Helper Thread Prefetching for Loosely-Coupled Multiprocessor Systems[*]

Changhee Jung[1‡], Daeseob Lim[2‡], Jaejin Lee[3], and Yan Solihin[4†]

[1]Embedded Software Research Division
Electronics and Telecommunications Research Institute
Yuseong-Gu, Daejeon, 305-530, Korea
chjung@etri.re.kr

[2]Department of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093-0114 USA
dalim@cse.ucsd.edu

[3]School of Computer Science and Engineering
Seoul National University
Seoul 151-744, Korea
jlee@cse.snu.ac.kr

[4]Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695-7911 USA
solihin@eos.ncsu.edu

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 fine-grain 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, inter-processor 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 in a standard, unmodified CMP system, and in 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.

[*]This work was supported in part by the Ministry of Education under the Brain Korea 21 Project, by the Ministry of Information and Communication under the IT Leading Research and Development Project, and by the Institute of Computer Technology at Seoul National University. Correspondence to jlee@cse.snu.ac.kr.
[†]Solihin was supported by National Science Foundation through grants CCF-0347425 and CNS-0406306, and by IBM through IBM Faculty Partnership Award.
[‡]Jung and Lim did their work when they were students of Seoul National University.

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 [2, 10, 11, 13, 16, 19]. 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 Multi-Threaded (SMT) system [2, 4, 10, 11, 13, 16]. 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 [6]; 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 CMP or an intelligent memory system. Loosely-coupled processors have an advantage that threads do not contend for fine-grain 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 inter-processor 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 [12, 17], yielding very large prefetching regions (millions of instructions).

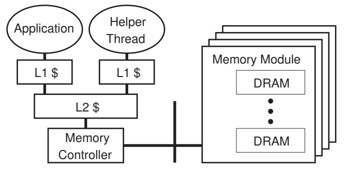
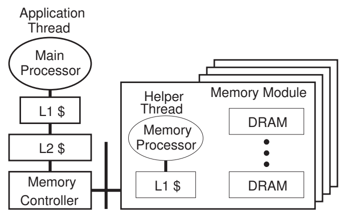
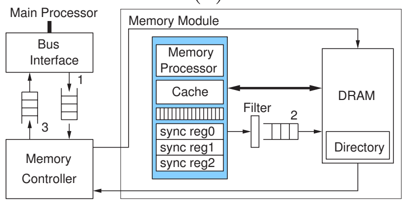


Figure 1. The CMP architecture used in this paper.



(a)



(b)

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

- 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.
- A mechanism for the prefetching thread to instantly catch up to the application execution.
- Characterization and analysis of the effects of non-memory operations on the helper thread's performance.
- 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.

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 that 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.

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 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.

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.

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; Section 6 presents the evaluation results; Section 7 discusses related work; and Section 8 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 Figure 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).

2.2. The Intelligent Memory Architecture

For intelligent memory, we need the extra hardware support shown in Figure 2. Figure 2(a) 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. Firstly, commodity DRAM chips can be used because they do not need to be modified. Secondly, 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

```
long refresh_potential
   (network_t * net) {
 node_t * node, * tmp;
 ... // some computation
 while (node != root) {
  while (node) {
   if(node->orientation
     == UP) {
    node->potential
     = node->basic_arc->cost
     + node->pred->potential;
   } else {
    node->potential
     = node->pred->potential
     - node->basic_arc->cost;
    checksum++;
   }
   tmp = node;
   node = node->child;
  }
  node = tmp;
  while (node->pred) {
   tmp = node->sibling;
   if(tmp) {
    node = tmp;
    break;
   } else
    node = node->pred;
  }
 }
}
```

(a)

```
long refresh_potential
   (network_t * net) {
 node_t * node, * tmp;

 while (node != root) {
  while (node) {
   pref(node);
   pref(node->pred);
   pref(node->basic_arc);
   tmp = node
   node = node->child;
  }
  node = tmp;
  while (node->pred) {
   tmp = node->sibling;
   if (tmp) {
    node = tmp;
    break;
   } else
    node = node->pred;
  }
 }
}
```

(b)

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

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 Figure 2(b), 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 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 to the main processor are placed in *queue3*. When prefetch replies reach the memory controller, their addresses are compared to the main processor's miss requests in *queue1*. If the prefetched 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.

Figure 2(b) 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 are 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. Figure 2(b) 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 one of the clean states (clean-exclusive or clean-shared).

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, the block is returned to the processor, while the stale bit is set (to one) in the directory. If the main processor writes to a line that is in a clean state, it sends a *stale bit update* request to the memory processor, which sets the stale bit in the directory. 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.

```
long refresh_potential(network_t * net) {
  node_t * node, * tmp;
  int syncInterval = LOOP_SYNC_INTERVAL;
  sema_init(&sem_loop_sync, MAX_DIST);
  sema_V(&sem_helper_start);
  ... // some computation
  while (node != root) {
    while (node) {
      ...
    }
    ...
    if (!(- -syncInterval)) {
      syncInterval = LOOP_SYNC_INTERVAL;
      node_main = node;
      sema_V(&sem_loop_sync);
    }
  }
}
```

(a)

```
long refresh_potential(network_t * net) {
  node_t * node, * tmp;
  int syncInterval = LOOP_SYNC_INTERVAL;
  sema_P(&sem_helper_start);

  while (node != root) {
    while (node) {
      pref(node);
      pref(node->pred);
      pref(node->basic_arc);
      ...
    }
    ...
    if (!(- -syncInterval)) {
      syncInterval = LOOP_SYNC_INTERVAL;
      if (sem_loop_sync.count > MAX_DIST) {
        sema_init(& sem_loop_sync, MAX_DIST);
        node = node_main;  // catch up
      } else
        sema_P(&sem_loop_sync);
    }
  }
}
```

(b)

Figure 4. Synchronizing the main and helper threads in a CMP: (a) the main thread and (b) the helper thread.

```
long refresh_potential(network_t * net) {
  node_t * node, * tmp;

  SYNC_CLEAR_MAIN();
  SYNC_SIGNAL();
  ... // some computation
  while (node != root) {
      ...
    SYNC_MAIN_INC();
    SYNC_SET_ADDR(node);
  }
}
```

(a)

```
long refresh_potential(network_t * net) {
  node_t * node, * tmp;

  SYNC_CLEAR_HELPER();
  SYNC_WAIT();
  while (node != root) {
    ...
    SYNC_HELPER_INC();
    while (SYNC_TEST&SET(MAX_DIST,&node));
  }
}
```

(b)

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

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 [12, 17]. The prefetching sections are typically loop nests or a non-leaf subroutine consisting 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 [10, 11, 13].

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

1. 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 by 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.

Figure 3(a) shows the original `refresh_potential()` subroutine in *mcf* of Spec2000, which is selected as a prefetching section. In the constructed helper thread (Figure 3(b)), `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 (line `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.

Our helper thread prefetching mechanism does not distinguish multiple access streams (e.g., one loop accesses two different linked lists simultaneously), and performs well in such cases.

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, 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 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.

Figure 4(a) shows the original prefetching section of the application thread, augmented with synchronization that controls the helper thread's execution distance, while Figure 4(b) 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 `03` in (b), it busy waits there until the application thread calls `sema_V(&sem_helper_start)` on line `04` in (a). The helper thread will synchronize 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 `03`, 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 `14`). 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 `16`), and setting its current traversal node to the application thread's current node `node_main` (line `17`). If it is not running behind, it simply consumes a signal and continues, or waits until a signal is available (line `19`).

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 uncachable region in memory so that the updates by the application and the helper thread are seen by each other. If the semaphores were cachable, 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. However, when the semaphores are not cached, each helper 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 (Figure 2(b)).

The first two registers (main_count, 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.

Figure 5(a) and (b) 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 `04` in (b) until the application thread calls `SYNC_SIGNAL()` on line `04` in (a). `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 `08` in (b). 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

Not only can we apply this synchronization mechanism to pointer chasing loops, but also to regular loop nests with an outer loop index variable `i` found in scientific/numerical applications. 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 the sync_addr, allowing the helper thread to copy its value and catch up when it discovers that it is running behind. Although there may be more complex code in which the *catch-up* mechanism cannot be applied, we do not encounter such a case in in the prefetching sections of the benchmarks that we evaluate.

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.

We apply the helper thread construction algorithm by hand. We identify the target loops for helper threading using profiling. We use the same input in profiling as well as testing.

Simulation Environment. The evaluation is performed using a cycle-accurate execution-driven simulator that models dynamic superscalar processor cores, CMP, and intelligent memory [8]. The MIPS instruction set is used for simulation. 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 3.2GHz cycles. We model a uni-programmed 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 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 following delays for simulation:

- 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.
- 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, not 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 [9], but the prefetched lines are placed in the L1 cache.

5. Performance Characterization

In this section, we evaluate and analyze the effects of non-memory operations on the prefetching performance. Intuitively, the more non-memory 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 non-memory operations is varied.

To achieve that, we create a synthetic benchmark that performs linked-list traversal on a fixed number of nodes, and vary the amount of non-traversal operations in each iteration. Figure 6(a) and (b) show the speedup ratio 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 for the entire traversal, including the synchronization overhead. Therefore, when non-memory operations are added into the code, the execution time increases along the x-axis. The helper thread runs on the memory processor in DRAM in the intelligent memory configuration. The figure shows that there are three cases.

In **Case 1**, when there is a small fraction of L2 cache misses eliminated, there is a slight slow down in the application. This is because the helper thread code is not much shorter than the application code. When the helper thread runs on the memory processor, it is either slower or not much faster than the application 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 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:

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

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. The formula shows that if 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}). Note that the fraction of eliminated L2 cache misses in Figure 6(b) corresponds to $1 - \frac{Miss_{new}}{Miss_{orig}}$ in the formula. Since T_{orig} and T_{orig_mem} are constant in the experiment, the formula states that the application's speedup after prefetching is proportional to the fraction of the eliminated cache misses. This explains why the shape of the figure in Case 2 in Figure 6(a) follows that in Figure 6(b).

In **Case 3**, the helper thread cannot eliminate any more L2 cache misses (Figure 6(b)). However, the speedup in Figure 6(a) decreases since the non-memory-stall fraction of the execution time also grows, hence reducing the speedup.

We also have varied the number of nodes that are traversed. Although the results are not shown in Figure 6, they exactly follow the trends in the figure.

6. Evaluation and Discussion

6.1. Prefetching Performance

Figure 7 compares the execution time of the entire applications (not just the targeted loops) for the following cases: a) there is no prefetching (*nopref*); b) there is only hardware processor-side prefetching (*c8*); c) our helper thread prefetching is running on a memory processor in the DRAM chip (*helper*); d) the hardware prefetching is backed up by the helper thread prefetching in the memory processor in DRAM (*c8+helper*); e) in the DIMM (*DIMM*); and f) 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*).

Appl	Suite	Problem	Input	# of target loops	% execution time
bzip2	SpecInt2000	Compression/Decompression	Reference	1	29.51%
cg	NAS	Conjugate gradient	Class W	1	91.63%
em3d	Olden	Electromagnetic wave propagation in 3D	200,000	1	47.55%
equake	SpecFP2000	Seismic wave propagation simulation	Reference	1	65.16%
mcf	SpecInt2000	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	SpecInt2000	Word processing	Subset of Reference	1	19.48%
swim	SpecFP2000	Shallow Water Modeling	Train	3	99.21%

Table 1. Applications used.

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 prefetcher

Table 2. Parameters of the simulated architecture. Latencies correspond to contention-free conditions. *RT* **stands for round-trip** *from the processor*. **All cycles are 3.2 GHz cycles.**

On average, *beyondL2(Target)* is the most significant component of the processor's stall time in *nopref*, accounting for 61% 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. *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 average, *c8* reduces the execution time by 11%.

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 the best on average. It removes over 50% of the *beyondL2(Target)* time and reduces the total execution time by 24%, 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.

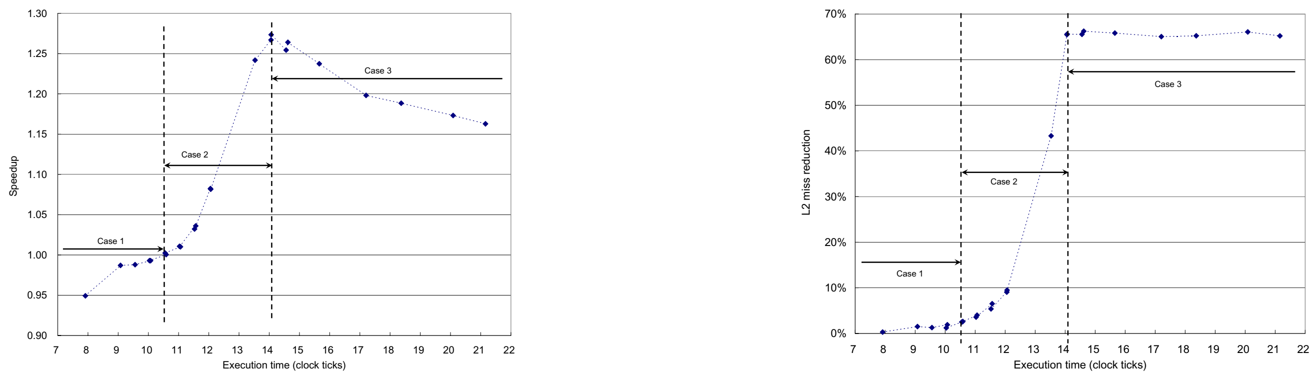
Finally, *DIMM* and *CMP* also achieve very good speedups. Despite the memory processor suffering from higher memory access latency in the DIMM, *DIMM* achieves an average speedup of 1.26. In *CMP*, the very high memory access latency in the CMP processor is partially offset by the speed of the processor. 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

Figure 8 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 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 L2 cache (*Redundant*). The figure shows that on average, almost 60% of the L2 cache misses are prefetched by the *helper*. However, *helper* also generates 27% 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

Figure 9 compares the execution time of the application thread using *c8+helper* prefetching when the synchronization uses un-



(a) (b)

Figure 6. The prefetching performance for a fixed number of nodes in a linked-list traversal: (a) speedup and (b) the fraction of L2 cache misses that are eliminated. The unit of execution time is one million clock ticks.

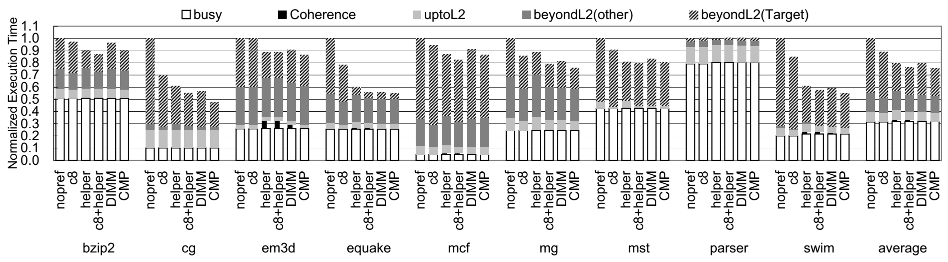


Figure 7. Execution time of the applications with different prefetching schemes and different architectures.

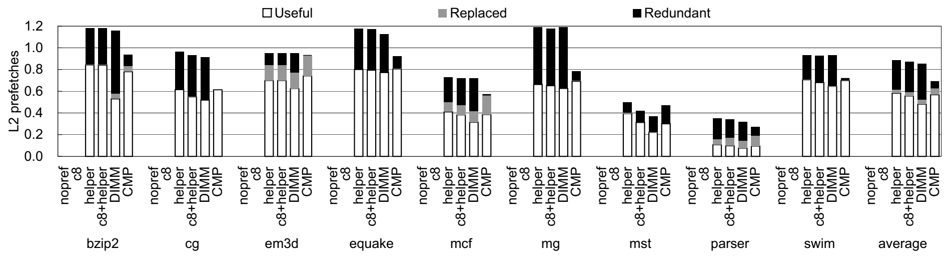


Figure 8. Breakdown of the L2 misses and lines prefetched by the helper thread. Note that *c8* does not have any bars because it prefetches data into the L1 cache of the main processor.

cachable 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% on 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).

6.4. Main Memory Bus Utilization

Figure 10 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 10-15% 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% in *c8+helper* for *bzip2*.

6.5. Synchronization Intervals

Our synchronization is not performed for every iteration. Instead, it is performed for 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:

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

Thus, if LOOP_SYNC_INTERVAL*MAX_DIST is 120 and LOOP_SYNC_INTERVAL is 40, then MAX_DIST is 3. We found these values through experiments. 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.

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 3 target loops), the

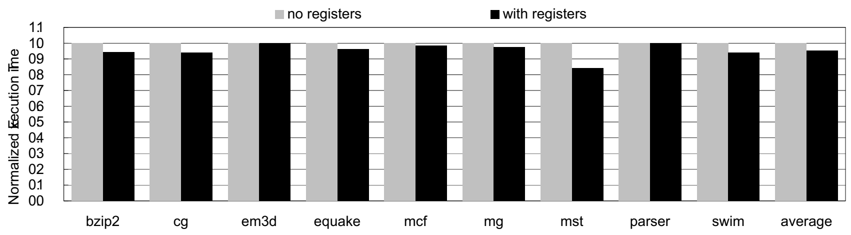


Figure 9. The performance comparison of the cases with and without special synchronization registers.

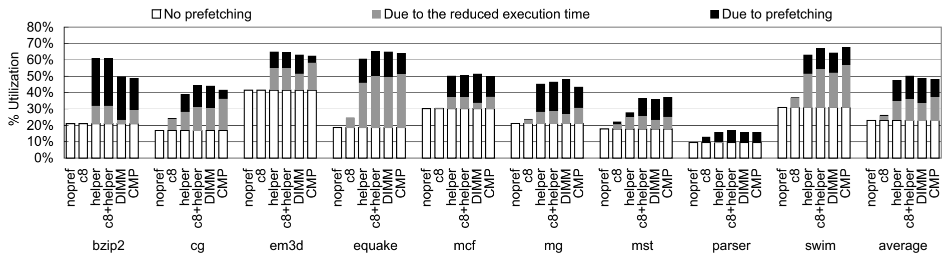


Figure 10. Main memory bus utilization for the target loops.

helper thread's code size is very small compared to the application.

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. Related Work

Most previous studies in executing a helper thread to hide memory latency of the application thread focus on executing the threads in a tightly-coupled system [2, 3, 4, 6, 13, 15, 16, 19, 21, 22], such as in a simultaneous multi-threaded (SMT) uniprocessor system. Typically, a program slice is extracted for each critical instruction (i.e., instructions that frequently miss in L1 cache and hard-to-predict branches). Each program slice is converted into a helper thread. Zilles and Sohi [21, 22] and Roth and Sohi [16] extract the slices statically, while Collins et al. extract the slices dynamically [4]. 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. In contrast, our approach provides a much larger prefetching thread granularity that tolerates thread management and synchronization overheads well.

Kim *et al.* [10, 11] showed that helper thread prefetching is quite effective for SMT processors, including the Intel Pentium 4 with Hyper-Threading. 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 a new mechanism to allow a lagging helper thread to catch up with the application and architecture support for intelligent memory.

Similar to our work, Brown *et al.* [2] proposes 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, 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 is no shared caches, our scheme produces speedups comparable to a CMP system.

Sundaramoorthy *et al.* [15, 19] proposes the Slipstream approach that observes instruction retirement stream and removes instructions that were not executed in the main 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.

Our work is also related to memory-side prefetching [1, 5, 7, 14, 18, 20], where the prefetching is initiated by an engine that resides in the memory system. Some manufacturers have built such engines [14], such as the NVIDIA chipset which includes the DASP controller in the North Bridge chip [14]. 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.* [5] 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 [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. Solihin *et al.* [17] 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.

Other studies proposed specialized programmable engines. For example, Hughes [7], and Yang and Lebeck [20] 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.

8. Conclusion

We have presented a helper thread prefetching scheme that works effectively on loosely-coupled processors, such as in a standard chip multi-processor (CMP) system and an intelligent memory. To alleviate this high inter-processor communication in such a system, we apply two novel techniques. Firstly, 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. Secondly, 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. We found that this feature 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 in a standard, unmodified CMP, and in an intelligent memory where a simple processor is embedded in memory. 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. No hardware modifications are made to the CMP, while for the intelligent memory, simple hardware support for synchronization, coherence, and handling prefetch requests are needed.

References

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

[2] J. A. Brown, H. Wang, G. Chrysos, P. H. Wang, and J. P. Shen. Speculative Precomputation on Chip Multiprocessors. In *the 6th Workshop on Multithreaded Execution, Architecture (MTEAC-6)*, November 2002.

[3] R. S. Chappell, J. Stark, S. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of The 26th International Symposium on Computer Architecture (ISCA'99)*, pages 186–195, May 1999.

[4] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of The 34th International Symposium on Microarchitecture (MICRO-34)*, December 2001.

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

[6] G. K. Dorai and D. Yeung. Transparent threads: Resource allocation in smt processors for high single-thread performance. In *Proceedings of the 11th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT-11)*, September 2002.

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

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

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

[10] 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. In *the 2nd International Symposium on Code Generation and Optimization (CGO 2004)*, pages 27–38, March 2004.

[11] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 159–170, October 2002.

[12] J. Lee, Y. Solihin, and J. Torrellas. Automatically mapping code in an intelligent memory architecture. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, January 2001.

[13] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 29th International Symposium on Computer Architecture*, June 2001.

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

[15] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proceedings of The 33rd International Symposium on Microarchitecture (MICRO-33)*, pages 269–280, December 2000.

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

[17] Y. Solihin, J. Lee, and J. Torrellas. Automatic code mapping on an intelligent memory architecture. *IEEE Transactions on Computers: Special Issue on Advances in High Performance Memory Systems*, 50(11), 2001.

[18] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[19] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of The 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 257–268, October 2000.

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

[21] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *Proceedings of The 27th International Symposium on Computer Architecture (ISCA'00)*, pages 172–181, June 2000.

[22] C. B. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of The 28th International Symposium on Computer Architecture (ISCA'01)*, July 2001.