycle*, we used Intel VTune Performance Analyzer 8.1 [23]. We expect that our helper thread prefetching will not improve the performance of *equake*, *mg*, and *parser* because their values of *L2 cache line fills per cycle* are relatively low.

Fig. 14 compares the execution time of each application with and without helper thread prefetching. The bars labeled *base* show the execution time of each application

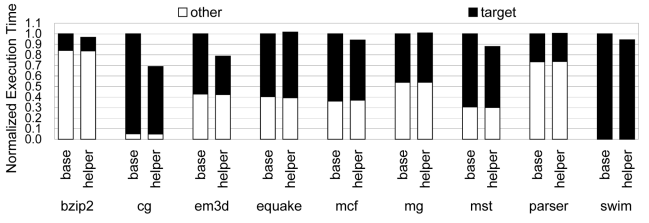


Fig. 14. Execution time of the applications and target loops on Intel Core Duo.

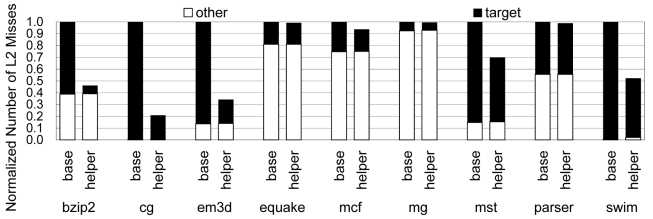


Fig. 15. The number of L2 cache misses of the applications and target loops on Intel Core Duo.

on Intel Core Duo with the hardware prefetcher, and the bars labeled *helper* show the normalized execution time of each application to *base* when the helper thread prefetching is used in addition to the hardware prefetching. Fig. 15 shows the number of L2 misses reduced by our helper thread prefetching.

We see that *bzip2*, *cg*, *em3d*, *mcf*, *mst*, and *swim* have a relatively high value of *L2 cache misses per cycle*. *Helper* reduces the execution time significantly for these applications except for *bzip2*. In *bzip2*, the portion of the target loop in the execution time is small (15.75 percent) compared to other applications. Thus, its performance improvement is small even though its L2 cache miss reduction in the target loop is high in Fig. 15. As we expected from the values of *L2 cache misses per cycle* for the target loops, the L2 cache miss reductions in *equake*, *mg*, and *parser* are small. Despite the L2 cache miss reduction, their performance is degraded due to the synchronization overhead between the application and helper threads.

Similar to the simulation result in Section 6, our synchronization is not performed in every iteration in order to reduce the synchronization overhead. The helper thread is allowed to run ahead by MAX_DIST*LOOP_SYNC_INTERVAL iterations, whose values are shown in the following table.

bzip2	4000	cg	12	em3d	8
equake	100	mcf	256	mg	40
mst	50	parser	200	swim	3

Overall, *CMP* is an attractive architecture to perform helper thread prefetching. Our helper thread prefetching plus hardware L2 prefetching achieves an average speedup of 1.15 over the hardware L2 prefetching for the applications with high *L2 cache misses per cycle* on a real CMP system.

8 Related Work

8.1 Prefetching for SMT Architectures

Most previous studies in executing a helper thread to hide the memory latency of the application thread focus on executing the threads in a tightly coupled system [2], [5], [6], [7], [8], [9], [10], [11], [24], [25], [26], such as in an SMT uniprocessor system. Typically, a program slice is extracted for each critical instruction (i.e., instructions that frequently miss in the L1 cache and hard-to-predict branches). Each program slice is converted into a helper thread. Zilles and Sohi [8], [26] and Roth and Sohi [6] extract the slices statically, while Collins et al. [10] extract the slices dynamically. As a result of the design, the thread granularity is small. This would result in high thread management overhead in loosely coupled systems such as a CMP or an intelligent memory. In addition, the prefetching thread is typically not synchronized due to the granularity. Although Collins et al. [9] proposed a hardware mechanism to prevent their prefetching threads from running too far ahead of the application thread, they do not consider the case where the prefetching helper thread lags behind the application thread. Our approach detects such cases and overcomes the problem of lagging behind with our instant *catch-up* mechanism without requiring hardware support (in the case of CMP). We also present how helper thread prefetching can work on intelligent memory. Also, our approach provides a much larger prefetching thread granularity that tolerates thread management and synchronization overheads well.

Kim et al. [3], [4] showed that helper thread prefetching is quite effective for SMT processors, including the Intel Pentium 4 with hyperthreading. Although their prefetching sections are loop-based regions similar to ours, they only evaluate it for an SMT platform, while we evaluate and optimize our scheme for a CMP and an intelligent memory. Their helper thread is synchronized by suspending and resuming it, resulting in very high synchronization overheads and custom SMT-specific hardware support. Consequently, their approach requires more complex thread management support. In contrast, we use a much faster user-level semaphore without thread suspension/resumption. Furthermore, we provide the new mechanism to allow a lagging helper thread to catch up with the application and architecture support for intelligent memory.

Zhang et al. [27] use helper threads to dynamically optimize the hot traces of the application thread. Hot traces are dynamically detected with the help of a hardware branch history profiler, and their performance is continuously monitored to adaptively determine the best prefetch distance using special hardware tables. Prefetch instructions are inserted into the hot traces (i.e., inlined) by a software optimization thread that is triggered by hardware-generated hot events. Ineffective prefetches are automatically repaired by the software helper thread. Since their scheme is based on predicting stride addresses, it cannot be applied to pointer-chasing applications that lack stride access patterns nor to the applications with complicated control structures that cause difficulties in predicting strides. In addition, their scheme is not free from the drawbacks of inlined prefetching, such as increased register pressure and extra execution cycles due to prefetch address calculation. In contrast, prefetching is decoupled from the application thread in our approach. Our scheme does not suffer from the drawbacks of conventional inlined prefetching. Moreover, our scheme handles irregular access patterns, and it does not require special hardware components for dynamic optimization.

8.2 Prefetching for CMP Architectures

Similar to our work, Brown et al. [2] proposed helper thread prefetching for a CMP system. However, there are several major differences. First, they use very fine-grain prefetching threads, where each thread consists of fewer than 15 instructions and prefetches for only one delinquent load. Such a fine granularity results in high complexities. One complexity is

that prefetching threads need to be spawned early to tolerate spawning overhead and cache miss latency, which is limited by data dependence. Second, chained spawning with multiple helper threads is required for the threads to stay ahead, requiring multiple CMP cores to execute them. Third, the CMP's cache coherence needs read-broadcast and cache-to-cache transfer modifications. Finally, such a granularity will incur a high overhead on the intelligent memory where the communication latency is very high. In contrast, our prefetching scheme relies on large loop regions with millions of instructions, producing several consequences. First, we only require one CMP core to execute one helper thread. Second, our helper thread is only spawned once for each application. Third, the synchronization overhead is minimized by performing it for every multiple of loop iterations. Fourth, the CMP can be kept unmodified while delivering good speedups. Finally, even on intelligent memory where the communication latency is very high and there are no shared caches, our scheme produces speedups comparable to a CMP system.

Lu et al. [28] proposed a helper thread prefetching technique that exploits dynamic optimizations. They use a loop-based prefetching scheme and evaluate it on a real two-way Sun UltraSparc CMP machine. Once a hot loop with many cache misses is selected by the runtime performance monitor, the code for the prefetching helper thread is generated by the runtime optimizer. Since they piggyback the prefetching helper thread on the dynamic optimizer thread, it is important that the prefetching helper thread and the optimization thread are scheduled well so that the dynamic optimization tasks such as phase change detection can take place in a timely manner. Their main thread and prefetching helper thread are synchronized with each other based on a loop iteration counter. When the prefetching helper thread reads the counter and if the main thread's loop iteration has reached a predefined threshold value, the prefetching helper thread skips the current synchronization interval in the loop and starts prefetching for a new synchronization interval with the live-in values provided by the main thread. Even though their counter-based synchronization mechanism is somewhat similar to our semaphore-based synchronization mechanism for CMP architectures, our synchronization mechanism tries to maintain a predetermined iteration distance between the main thread and the helper thread to prefetch in a timely manner and to prevent cache pollution due to prefetching. Moreover, our approach has no scheduling issues between the prefetching helper thread and the dynamic optimization thread, nor does it have any overhead due to dynamic optimization.

Sundaramoorthy et al. [7], [24] proposed the Slipstream approach that observes an instruction retirement stream and removes instructions that were not executed in the application thread to create a shortened thread that is speculative. Although they evaluate their scheme on a CMP, the cores need to be tightly integrated to provide communication of register values and recovery from misspeculation.

8.3 Memory-Side Prefetching

Our work is also related to memory-side prefetching [29], [30], [31], [32], [33], [34], [35], where the prefetching is initiated by an engine that resides in the memory system. Some manufacturers have built such engines [32] such as the Nvidia chipset, which includes the DASP controller in the North Bridge chip [32]. It seems that it is mostly targeted to stride recognition. It also buffers prefetched 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. [30] proposed the Content-Based prefetcher, which is a hardware controller that monitors the data coming from the memory. If an item appears to be an address, the engine prefetches it, allowing automatic pointer chasing. Alexander and Kedem [29] proposed 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. Solihin et al. [14] proposed a thread that runs in an intelligent memory that observes, learns, and prefetches for the miss streams of the applications. In contrast to those studies, our helper thread is constructed out of the application's code.

Another related system is Impulse [36], an intelligent memory controller capable of remapping physical addresses to improve the performance of irregular applications. Impulse implements next-line prefetching, and it buffers the data in the memory controller rather than sending it to the processor. Zhang et al. [35] proposed a pointer-based prefetching mechanism using Impulse. The memory controller identifies accesses to linked objects traversed by pointer chasing and remaps them to a contiguous address region using special hardware logic. If any node whose address is falling into the remapped region is accessed, the memory controller prefetches all objects pointed to by the node. Such address remapping and management require special operating system support. Our scheme is different in that prefetching is done by software with minimal hardware support, and it does not require such operating system support.

Other studies proposed specialized programmable engines. For example, Hughes [31] and Yang and Lebeck [34] proposed 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. Because we do not use specialized prefetching hardware, our helper thread is not limited to just prefetching linked data structures.

The preliminary work of this paper has been presented in [37].

9 Conclusions

We have presented a helper thread prefetching scheme that works effectively on loosely coupled processors, such as in a standard CMP system and an intelligent memory. To alleviate this high interprocessor communication in such a system, we apply two novel techniques. First, instead of extracting a program slice per delinquent load instruction, our helper thread extracts a program slice for a large loop-based code section. Such a large granularity helps to decrease the overheads of communication and thread

management. Second, we present a new synchronization mechanism between the application and the helper thread that exploits loop iterations. The synchronization mechanism precisely controls how far ahead the execution of the helper thread can be with respect to the application and, at the same time, allows the helper thread to catch up to the application when it lags behind. It is also important to make the helper thread run sufficiently far ahead of the application to achieve effective prefetching. In general, the fewer memory access operations a prefetching section has, the shorter the helper thread can be (due to our helper thread construction algorithm), allowing the helper thread to run sufficiently ahead of the main thread. We found that our synchronization mechanism is important in ensuring prefetching timeliness and avoiding cache pollution.

To demonstrate that helper thread prefetching in a loosely coupled system can be done effectively, we evaluate our prefetching by simulating a standard unmodified CMP and an intelligent memory where a simple processor is embedded in memory. We also evaluate our scheme on a real CMP system with an Intel Core Duo processor. The evaluation results show that our helper thread prefetching is attractive to the loosely coupled multiprocessor systems.

Our future work will extend our current proposal across multiple dimensions for the intelligent memory architecture. First, it will be extended to target multiple DRAM chips or DIMMs where additional hardware supports are needed for interchip (or inter-DIMM) communications. Second, it currently targets a uniprogrammed environment where the application and the helper thread execute concurrently without context switches. Our future work will extend it to a multiprogramming environment. There will be additional live-in data values and hardware supports such as page table pointers and TLBs to the memory processor. Third, the CMP system with parallel workloads is another good target for our memory-processor-based approach. Finally, we will investigate the effect of caching the prefetched data in various places along the path from the L2 cache to the memory processor.

ACKNOWLEDGMENTS

This work was supported in part by the IT R&D program of MIC/IITA (2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software), by the Ministry of Education, Science and Technology under the BK21 Project, and by the US National Science Foundation (NSF) through Grant CCF-0347425. ICT at Seoul National University provided research facilities for this study. A preliminary version of this paper appeared in the Proceedings of the 2006 IEEE International Parallel and Distributed Processing Symposium (IPDPS '06).

REFERENCES

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

[2] J.A. Brown, H. Wang, G. Chrysos, P.H. Wang, and J.P. Shen, "Speculative Precomputation on Chip Multiprocessors," *Proc. Sixth Workshop Multithreaded Execution, Architecture and Compilation (MTEAC '02)*, Nov. 2002.

[3] D. Kim, S.-W. Liao, P.H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J.P. Shen, "Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processor," *Proc. Second Int'l Symp. Code Generation and Optimization (CGO '04)*, pp. 27-38, Mar. 2004.

[4] D. Kim and D. Yeung, "Design and Evaluation of Compiler Algorithms for Pre-Execution," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, pp. 159-170, Oct. 2002.

[5] C.-K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," *Proc. 29th Int'l Symp. Computer Architecture (ISCA '01)*, June 2001.

[6] A. Roth and G. Sohi, "Speculative Data-Driven Multithreading," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture (HPCA '01)*, pp. 37-48, Jan. 2001.

[7] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving Both Performance and Fault Tolerance," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, pp. 257-268, Oct. 2000.

[8] C.B. Zilles and G.S. Sohi, "Understanding the Backward Slices of Performance Degrading Instructions," *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00)*, pp. 172-181, June 2000.

[9] J.D. Collins, H. Wang, D.M. Tullsen, C.J. Hughes, Y. fong Lee, D. Lavery, and J.P. Shen, "Speculative Precomputation: Long-Range Prefetching of Delinquent Loads," *Proc. 28th Int'l Symp. Computer Architecture (ISCA '01)*, pp. 14-25, July 2001.

[10] J.D. Collins, D.M. Tullsen, H. Wang, and J.P. Shen, "Dynamic Speculative Precomputation," *Proc. 34th Int'l Symp. Microarchitecture (MICRO '01)*, Dec. 2001.

[11] G.K. Dorai and D. Yeung, "Transparent Threads: Resource Allocation in SMT Processors for High Single-Thread Performance," *Proc. 11th Ann. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '02)*, Sept. 2002.

[12] B. Sinharoy, R.N. Kalla, J.M. Tendler, R.J. Eickemeyer, and J.B. Joyner, "Power5 System Microarchitecture," *IBM J. Research and Development*, vol. 49, nos. 4-5, 2005.

[13] J. Lee, Y. Solihin, and J. Torrellas, "Automatically Mapping Code in an Intelligent Memory Architecture," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture (HPCA '01)*, Jan. 2001.

[14] Y. Solihin, J. Lee, and J. Torrellas, "Automatic Code Mapping on an Intelligent Memory Architecture," *IEEE Trans. Computers*, special issue on advances in high performance memory systems, vol. 50, no. 11, 2001.

[15] J. Renau et al., *SESC*, http://sesc.sourceforge.net, 2004.

[16] M. Weiser, "Program Slicing," *Proc. Fifth Int'l Conf. Software Eng. (ICSE '81)*, pp. 439-449, 1981.

[17] P. Tu and D. Padua, "Automatic Array Privatization," *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems*, LNCS 1808, Springer, pp. 247-281, 2001.

[18] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[19] 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 (ISCA '90)*, pp. 364-373, May 1990.

[20] Intel, *IA-32 Intel Architecture Optimization Reference Manual*, Apr. 2006.

[21] Intel, *Intel C++ Compiler 9.1 for Windows*, 2006.

[22] Intel, *Intel Visual Fortran Compiler 9.1*, 2006.

[23] Intel, *Intel VTune Performance Analyzer 8.1*, 2006.

[24] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A Study of Slipstream Processors," *Proc. 33rd Int'l Symp. Microarchitecture (MICRO '00)*, pp. 269-280, Dec. 2000.

[25] 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 (ISCA '99)*, pp. 186-195, May 1999.

[26] C.B. Zilles and G.S. Sohi, "Execution-Based Prediction Using Speculative Slices," *Proc. 28th Int'l Symp. Computer Architecture (ISCA '01)*, July 2001.

[27] W. Zhang, B. Calder, and D. Tullsen, "Dynamic Speculative Precomputation," *Proc. Fourth IEEE Int'l Symp. Code Generation and Optimization (CGO '06)*, Mar. 2006.

[28] A.D. Jiwei Lu, W.-C. Hsu, K. Nguyen, and S.G. Abraham, "Dynamic Helper Threaded Prefetching on the Sun UltraSparc CMP Processor," *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '05)*, pp. 93-104, Nov. 2005.
[29] T. Alexander and G. Kedem, "Distributed Predictive Cache Design for High Performance Memory Systems," *Proc. Second Int'l Symp. High-Performance Computer Architecture (HPCA '96)*, pp. 254-263, Feb. 1996.
[30] R. Cooksey, D. Colarelli, and D. Grunwald, "Content-Based Prefetching: Initial Results," *Proc. Second Workshop Intelligent Memory Systems*, pp. 33-55, Nov. 2000.
[31] C.J. Hughes, "Prefetching Linked Data Structures in Systems with Merged DRAM-Logic," master's thesis, Technical Report UIUCDCS-R-2001-2221, Univ. of Illinois at Urbana-Champaign, May 2000.
[32] NVIDIA, *Technical Brief: NVIDIA nForce Integrated Graphics Processor (IGP) and Dynamic Adaptive Speculative Pre-Processor (DASP '02)*, http://www.nvidia.com/, 2002.
[33] Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," *Proc. 29th Int'l Symp. Computer Architecture (ISCA '02)*, May 2002.
[34] C.-L. Yang and A.R. Lebeck, "Push versus Pull: Data Movement for Linked Data Structures," *Proc. Int'l Conf. Supercomputing (ICS '00)*, pp. 176-186, May 2000.
[35] L. Zhang, S. McKee, W. Hsieh, and J. Carter, "Pointer-Based Prefetching within the Impulse Adaptable Memory Controller: Initial Results," *Proc. Workshop Solving the Memory Wall Problem*, June 2000.
[36] J.B. Carter et al., "Impulse: Building a Smarter Memory Controller," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture (HPCA '99)*, pp. 70-79, Jan. 1999.
[37] C. Jung, D. Lim, J. Lee, and Y. Solihin, "Helper Thread Prefetching for Loosely-Coupled Multiprocessor Systems," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS '06)*, Apr. 2006.

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 (UIUC) in 1999. After obtaining his PhD degree, he spent half a year at UIUC as a visiting lecturer and postdoctoral research associate. He was an assistant professor in the Department of Computer Science and Engineering, Michigan State University, from January 2000 to August 2002 before joining Seoul National University, where he is currently an associate professor in the School of Computer Science and Engineering. His research interests include compilers, computer architectures, parallel processing, and embedded systems. He is a member of the IEEE and the IEEE Computer Society.

Changhee Jung received the BS degree in computer engineering from Chungnam National University and the MS degree in computer science and engineering from Seoul National University in 2003 and 2005, respectively. He is currently a PhD student in computer science at the Georgia Institute of Technology, Atlanta. From 2005 to 2008, he was a member of the research staff at the Electronics and Telecommunications Research Institute (ETRI), Korea. His research interests include compilers and computer architecture for high-performance and embedded systems. He received the silver prize in the 11th SAMSUNG HumanTech Thesis Competition in 2005. He is a student member of the IEEE and the IEEE Computer Society.

Daeseob Lim received the BE degree in computer science and engineering from Seoul National University, South Korea, in 2004 and the MS degree in computer science from the University of California, San Diego, in 2007. He is currently working toward the PhD degree in the School of Computer Science and Engineering, Seoul National University. His research interests include multimedia network streaming, video-on-demand server architecture, and content distribution network. He is a student member of the IEEE and the IEEE Computer Society.

Yan Solihin received the BS degree in computer science from Institut Teknologi Bandung in 1995, the BS degree in mathematics from Universitas Terbuka Indonesia in 1995, the MASc degree in computer engineering from Nanyang Technological University 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 an associate professor in the Department of Electrical and Computer Engineering, North Carolina State University, Raleigh. He has engaged in teaching computer architecture since 2002. His research interests include high-performance computer architecture, computer architecture support for security and reliability, and performance modeling. He has published more than 40 journal and conference papers in those areas. He has served on the editorial board of the *Journal of Instruction Level Parallelism* and in the program committee of computer architecture and performance modeling symposia and conferences. He was a recipient of 2005 IBM Faculty Partnership Award, 2004 NSF Faculty Early Career Award, and 1997 AT&T Leadership Award. He has graduated three PhD students and eight master's degree students and is currently advising six PhD students. He is a senior member of the IEEE and the IEEE Computer Society, and a member of the ACM SIGMICRO.