

Efficient Emulation of Hardware Prefetchers via Event-Driven Helper Threading

Ilya Ganusov and Martin Burtscher
Computer Systems Laboratory, Cornell University
Ithaca, New York
{ilya, burtscher}@csl.cornell.edu

ABSTRACT

The advance of multi-core architectures provides significant benefits for parallel and throughput-oriented computing, but the performance of individual computation threads does not improve and may even suffer a penalty because of the increased contention for shared resources. This paper explores the idea of using available general-purpose cores in a CMP as helper engines for individual threads running on the active cores. We propose a lightweight architectural framework for efficient event-driven software emulation of complex hardware accelerators and describe how this framework can be applied to implement a variety of prefetching techniques. We demonstrate the viability and effectiveness of our framework on a wide range of applications from the SPEC CPU2000 and Olden benchmark suites. On average, our mechanism provides performance benefits within 5% of pure hardware implementations. Furthermore, we demonstrate that running event-driven prefetching threads on top of a baseline with a hardware stride prefetcher yields significant speedups for many programs. Finally, we show that our approach provides competitive performance improvements over other hardware approaches for multi-core execution while executing fewer instructions and requiring considerably less hardware support.

Categories and Subject Descriptors

B.8 [Performance and Reliability]: General

General Terms

Performance

Keywords

prefetching, helper threading, multi-core architectures

1. INTRODUCTION

All major high-performance microprocessor manufacturers have announced or are already selling chips with two

to eight cores. Future generations of these processors will undoubtedly include more cores. While multiple cores are immediately beneficial in multiprogrammed environments and for parallel applications, the performance of individual computation threads does not improve and may even suffer a penalty because of increased contention for shared resources such as caches. Moreover, manually parallelizing applications to obtain a benefit from multiple cores increases the software complexity and cost. Finally, many general-purpose applications, including ones that are easy to parallelize, exhibit limited scalability and may not be able to take advantage of additional cores beyond a certain point.

Improving the performance of single threads in a multi-core environment has proven to be difficult. Moreover, multi-core architectures favor simpler and smaller cores, which limits the opportunity to exploit the available ILP with wide-issue cores. On the other hand, special-purpose hardware accelerators that are located outside the core can improve a thread's performance by eliminating control and memory bottlenecks (e.g., advanced branch predictors and data prefetchers), but they often result in significant chip area additions and additional complexity. In light of these trends, architectural techniques that allow the use of additional cores to speed up single threads are becoming an attractive alternative [18].

This paper describes and evaluates a lightweight architectural framework that allows otherwise idle cores in a CMP to function as helper engines for the individual threads running on the active cores. Our technique employs special helper threads to emulate hardware prefetchers. These threads are launched when a helper core receives a special event trigger from a core that is running in conventional, non-helper mode. Event triggers are sent to the helper cores through a unidirectional communication interface, which can be configured to attach different data to the triggers.

We demonstrate the viability and effectiveness of the proposed framework by using it to implement a wide variety of simple and complex hardware prefetching algorithms [3, 5, 7, 8]. The performance of the emulated accelerators is within 5% of pure hardware implementations. Furthermore, we demonstrate that running prefetching event-driven helper threads (EDHTs) on top of a baseline with hardware stride prefetching yields speedups of 5%-100% on a wide range of programs. Finally, we implement two other recently proposed hardware-only techniques for multi-core execution [4, 26] and show that even without customizing the EDHT prefetching threads for each application, EDHT can provide competitive performance improvements while

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FACT'06, September 16–20, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-264-X/06/0009 ...\$5.00.

executing fewer instructions and requiring considerably less hardware support.

We believe that such a framework allows multi-core processors to provide immediate benefits and presents a relatively simple yet effective architectural enhancement to exploit additional cores to speed up individual threads. Unlike previously proposed approaches for software prefetching, our EDHT mechanism can improve performance without the need to modify or analyze the original binary. Moreover, EDHT solves many problems that have hampered the introduction of complex hardware prefetching algorithms into commercial microprocessors. Specifically, EDHT needs minimal hardware modifications, does not require specialized hardware storage for prediction tables, and can be easily reconfigured to customize prefetching algorithms for individual applications.

The rest of the paper is organized as follows. Section 2 provides an overview of outcome prediction-based prefetching algorithms. Section 3 presents the architectural support for EDHT as well as its operation. Sections 4 and 5 describe the simulation methodology and evaluate our design. Section 6 discusses related work and Section 7 concludes the paper.

2. DATA PREFETCHING TECHNIQUES

Hardware prefetchers often rely on various kinds of address predictors to dynamically predict which memory addresses to prefetch. In this work, we examined stride-based address prediction and Markov address prediction. This section briefly discusses both approaches and introduces the basic algorithms that we later use.

2.1 Stride Prefetching

Stride prefetchers represent the most common form of prefetching based on outcome prediction. The stride prefetcher is usually located in the cache controller, where it monitors the stream of cache miss requests observed by the cache. Stride prefetchers identify distinct streams within the sequence of cache misses, associate strides with each of the streams, and issue memory requests for the next few addresses in the stream. The simplest form of stride prefetching is next-sequential prefetching, in which the prefetcher issues a request for cache line $L+1$ as soon as line L is referenced [17].

In many cases cache miss addresses are composed of several interleaved streams. A typical case of multiple streams is the traversal of several arrays within a matrix-matrix multiplication loop. Stride prefetchers employ special mecha-

nisms to decipher and disambiguate interleaving streams. If the memory hierarchy propagates the program counter (PC) of the instructions that cause cache misses, the prefetcher can attribute misses to specific instructions and track streams on a per PC basis [3]. We call this *local stride prefetching*. The conventional implementation of a local stride prefetcher uses a table to store stride-related local history information and is shown in Figure 1a. The PC of a load instruction indexes the table. Each table entry holds the load's last stride and the last address. A prefetch is triggered when a load causes a cache miss and its last stride is equal to the current stride.

If PC information is not available, then a *global stride prefetcher* can be used. Such stride prefetchers need to identify distinct streams within the global memory access pattern. Minimum delta prediction and memory partitioning were proposed to handle this problem. Minimum delta prediction associates a miss with the stream or prior miss that is the closest. Memory partitioning separates the physical memory address space into regions and attributes all misses falling within a single region to a single stream [25]. Figure 1b shows the organization of a global stride prefetcher. When a cache miss occurs, the global miss history buffer is searched for the minimum difference between the previously observed addresses and the current miss address. An address stream is identified if the global miss history contains an address that differs from the current address by two minimum deltas.

For each identified address stream, the prefetcher allocates a prefetch stream buffer from a limited number of available buffers. This buffer contains information about the current stream base address and the associated stride. Typically, stride prefetchers use an LRU replacement policy for stream buffers. The newly allocated stream buffer issues prefetches and then waits for the prefetched cache lines to be requested by the processor. Upon receiving such a notification, the stride prefetcher looks up its stream table to see which stream entry the consumed address belongs to. If it finds a match, it increments the corresponding stream address by the stream's stride and issues one new prefetch to keep up with the data consumption of the processor.

2.2 Markov Prefetching

Markov prefetching [7] is another example of outcome-based prediction. A Markov predictor assumes that the address stream of a program can be approximated by a Markov model. A Markov model is a probabilistic state machine with a set of states and state transition probabilities. Each transition from state A to B is assigned a weight representing the fraction of A states that are followed by B states. Figure 2a presents an example of a Markov model. The states in the Markov model are determined by the set of previously seen values.

Markov models are usually characterized by two parameters. The first parameter determines what kind of values defines a state. In case of prefetching, the most common way is to use either the absolute addresses or the differences between consecutive addresses. The second parameter determines how many values are used to determine the state. An order n Markov predictor associates each state with the n previous values.

Markov prefetching techniques incrementally build an Markov model for the target application at run-time. This model

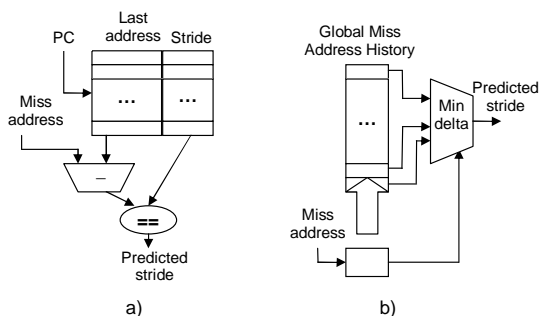


Figure 1: Organization of stride prefetchers

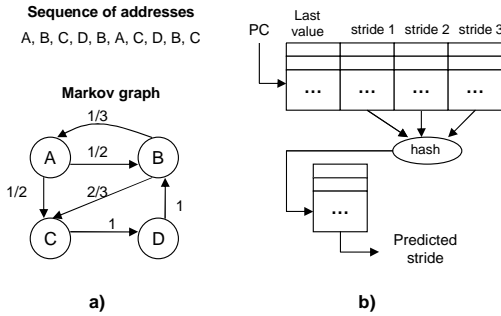


Figure 2: Markov prefetching

is later used by a prefetching mechanism to predict future addresses. Previous work on hardware-based Markov prefetching concentrated on finding optimal parameters for an accurate Markov model that work well for many applications and on devising efficient hardware designs to store this model. The most common hardware implementation in the literature is based on a two-level table representation. The first table contains information to determine the current state (i.e., node in a Markov model). As in the case of stream prefetchers, there are global and a local versions of Markov prefetchers. A local Markov prefetcher, shown in Figure 2b, uses the load’s PC to index a first-level table, which stores a local history of the last three deltas for that load. A global Markov prefetcher uses a global history of the last three deltas (instead of a per-PC local history).

The calculated state serves as an index into the second-level table, which stores predictions (the immediate neighbors of the current node). To limit the total area required for the table, the second-level table usually contains only a limited number of unique predictions. A Markov prefetcher can prefetch the addresses predicted by the adjacent nodes in the Markov model. We refer to this policy as *width* prefetching. However, it is also possible to perform *depth* prefetching in which the sequence of arcs in the Markov model is traversed with prefetching initiated at each node along the path. We use a combination of width and depth prefetching in our experiments.

This subsection presents an overview of how Markov prefetching works. Markov prefetchers incrementally build a Markov model for the address stream generated by the program. A Markov model can be characterized by three parameters: the kind of values that determine the state, the model order, and the maximum number of node neighbors in the graph. Prefetchers utilize Markov models to make predictions. These prefetchers are most commonly characterized by their prefetch distance and depth.

3. IMPLEMENTATION

As demonstrated in the previous section, hardware prefetching algorithms based on outcome prediction are naturally decoupled from the execution of the target thread. The only data dependence between the target thread and the prefetching algorithm is the information about the cache address and the load instruction that caused it. As such, these prefetch mechanisms could be emulated by a software prefetching thread that is started whenever the target thread experiences a cache miss. However, a software implementation of a hardware prefetcher faces two obstacles. First,

such prefetch threads would need to be spawned quickly on microarchitectural events as opposed to program events in the conventional multithreading paradigm. Second, complex prefetch algorithms, such as Markov prefetching, often require a large amount of state that needs to be stored somewhere.

To overcome these issues, it is necessary for a processor architecture to support explicit communication of microarchitectural events from one thread to another. Ideally, the target thread should not be aware of this communication so that prefetching can be easily turned on and off depending on the availability of hardware resources. To achieve this, we propose Event-Driven Helper Threading (EDHT), which provides a way for a low-latency, unidirectional event trigger transmission between regular and helper threads. EDHT threads execute on a conventional core of a chip multiprocessor and the state of the underlying prefetching algorithm is loaded into the private data cache of the core via the conventional memory hierarchy. The rest of this section explains the hardware support for EDHT and its operation.

3.1 Overview of Operation

Figure 3a shows a typical pointer chasing loop. Each iteration of the loop processes a node of a linked list and fetches the next node to be processed. Assuming that the linked list is not cache-resident, each access to the field `data` of the list’s node will result in a cache miss and stall the processor. As shown in Figure 3a, this access translates into a single load instruction `data=pointer->data`. Markov prefetchers are a good fit for prefetching such linked list traversals since they can memorize previous traversals of this list and accurately predict the next node’s location.

A typical hardware implementation of a Markov prefetcher will snoop the cache miss addresses coming out of the cache. For each observed cache miss, it will look up its prediction table similar to the one shown in Figure 2b and determine which addresses are likely to be referenced next. After that, it will issue prefetch requests for those addresses into the memory hierarchy in the hope that the program will soon request those data as well.

Figure 3b demonstrates a conceptual prefetching thread that can emulate a prefetcher. The prefetching thread consists of an infinite while loop. In each iteration, the prefetching loop stalls waiting for cache miss event data to arrive from the target thread. When the target thread experiences

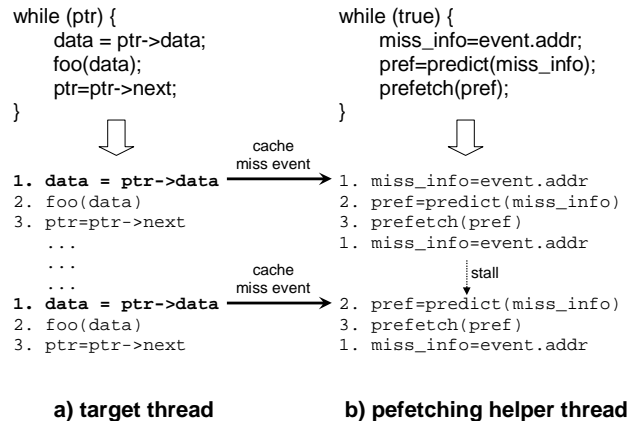


Figure 3: Code example

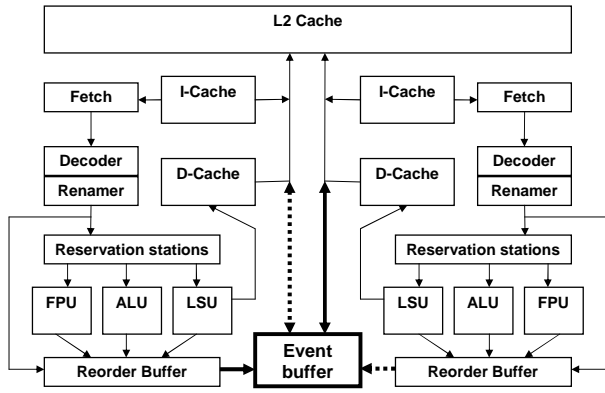


Figure 4: System Architecture

such a miss, the prefetching thread loads the cache miss address along with the associated PC value and uses this information to run its prediction algorithm. Finally, it issues a prefetch instruction and loops back. If by this time there is a new miss event waiting in the event queue, it immediately executes the prefetching algorithm with the new data. Otherwise, it stalls until a new event arrives.

Of course, a dedicated hardware implementation will execute the prediction algorithm much faster than its software thread equivalent. Hence, there may be situations when the frequency of cache miss events outpaces the speed at which the prefetching thread can process them. In this case, limited buffering can be provided to store unprocessed events. The next subsection describes the organization of such a buffer and its interaction with other components of the CPU.

3.2 Implementation and Hardware Support

Even if the architecture provides a way to communicate architectural events to user-level threads, prefetching-based event-driven helper threading is likely to perform poorly on current hardware due to the following reasons. First, synchronization must occur via the operating system or via spin-locks. In the OS case, the trap and return time is so large that it can negate any performance advantage. In addition, the communication between the threads must occur via shared memory, increasing the contention for the shared cache ports and wasting dynamic power. Thus, for EDHT-based prefetching to be successful, it is imperative that the implementation of the event delivery mechanism be efficient.

Since much of the problem related to the implementation of fast inter-thread communication is due to the use of shared memory, we propose to use an *event buffer*. Figure 4 shows the architecture of an out-of-order CMP that supports event-driven helper-threading. The event buffer and associated datapaths are highlighted in bold. The regular computation thread executes on the “left” core, while its EDHT prefetching thread is running on the “right” core. The event buffer is located between the two cores. It represents a FIFO structure of limited capacity. It receives information about cache miss events from the ROB of one of the cores. The ISA abstraction for the event buffer is an I/O device that can be accessed by program threads via I/O read instructions from a reserved address. The event buffer controller snoops the cache bus and supplies data for all event buffer I/O reads.

The instructions in the left core execute and commit normally. If a load instruction experiences a cache miss, a prefetch trigger is transmitted to the event buffer once the load commits. Event transmission is also triggered if a load instruction loads data that has been prefetched by the prefetching thread. The data that needs to be transferred includes the instruction’s PC value and the referenced memory address. Some prefetching mechanisms also require information on whether the event was triggered by a cache miss or a correctly prefetched cache miss.

In the meantime, the prefetch thread is running on the helper core. When the prefetching thread issues an I/O read instruction to obtain data about a cache miss event, it stalls waiting for a reply. When the event buffer receives this data, it supplies the received information to the stalling read request. Then the prefetching thread calculates the prefetch addresses based on the underlying algorithm and executes a non-binding prefetch load instruction that will fetch data into the shared memory hierarchy. After that, it loops back and issues another I/O read instruction to obtain information about the next architectural event.

The cache lines that were prefetched by the helper core are tagged. When the regular core references a prefetched cache line, it marks the executed load instruction as the consumer of the prefetched data. When this load instruction commits, it causes a prefetch trigger to be transmitted to the helper core as if this instruction had experienced a cache miss. This mechanism essentially mimics the way traditional hardware prefetchers work and allows the helper core to stay ahead of the data consumption of the regular thread.

The operation of EDHT has the following implications on the operating system. EDHT threads and the main application run in the same address space and hence share a page table. EDHT threads do not use absolute code addresses (they are fully relocatable) and use a thread-private data area to avoid conflicts with the main thread. The OS scheduler should also be aware of the helper threads associated with each application and schedule them as a group to execute on cores that share at least one level of memory hierarchy. Note that, while we have presented EDHT for two cores running a single main thread, the technique can be extended to support multiple cores by providing either additional event buffers between pairs of cores or the ability to communicate events from different cores to a centralized buffer.

3.3 Prefetching Algorithms

To demonstrate the efficiency and flexibility of the proposed architectural framework, we implement a number of prefetching algorithms on it. This section presents a detailed description of these algorithms.

First, we implement a conventional stride prefetcher. We evaluate both a local and a global stride prefetcher and name them *lstride* and *gstride*, respectively. The local stride prefetcher uses a table with 1K entries to keep track of strides on a per-load basis. The global prefetcher keeps a history of the 8 last miss addresses to identify address streams. Both kinds of prefetchers have 8 stream buffers and utilize an LRU replacement policy for buffer allocation.

Second, we implement two prefetching algorithms that are based on a third-order delta-correlation Markov model with a table size that fits into the L1 data cache. Unlike with stride prefetching, the algorithmic difference between the

Table 1: EDHT threads for emulating hardware prefetching mechanisms

prefetch algorithm	description	# of insns	load insns	longest dep. chain
global stride	8 simultaneous streams, 8-entry miss history buffer, prefetch distance of 8	52 (23)	12 (10)	15 (10)
local stride	1K-entry PC-indexed prediction table, prefetch distance of 8	18 (23)	4 (10)	6 (10)
DFCM	128-entry 3rd order L1 table, 2bc confidence, 16K-entry L2 table, select-fold-shift-xor (SFSX) hash function, prefetch distance of 8	26	6	7
markov	128-entry 3rd order L1 table, 8K-entry L2 table storing 2 distinct predictions, SFSX hash function, prefetch distance of 4	29	7	8
correlation	128-entry 1st order L1 table, 256K-entry L2 table storing 2 distinct sequences of 4 predictions, prefetch distance of 4	24	6	6

global and local versions of the Markov prefetcher is minimal. Therefore, we evaluate only local prefetchers. Our first prefetching algorithm is based on the differential finite context method (*DFCM*) value predictor [5]. This algorithm uses a 2-bit counter-based confidence estimator in the first-level table to suppress low-confident predictions for load instructions that exhibit unpredictable behavior. The second algorithm is more similar to a conventional Markov model. It stores two distinct values (predictions) in each entry of its second-level table. Each value in the second-level table is associated with a 1-bit confidence counter, which is incremented every time a particular prediction is observed to be correct. Thus, the confidence is associated with transitions in the Markov graph rather than with the predictability of individual load instructions. This algorithm uses a prefetch width of two and a prefetch distance of four, generating up to eight prefetch addresses on each invocation of the algorithm. We call this algorithm *Markov* prefetcher.

Finally, we show how our framework can be used to implement prefetching schemes based on very large Markov models. To this end, we implement a first order Markov prefetcher with address correlation containing 256K entries in its second-level table. Each entry contains two Markov graph neighbors. In addition to the delta for the next address, each table entry records the four deltas that last followed the most recent address in MRU order. This organization of the Markov prefetcher is similar to the replicated correlation prefetcher used by Solihin et al. [22]. Each value in the second-level table is associated with a 1-bit confidence counter, similar to our *Markov* algorithm. When a table entry is accessed by the prefetcher, all addresses recorded in this entry are prefetched. Each second-level table entry is configured to fit into an L2 cache line. We call this algorithm *Correlation* prefetcher.

To emulate these prefetching techniques, we manually constructed five different prefetching threads. Table 1 summarizes the properties of these threads. The third column provides the number of instructions in each thread up to the first prefetch instruction. The stride prefetching mechanisms execute different instruction sequences depending on whether the event is associated with a cache miss or an access to a correctly prefetched cache line. The values in parentheses indicate the properties of a thread associated with the access to prefetched data. The fourth column specifies the total number of load instructions, and the last column indicates the length of the longest instruction dependence chain. Note that the number of static and dynamic instructions for each prefetching thread up to the issue of the first prefetch is the same because we removed all branches from the code via loop unrolling and extensive use of conditional move instructions. Global stride prefetching requires

Table 2: Simulated processor parameters

Processor	
Fetch/decode/commit	4/4/4 instructions per cycle
I-window/ROB/LSQ size	64/128/64 entries
Int/FP registers	184
LdSt/Int/FP units	2/4/2
Branch predictor	16K-entry bimodal/gshare hybrid
BTB	2K-entry, 4-way associative
RAS entries	16
Misprediction penalty	minimum 12 cycles
Memory Subsystem	
Cache sizes	64KB L1, 64KB DL1, 2MB L2
Cache associativity	2-way L1, 8-way L2
Cache latencies	3 cyc L1, 12 cyc L2
Cache line sizes	64B L1, 64B L2
Cache MSHRs	16 L1, 24 L2
Main memory latency	minimum 400 cycles
Memory bus	split-transaction, 8B-wide, 1/4 CPU frequency; contention, queuing, bandwidth modeled

Table 3: Benchmark suite details

App.	perfect speedup	stride covorg	App.	perfect speedup	stride covorg
bzip2	24.5	70.4	fma3d	217.4	81.4
crafty	2.1	1.3	galgel	2.0	83.1
eon	0.2	35.2	lucas	431.6	94.1
gap	24.6	62.2	mesa	23.0	90.4
gcc	30.3	45.7	mgrid	203.2	93.3
gzip	3.3	95.2	sixtrack	4.8	49.5
mcf	1399.6	52.5	swim	689.8	99.6
parser	103.5	71.6	wupwise	134.7	53.3
perlbnk	7.4	45.7	bh	0.2	80.1
twolf	1.8	15.6	bisort	27.9	19.3
vortex	34.6	26.3	em3d	28.2	74.1
vpr	120.5	5.4	health	834.3	2.2
ammp	28.7	22.1	mst	222.7	22.8
applu	198.4	77.5	perimeter	90.6	55.3
apsi	10.8	91.1	power	0.02	49.3
art	183.5	97.3	treeadd	158.6	95.4
equake	675.6	91.3	tsp	13.9	54.8
facerec	178.5	53.7	voronoi	35.4	27.6

the most instructions due to the large number of comparisons when the global history is searched for two repeating strides. The next section evaluates how well these prefetching threads work.

4. EVALUATION METHODOLOGY

We evaluate our approach using an extended version of the SimpleScalar v4.0 simulator [10]. The baseline is a two-way CMP consisting of two identical four-wide dynamic superscalar cores that are similar to an Alpha 21264 (Table 2). The minimum memory latency for the baseline processor is 400 cycles. We model bandwidth and contention on the

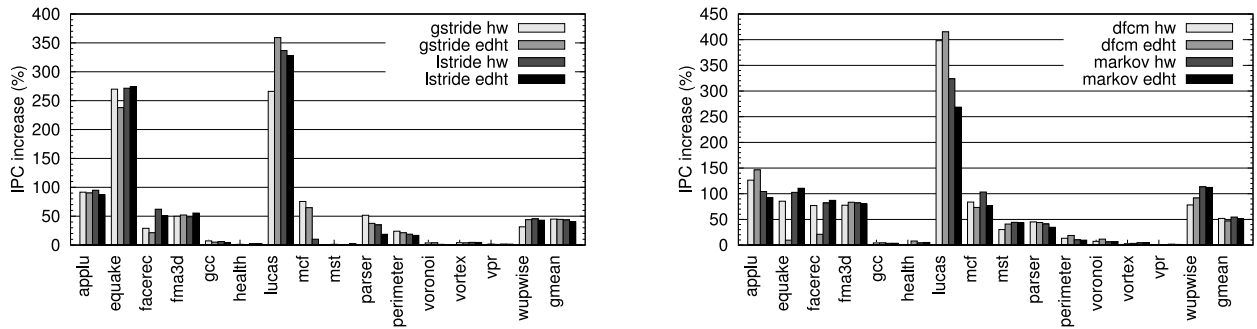


Figure 5: Performance of hardware prefetchers and their EDHT counterparts

memory bus and limit the number of outstanding bus transactions to 32. We used CACTI 3.2 [21] to determine the simulated cache access latencies. The event buffer contains 20 entries. The communication latency between each core and event buffer is 5 cycles. In all modeled configurations we assume that one of the cores in the CMP can be used for executing a prefetching thread. All evaluated prefetching algorithms monitor L2 cache misses and issue prefetch requests for the L2 cache only.

To perform our evaluations, we considered all 36 programs from the SPEC CPU2000 and Olden benchmark suites [6, 13]. Table 3 provides information related to cache misses in every program. The first column for each program shows the speedup with a perfect L2 cache over the baseline without prefetching. In this study, we define a program to be memory-bound if it experiences more than 30% speedup with a perfect L2 cache. Since prefetching techniques target memory latencies, we eliminate from this evaluation the 17 programs that are not memory-bound. The second column specifies the percentage of original cache misses that can be prefetched by a hardware global stride prefetcher. We further eliminate all regular programs that are easily prefetchable by a simple hardware stride prefetcher and thus would get little benefit from a complex prefetching algorithm (*art*, *mgrid*, *swim*, and *treeadd*).

The SPEC CPU2000 programs are run with the SPEC-provided reference inputs. If multiple reference inputs are given, we simulate the corresponding programs with up to the first three inputs and average the results from the different runs. We simulate *vpr* with only one of the reference inputs (routing) because SimpleScalar could not simulate it correctly with the other input (placement). The Olden programs are run with the parameters commonly used in previous research on prefetching [13]. The C programs were compiled with Compaq’s C compiler V6.3-025 using “-O3 -arch ev67 -non_shared” plus feedback optimization. The C++ and Fortran 77 programs were compiled with g++/g77 V3.3 using “-O3 -static”. The Fortran 90 programs were compiled with Compaq’s f90 compiler V5.3-915. We use the SimPoint 3.1 toolset to identify representative simulation points. Each program is simulated for 500 million instructions after fast-forwarding past the number of instructions determined by SimPoint.

5. EXPERIMENTAL RESULTS

In this section, we experimentally measure the effectiveness of our proposed mechanism. In Section 5.1, we evaluate the performance of various prefetching schemes based

on EDHT and compare the speedups with those of hardware implementations of the same prefetching mechanisms. In Section 5.2, we demonstrate how EDHT-based prefetching can improve single thread performance by combining hardware stride prefetching with Markov EDHT prefetching. Finally, Section 5.3 illustrates how EDHT prefetching compares to two other previously proposed hardware techniques that use extra cores to speed up single threads.

5.1 Prefetching Emulation Performance

In this section, we evaluate and compare emulation-based prefetchers with their conventional, hardware-based counterparts. The baseline machine for this experiment is described in Table 2. We measure the performance of four different prefetching schemes: global stride prefetching (*gstride*), local stride prefetching (*lstride*), differential finite-context method prefetching (*dfcm*), and Markov prefetching (*markov*). The hardware implementations of each scheme are marked with a *hw* identifier, while the EDHT versions have an *edht* identifier after the name of the prefetching algorithm. Figure 5 presents speedups for individual programs as well as the geometric mean. The performance of the stride prefetching techniques is shown in the left panel, *dfcm* and *markov* in the right panel.

The results show that the prefetching techniques used in our study are very effective, attaining significant speedups for the majority of the programs. When the hardware implementations are compared with the EDHT implementations, we find that hardware outperforms helper threads only slightly. In case of stride prefetching, hardware is significantly better (over 5%) only on six out of 15 programs. The results for the *dfcm* and *markov* prefetching schemes are similar. The main reason for the performance difference is the delayed issue of the prefetch requests by the helper thread. Helper threads are triggered only when delinquent load instructions commit and they take longer to compute and issue prefetches. Hardware prefetchers initiate the prefetching algorithm as soon as a delinquent load instruction issues and generate prefetch requests much faster. A high frequency of cache miss events is another reason for decreased speedup. In case of *dfcm* prefetching, a high frequency of cache misses in *quake* and *facerec* causes 86% and 60% of the cache miss events to be dropped. EDHT *markov* prefetching faces similar problems in *facerec* and *wupwise*.

Surprisingly, some programs exhibit higher speedups with EDHT prefetching. Hardware prefetchers observe cache misses in the issue order of the load instructions, while EDHT threads observe the sequence of cache miss events

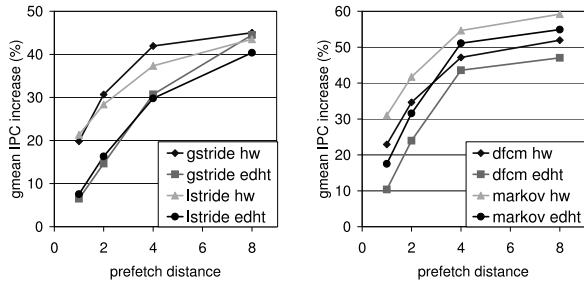


Figure 6: Sensitivity to prefetch distance for stride and markov prefetchers

in commit order. Commit order allows to get a more precise cache miss history. In addition, EDHT threads never suffer from cache miss history pollution caused by wrong-path loads.

It might be unexpected that the performance of the hardware and software implementations of these prefetching algorithms differs by so little. After all, hardware prefetches are issued at least 400 cycles earlier than software prefetches. It seems unlikely that such a big delay would have so little impact of performance.

Figure 6 explains this phenomenon. It illustrates the average speedup obtained by the prefetching schemes as the prefetch distance is varied from 1 to 8. The most interesting feature of this graph is how the performance difference between each hardware and EDHT pair rapidly decreases with increasing prefetch distance. For example, with a prefetch distance of 1 the average speedup for a hardware global stride prefetcher is 20%. The corresponding EDHT prefetcher achieves only 7% speedup. However, prefetching with higher distances decreases the relative performance gap. At a prefetch distance of 8 both hardware and EDHT prefetching perform almost the same. Higher prefetch distances provide timelier prefetches and at some point it does not matter whether prefetch requests are issued with a 400-cycle delay or not. They are still issued early enough to mask the full memory latency. Therefore, a high prefetch distance is the key to good performance of software prefetching.

5.2 Combining Hardware and EDHT Prefetching

Current high-performance microprocessors already include some form of stride prefetching. More complicated prefetching schemes are typically not implemented because of algorithm complexity and/or large storage requirements. EDHT offers an attractive alternative implementation of complex prefetching schemes. In this section we investigate how hardware prefetching can be combined with a more complex prefetching scheme implemented as an EDHT thread. The base machine for this experiment includes the hardware global stride prefetcher described in Table 1. We measure the performance of four different EDHT prefetching schemes: local stride prefetching (*lstride*), differential finite-context method prefetching (*dpcm*), Markov prefetching (*markov*), and duplicated correlation prefetching (*correlation*). Figure 7 shows the program speedups relative to the *gstride hw* baseline.

The results show that adding a local stride prefetcher in most cases provides little additional benefit. Only *lucas* and *wupwise* experience a speedup of around 10%. On the other

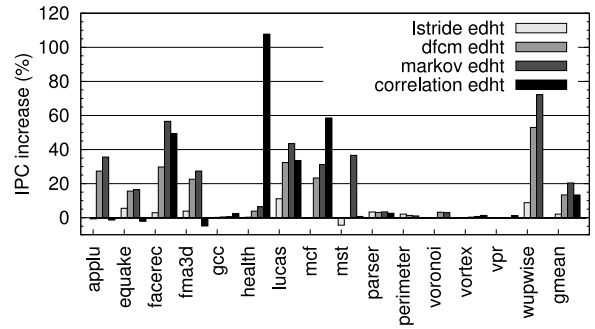


Figure 7: Speedup provided by different EDHT prefetching mechanisms over a baseline with hardware stride prefetching

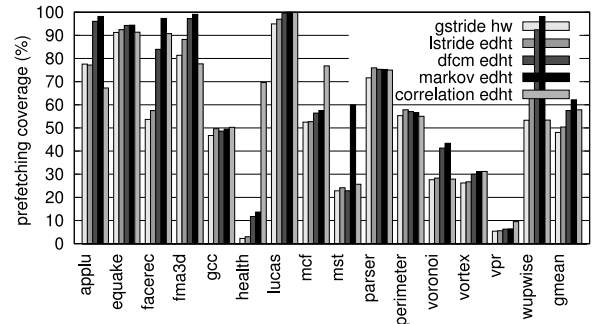


Figure 8: Prefetching coverage

hand, *markov* EDHTs deliver significant speedups for eight out of the 15 programs used in our study. The performance of the *dpcm* prefetcher is similar to *markov*, except on *mst*. The *correlation* prefetcher performs well for four programs. It is especially successful with *health* and *mcf*, where it significantly outperforms all other prefetching algorithms. On average, local stride prefetching improves performance by 2%, DFCM by 13%, markov by 20%, and correlation by 13%.

To gain additional insight about the prefetching activity, we measure the prefetch coverage of the different algorithms. We define prefetch coverage as the ratio of the total number of useful prefetches to the total number of L2 cache misses incurred by the application. Figure 8 illustrates that the speedup numbers of each algorithm largely correspond to their prefetch coverages. The *markov* EDHT is very successful on all SPECfp programs where it pushes the prefetch coverage to almost 100%. It also works well on the Olden programs *mst* and *voronoi*. The correlation prefetcher is at its best on *health*, where it increases the prefetch coverage from 2% to 70%. The integer programs from the SPECint benchmark suite prove to be the toughest targets for prefetching since *mcf* is the only integer program that experiences a significant coverage increase.

Overall, this section demonstrates that the combination of hardware stride prefetching and more complex prefetching mechanisms implemented in our EDHT framework can yield significant performance improvements for a wide variety of programs. Both approaches exhibit significant synergy, as the hardware prefetcher detects and prefetches simple patterns, causing only undetectable patterns to be exposed to the more complex prefetching algorithms. In addition, we

also observe that some algorithms suit applications much better than others. This further justifies the use of EDHT-based prefetching as prefetching threads can be easily customized on a per-program basis.

5.3 Comparison with Other Multi-Core Prefetching Techniques

The previous subsections showed that emulating hardware prefetchers as EDHTs is quite effective and provides significant speedups over the baseline with or without a hardware stride prefetcher. In this section, we compare our mechanism with two other recently proposed hardware-only techniques that use an extra core of a CMP to speed up single threads.

First, we implement and evaluate Future Execution (FE) [4]. FE dynamically creates prefetching threads by directing a copy of the stream of committed instructions to a helper core. On the way to the helper core, a value predictor modifies this stream to compute the results that these instructions are likely to produce during their n^{th} next dynamic execution. Executing this modified instruction stream on another core computes predictions for the future data addresses and issues prefetches into the shared memory hierarchy. We implemented FE with a 4K-entry stride-two-delta hardware value predictor and a buffering capacity of 100 instructions between the cores.

Second, we evaluate the performance of the dual-core execution (DCE) architecture [26]. Instead of using the idle core to run specialized prefetching threads, this technique uses it to launch and execute a copy of the original program in runahead mode [15]. This runahead thread attempts to follow the program path and to execute all instructions that are not dependent on the load instructions that miss in the cache. Thus, it effectively extends the instruction window and allows to issue load requests for data that may be needed in the near future. The non-speculative core re-executes all instructions committed by the runahead core and makes sure that the program execution stays on the correct path. We implement a variation of DCE with a 2K-entry result queue between the cores, a 4KB runahead cache, and an optimistic 1-cycle latency to copy architectural state between the cores.

Note that both FE and DCE impose much higher hardware requirements and complexity than the EDHT framework. FE needs a prediction table and a high-bandwidth communication link between the cores. DCE requires hardware support for a large result queue, a runahead cache, and misprediction recovery logic. Both FE and DCE also require special multiplexing support to fetch instructions from another core (in addition to from the instruction cache).

Figure 9 shows the speedup of the different techniques relative to the *gstride hw* baseline. We chose the *markov edht* algorithm to represent EDHT prefetching since it performs the best on average. Out of the 15 programs used in our study, EDHT performs the best on three programs and DCE provides a significant performance lead (of over 5%) on five. On average, the EDHT Markov prefetcher delivers 20% speedup, the FE technique improves performance by 16%, and DCE shows the best average speedup of 24%. EDHT compares favorably with FE and DCE on floating-point programs, but it does not perform as well on the integer applications. While future execution has the lowest average speedup, it performs very consistently across both the integer and the floating-point applications. One of the explanations for DCE's performance advantage on the

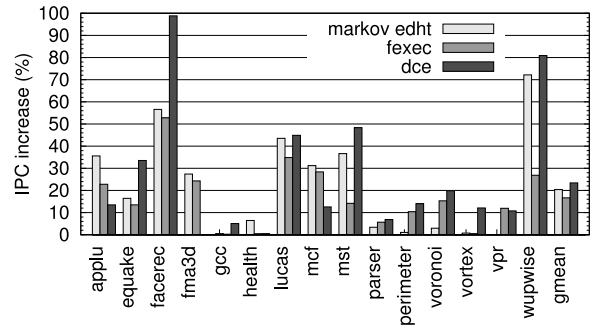


Figure 9: Speedup provided by different multi-core prefetching mechanisms over a baseline with stride prefetching

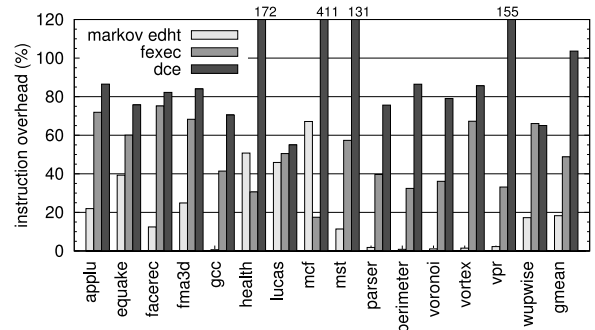


Figure 10: Instruction overhead

integer programs is its ability to significantly decrease the penalty of the branch mispredictions observed by the target thread. Both EDHT prefetching and future execution can only reduce load latencies. Nevertheless, if we exploit the customization capability of EDHT prefetching and use *correlation* threads for *health* and *mcf*, the geometric mean speedup of EDHT prefetching increases to 27% and thus exceeds that of both FE and DCE.

Figures 10 and 11 provide additional insight about the operation of the evaluated techniques. Figure 10 shows the number of instructions issued in the helper core relative to the total number of instructions issued in the regular core. In case of DCE, we measured this parameter as the total increase in the number of issued instructions compared to single-core execution. In almost all programs EDHT executes the least number of instructions. The FE overhead varies from 18% to 70%, while DCE in four cases increases the total number of executed instructions by more than a factor of two. This happens mainly due to fetching and executing many instructions along a mispredicted branch path. With the exception of *wupwise*, DCE always executes more instructions than EDHT or FE. On average, the DCE technique executes about 104% extra instructions, compared to 48% for future execution and 18% for *markov* EDHT.

Figure 11 estimates how often the helper core is busy. In case of EDHT, we assume that the helper core is idle if it is fully stalled waiting for the next cache miss event to occur. For FE and DCE, the idle periods correspond to the cycles when the helper core's ROB is empty. DCE keeps both cores active almost all the time with an average occupancy of 97%. FE is less demanding and the helper core occupancy varies between 10% and 85%, resulting in an average occupancy

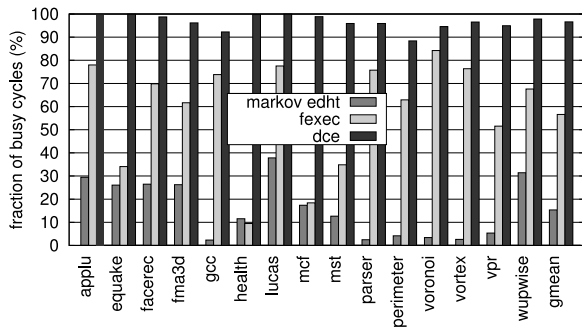


Figure 11: Helper core occupancy

of 56%. EDHT prefetching represents the lightest load on the helper core by activating it on average only 15% of the time. These results highlight another strength of EDHT-based helper threading. By virtue of the event-driven thread communication mechanism, helper threads become active for a short time only when a cache miss occurs. The other two hardware techniques are active all the time or/and have no fast way of exiting helper mode.

The results in this section demonstrate that prefetching based on EDHT can provide performance improvements that are on par with those provided by the dual-core execution paradigm and future execution. At the same time, it requires considerably less complex hardware support and executes fewer instructions, thus keeping the helper core available for other tasks. For example, in a CMP with more than two cores, several EDHTs could time-share on one helper core to prefetch for the regular computation threads running on the other cores.

6. RELATED WORK

Most data prefetching techniques are based on one of two broad classes of predictors - outcome prediction or operation prediction.

Outcome-based prefetchers observe the history of the addresses generated by a program and attempt to detect repeating patterns in the address stream. Once such a pattern is detected, outcome-based prefetchers extrapolate it to predict the addresses that are likely to be referenced in the near future. One of the first hardware prefetchers based on outcome prediction was the stream buffer proposed by Jouppi [8]. Subsequently, a number of other outcome prediction-based prefetching techniques have been introduced. Examples include stride prefetching [3], content-directed prefetching [2], Markov prefetching [7], and prefetching based on other value-prediction techniques [11, 20].

To be effective on a wide range of applications, many of the aforementioned prefetching techniques require relatively large dedicated tables. While several approaches have been proposed to reduce the table sizes of differential Markov predictors [9, 16], these techniques are not applicable to other important Markov-based algorithms. Moreover, hardware designs typically cannot be reconfigured, which is why designers prefer to implement conservative prefetching algorithms that are unlikely to hurt any application. These issues have hampered the introduction of many promising techniques into commercial microprocessors. EDHT represents an architectural framework that allows to use available cores in a CMP to run various prefetching algorithms, elim-

inating the need for dedicated hardware and table space. We further demonstrate that EDHT prefetching works naturally in combination with hardware stride prefetching, where the hardware prefetcher handles the simple patterns and the EDHT thread tackles the more complicated cases. Thus, the EDHT framework provides unique support for customizable prefetcher designs.

Similar to EDHT, Solihin et al. [22] propose to emulate hardware prefetching algorithms in software. Specifically, they employ user-level memory threads (ULMT) for memory-side correlation prefetching that are executed on a processor in the memory controller or in a DRAM chip. We found that PC information is crucial for the effective operation of Markov prefetchers, but PCs are not usually available at the memory interface. Hence, ULMT is algorithmically limited to exploiting only the global cache-miss history. In addition, ULMT observes cache misses in issue order. EDHT, on the other hand, has access to the PCs and observes the misses in program order, thus enabling it to achieve a higher prefetching coverage and accuracy. ULMT faces scalability challenges in multi-core environments where many threads might simultaneously try to access the shared memory. EDHT naturally scales with the number of cores. Finally, our EDHT approach mainly relies on the pre-existing hardware resources of a CMP and conventional user-level threads while ULMT requires a programmable memory processor, OS support to load the prefetching threads into specialized processors, and compiler support for the memory processor’s (most likely different) ISA.

The second class of prefetchers is based on *operation prediction*. Prefetchers of this kind also monitor the program’s address stream. However, instead of detecting the address patterns, they try to identify a sequence of operations that can produce this stream. Speculative precomputation prefetching techniques typically use additional execution pipelines or idle thread contexts in a multithreaded processor to execute helper threads that perform dynamic prefetching for the main thread. Such helper threads can be constructed statically [12, 19, 24] or dynamically by specialized hardware structures [1, 14]. Generally, these techniques create helper threads by extracting program slices that compute critical data addresses. Then they insert triggers for these helper threads into the original program. The execution of the helper threads dynamically precomputes critical data addresses ahead of the original program and issues prefetch requests.

EDHT differs from these approaches in several ways. First, EDHT threads are spawned on architectural events and as such do not require any explicit thread triggers to be inserted into the original program. Second, speculative precomputation threads sometimes require explicit progress synchronization with the main thread during their execution. EDHT threads require no synchronization with the main thread once they are launched. Third, EDHT helper threads can be generated without knowledge about the main program. Therefore, EDHT threads can provide benefits without the need for program analysis or recompilation.

Prefetching based on speculative precomputation can potentially provide a higher prefetching coverage since it is not limited by the predictability of the address stream. However, if one of the load instructions in the helper thread misses in the cache, the entire prefetching thread is stalled and cannot continue prefetching. For example, in a tight loop travers-

ing a long linked list that is stored in memory, speculative precomputation may be of little help. EDHT threads can prefetch such linked lists by dynamically linearizing the list and storing it in main memory. Nevertheless, we believe speculative precomputation and EDHT to be complementary.

Several hardware schemes have been proposed to utilize idle cores of a CMP to speed up single-threaded programs [4, 23, 26]. In this paper, we show that prefetching based on EDHT can provide speedups that are competitive with these approaches while imposing a significantly smaller hardware and execution overhead.

7. CONCLUSIONS

This paper explores the idea of exploiting available cores on a chip multiprocessor to improve the performance of individual program threads. We propose to use extra cores to execute prefetching threads that can emulate the behavior of complex outcome prediction-based prefetching algorithms. However, for this threading technique to be effective, a low overhead mechanism for communicating microarchitectural events is required. To accomplish this, this paper presents the *event-driven helper threading* (EDHT) framework, which uses lightweight hardware support for efficient event communication. EDHT solves many problems that have hampered the introduction of complex outcome-prediction prefetching algorithms into commercial systems. Specifically, the scheme needs minimal hardware modifications, does not need specialized hardware storage for prediction tables, and can be easily reconfigured to tailor prefetching algorithms for individual applications.

Our performance analysis revealed that EDHT-based prefetching provides essentially the same speedup as pure hardware implementations of prefetching algorithms. We further demonstrate that running prefetching EDHTs on top of a baseline with a hardware stride prefetcher yields speedups between 5% and 100% on a wide range of programs. Finally, we implement two other recently proposed hardware techniques for multi-core execution and show that even without customization, EDHT prefetching can provide competitive performance improvements while executing fewer instructions and requiring considerably simpler hardware support.

8. REFERENCES

- [1] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proc. 34th Intl. Symp. on Microarchitecture*, 2001.
- [2] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proc. 10th Intl. Conf. on Architectural support for programming languages and operating systems*, 2002.
- [3] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proc. 25th Intl. Symp. on Microarchitecture*, 1992.
- [4] I. Ganusov and M. Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *Proc. 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2005.
- [5] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential fcm: Increasing value prediction accuracy by improving table usage efficiency. In *Proc. 7th Intl. Symp. on High-Performance Computer Architecture*, 2001.
- [6] <http://www.spec.org/osg/cpu2000/>.
- [7] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proc. 24th Intl. Symp. on Computer architecture*, 1997.
- [8] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th Intl. Symp. on Computer Architecture*, 1990.
- [9] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for tlb prefetching: an application-driven study. In *Proc. 29th Intl. Symp. on Computer Architecture*, 2002.
- [10] E. Larson, S. Chatterjee, and T. Austin. Mase: a novel infrastructure for detailed microarchitectural modeling. In *Proc. Second Intl. Symp. on Performance Analysis of Systems and Software*, 2001.
- [11] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proc. 7th Intl. Conf. on Architectural support for programming languages and operating systems*, 1996.
- [12] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proc. 28th Intl. Symp. on Computer architecture*, 2001.
- [13] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1996.
- [14] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: an implementation of operation-based prediction. In *Proc. 15th Intl. Conf. on Supercomputing*, 2001.
- [15] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proc. Ninth Intl. Symp. on High-Performance Computer Architecture*, 2003.
- [16] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proc. 10th Intl. Symp. on High Performance Computer Architecture*, pages 96–106, 2004.
- [17] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proc. 21st Intl. Symp. on Computer architecture*, 1994.
- [18] J. Rattner. Multi-core to the masses. In *Proc. 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2005.
- [19] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proc. 7th Intl. Symp. on High-Performance Computer Architecture*, 2001.
- [20] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proc. 30th Intl. Symp. on Microarchitecture*, 1997.
- [21] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Tech. report WRL-2001-2, Compaq Western Research Laboratory, December 2001.
- [22] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proc. 29th Intl. Symp. on Computer architecture*, 2002.
- [23] S. T. Srinivasan, H. Akkary, T. Holman, and K. Lai. A minimal dual-core speculative multi-threading architecture. In *Proc. Intl. Conf. on Computer Design*, 2004.
- [24] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen. Helper threads via virtual multithreading. *IEEE Micro*, 24(6), 2004.
- [25] C. Zhang and S. A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *Proc. 14th Intl. Conf. on Supercomputing*, 2000.
- [26] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proc. 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2005.