Physical Experimentation with Prefetching Helper Threads on Intels Hyper-Threaded Processors

Dongkeun Kim[1,4], Steve Shih-wei Liao[1], Perry H. Wang[1], Juan del Cuvillo[2]
Xinmin Tian[2], Xiang Zou[3], Hong Wang[1], Donald Yeung[4], Milind Girkar[2], John P. Shen[1]

Microarchitecture Research Lab[1]
Intel Compiler Labs[2], Desktop Platforms Group[3]
Intel Corporation

Department of ECE
Institute for Advanced Computer Studies
University of Maryland at College Park[4]

{dongkeun.kim, shih-wei.liao, perry.wang, juan.b.del.cuvillo, xinmin.tian, chris.zou, hong.wang, milind.girkar, john.shen}@intel.com, yeung@eng.umd.edu

Abstract

Pre-execution techniques have received much attention as an effective way of prefetching cache blocks to tolerate the ever-increasing memory latency. A number of pre-execution techniques based on hardware, compiler, or both have been proposed and studied extensively by researchers. They report promising results on simulators that model a Simultaneous Multithreading (SMT) processor. In this paper, we apply the helper threading idea on a real multithreaded machine, i.e., Intel Pentium 4 processor with Hyper-Threading Technology, and show that indeed it can provide wall-clock speedup on real silicon. To achieve further performance improvements via helper threads, we investigate three helper threading scenarios that are driven by automated compiler infrastructure, and identify several key challenges and opportunities for novel hardware and software optimizations. Our study shows a program behavior changes dynamically during execution. In addition, the organizations of certain critical hardware structures in the hyper-threaded processors are either shared or partitioned in the multi-threading mode and thus, the tradeoffs regarding resource contention can be intricate. Therefore, it is essential to judiciously invoke helper threads by adapting to the dynamic program behavior so that we can alleviate potential performance degradation due to resource contention. Moreover, since adapting to the dynamic behavior requires frequent thread synchronization, having light-weight thread synchronization mechanisms is important.

1. Introduction

As the speed gap between processor and memory system increases, a processor spends significant amount of time on memory stalls waiting for the arrival of cache blocks. To tolerate the large memory latency, there have been a plethora of proposals for *data prefetching* [4, 15]. Recently, a novel thread-based prefetching technique, called *pre-execution*, has received much attention in the research community [1, 3, 6, 7, 10, 11, 12, 14, 16, 18, 19, 20, 25]. Compared to prediction-based prefetching techniques, pre-execution directly executes a subset of the original program instructions, called a *slice* [24], on separate threads alongside the main computation thread, in order to compute future memory accesses accurately. The prefetch threads run ahead of the main thread and trigger cache misses earlier on its behalf, thereby hiding the memory latency. To be effective, the pre-execution techniques require construction of efficient *helper threads* and processor-level support to allow multiple threads to run concurrently.

Since manual construction of helper threads is cumbersome and error-prone, it is desirable to automate this process. Researchers have proposed a number of systems to generate helper threads automatically. Collins et al. [6] proposed a hardware mechanism to construct helper threads at run-time using a post-retirement queue. Liao et al. [11] developed a post-pass binary adaptation tool to analyze an existing application binary, extract helper threads, and form an augmented new binary by attaching the helper threads to the original binary. Kim and Yeung [10] proposed a compiler framework to generate helper threads at the program source level. All these studies have evaluated the helper threading idea on simulation-based environments that model SMT processors [21] and demonstrated that it is a promising data prefetching technique.

With the advent of Intel Pentium® 4 processor with Hyper-Threading Technology [9, 13], a commercially available multithreaded processor supporting two logical processors simultaneously, it is possible to evaluate the helper threading idea on a physical SMT machine. In this paper, we develop a new optimization module in the Intel pre-production compiler to construct helper threads automatically, and evaluate the helper threading idea on the Intel Pentium 4 processor with Hyper-Threading Technology. To the best of our knowledge, there has been no published work that implements and experiments with helper threads on a real multithreaded processor. We provide the insights gained from our experience and discuss the important ingredients to achieve speedup on such a physical system. Although supporting SMT, the hyper-threaded processor does have unique implementation aspects that are distinct from the proposed design of a research SMT processor [21]. In this study, we find these unique aspects directly influence the tradeoffs for applying helper threads.

To improve the performance of an application program with helper threads, we observe that several key issues need to be addressed. First, the program behavior changes dynamically; for instance, even the same static load incurs different number of cache misses for different time phases. Therefore, a helper thread should be able to detect the dynamic program behavior at run-time. Second, since some of the hardware structures in the hyper-threaded processors are shared or partitioned in the multi-threading mode, resource contention can be an issue, and consequently, helper threads need to be invoked judiciously to avoid potential performance degradation due to resource contention. Lastly, we observe that the dynamic program behavior changes at a fine granularity. To adapt to the dynamic behavior, helper threads need to be activated and synchronized very frequently. Therefore, it is important to have low overhead thread synchronization mechanisms.

Helper threading has a very unique characteristic. Compared to traditional multi-threading where every thread should be executed and committed in pre-defined order to guarantee the correctness of program execution, helper threads only affect the performance of the application. Therefore, a helper thread does not have to be always executed and can be deactivated whenever the helper thread does not improve the performance of the main computation thread. This property of helper threads provides interesting opportunities for applying various dynamic optimizations. In particular, since the dynamic program behaviors such as cache misses are generally micro-architectural and not available at compile-time, they can be most effectively captured and adapted to when monitored at run-time. In this paper, we demonstrate that, in order to obtain significant benefit from helper threading, it is essential to employ run-time mechanisms to throttle helper threads dynamically at a fine granularity.

The rest of the paper is organized as follows. Section 2 presents the software infrastructures that enable our experiments. Section 3 describes our experimental framework and Section 4 details the three helper threading scenarios. Section 5 discusses the experimental results on an Intel Pentium 4 processor with Hyper-Threading Technology. Section 6 summarizes the key observations gained from the experiments. Section 7 discusses the related works and Section 8 concludes.

2. Software infrastructures for experiments

We develop two software infrastructures for our experiments. Section 2.1 describes the design of an optimizing compiler module to automatically construct helper threads and Section 2.2 presents user-level library routines for light-weight performance monitoring.

2.1. Compiler to construct helper threads

For pre-execution techniques to be widely used, it is necessary to automate the helper thread construction process. In this section, we describe the design of an optimization module built in the Intel compiler infrastructure. In the compiler, the helper thread construction consists of several steps. First, the loads that incur a large number of cache misses and also account for a large fraction of the total execution time are identified. After selection of an appropriate loop level surrounding the cache-missing loads, the pre-computation slices within that loop boundary are extracted and the live-in variables are identified as well. Finally, appropriate trigger points are placed in the original program and the helper thread codes are generated.

2.1.1. Delinquent load identification. The first step in the helper thread generation is to identify the top cache-missing loads, known as delinquent loads, through profile feedback. This is achieved by running the application on the Intel VTune^{TM} performance analyzer [23] to collect the clock cycles and L2 cache misses. The application program is profiled with the same input set as is used in the later experiments. However, we find different input sets do not usually affect the set of the identified delinquent loads. After reading in the VTune profiles, the compiler back-end optimizer correlates the data with its intermediate representation using source line numbers. From analyzing the profile information, the compiler module identifies the delinquent loads and also keeps track of the cycle costs associated with those delinquent loads, i.e., memory stall time. Finally, the delinquent loads that account for a large portion of the entire execution time are selected to be targeted by helper threads.

2.1.2. Loop selection. Once the target delinquent loads are identified, the compiler module forms a region within which the helper thread will be constructed. Previous research has shown that delinquent loads most likely occur within a heavily traversed loop nest [10, 11]. Thus, loop structures can naturally be used for picking an appropriate region. As to be further elaborated in Section 3.1.3, the cost of thread activation and synchronization in a real system is in the range of thousands of cycles. Therefore, a key criterion in selecting a proper loop candidate is to minimize the overhead of thread management. One goal is to minimize the number of helper thread invocations, which can be accomplished by ensuring the trip count of the outer-loop that encompasses the candidate loop is small. A complementary goal is to ensure that the helper thread, once invoked, runs for an adequate number of cycles in order to amortize the thread activation cost. Therefore, it is desirable to choose a loop that iterates a reasonably large number of times. In our loop selection algorithm, the analysis starts from the innermost loop that contains the delinquent loads and keeps searching for the next outer-loop until the loop trip-count exceeds a threshold (currently one thousand iterations) and the next outer-loop's trip-count is less than twice the trip-count of the currently processed loop. On the other hand, when the analysis reaches the outermost loop within the procedure boundary, the search ends and the loop is selected. The loop trip-count information is collected via VTune's instruction count profile. We observe this simple algorithm is effective, especially in the presence of the aggressive function inlining performed by the compiler.

2.1.3. Slicing. Next, the compiler module identifies the instructions to be executed in the helper threads and this process is enabled by slicing. We perform program slicing that is similar to backward slicing shown in [26]. Within the selected loop, the compiler module starts from a delinquent load and traverses the dependence edges backwards. To construct efficient and light-weighted helper threads, only the statements that affect the address computation of the delinquent load, including the change of the control flow, are selected as a slice and everything else is filtered out. Since helper threads should not affect the correctness of the main thread, all the stores to heap objects or global variables are removed from the slice. Once slicing is completed for each individual delinquent load, the extracted slices are merged to form a single thread of instruction sequence to target all the delinquent loads within the same loop boundary.

2.1.4. Live-in variable identification. Once a helper thread is formed, the live-in variables to the helper thread are identified. Through analysis, the variables used in the selected loop are divided into two groups: those that have upwards-exposed reads and those that are privatizable. The variables in the former group are selected as live-ins and will be explicitly passed through global variables that are declared solely for the use of helper threads.

2.1.5. Code generation. After the compiler analysis phases, the constructed helper threads are attached to the application program as a separate code. In addition, the codes to create, schedule, and invoke the helper thread are inserted as well. In this study, two static thread invocation strategies are investigated. One is a loop-based trigger where a helper thread is invoked at the entrance of the targeted loop. The other is a sample-based trigger where a helper thread is invoked once every few loop iterations of the targeted loop. Section 4 provides more details about our thread invocation schemes. For a given application, multiple helper threads can be constructed, each targeting different delinquent loads at different loop regions. However, since there are only two logical processors on the Pentium 4 hyper-threaded processor, it is essential to reduce the thread switching overhead. To do so, the compiler creates only one OS thread at the beginning of the program and recycles the thread to target multiple loops. The created helper thread enters a dispatcher loop and sleeps if there is no helper task to dispatch to. Whenever the main thread encounters a trigger point, it first passes the function pointer of the corresponding helper thread and the live-in variables, and wakes up the helper thread. After being woken up, the helper thread indirectly jumps to the designated helper thread code region, reads in the live-ins, and starts execution. When the execution ends, the helper thread returns back to the dispatcher loop and goes to sleep again.

2.2. EmonLite: User-level library routines for performance monitoring

In order to ensure a helper thread can adapt to the dynamic program behavior, we need a light-weight mechanism to monitor such dynamic events as cache misses and at very fine sampling granularity. Some existing tools, such as *pixie* [17], instrument the program code to use cache simulator to *simulate* the dynamic behavior of a program. However, this methodology may not accurately reflect the actual program behavior on a physical system and often slows down the program execution. To remedy these issues, we introduce a library of user-level routines, called *EmonLite*, which performs light-weight profiling through the direct use of the performance monitoring events supported on the Intel processors. Provided as a library of compiler intrinsics, EmonLite allows a compiler to instrument at any location of the program code to directly read from the *Performance Monitoring Counters* (PMCs). Therefore, event statistics such as clock cycles or L2 cache misses can be collected for a selected code region at a very fine granularity with high accuracy and low overhead. The Intel processors support performance monitoring of over 100 different micro-architectural events associated with branch predictor, trace cache, memory, bus, and instruction events. Compiler-based instrumentation via EmonLite enables collection of the chronology of certain instruction's dynamic events for a wide range of applications. Such profiling infrastructure can be leveraged to support dynamic optimizations such as dynamic throttling of both helper thread activation and termination.

2.2.1. EmonLite vs. VTune. The VTune performance analyzer can be used to sample various performance monitoring events over the entire program execution. It provides the profile data ranging in scope from process, to module, procedure, source-line, and even assembly instruction level. However, VTune provides only a summary of sampling profile for the entire program execution and therefore, it is difficult to extract the dynamic chronological behavior of a program from VTune's profile. In contrast, EmonLite provides the chronology of the performance monitoring events and thus, it enables analysis of time-varying behavior of the workload at a fine granularity and can lead to dynamic adaptation and optimization. In addition, EmonLite is a library of user-level routines and can be directly placed in the program source code to discriminately monitor the dynamic micro-architectural behaviors for the judiciously selected code regions. Lastly, while VTune's sampling based profiling relies on the buffer overflow of the PMCs to trigger an event exception handler registered at OS, EmonLite reads the counter values directly from the PMCs by executing four assembly instructions. Consequently, EmonLite is extremely light-weight; it rarely slows down the user program provided that the profiling sampling interval (i.e., how frequently the PMCs are read) is reasonably sized.

2.2.2. Components of EmonLite. The EmonLite library provides two compiler intrinsics that can be easily inserted into a user

```
while (arcin) {
  /* emonlite_sample() */
  if (!(num_iter++ % PROFILE_PERIOD)) {
    cur_val = readpmc(16);
    L2miss[num_sample++] = cur_val - prev_val;
    prev_val = cur_val;
  }

  tail = arcin->tail;
  if (tail->time + arcin->org_cost > latest) {
    arcin = (arc_t *) tail->mark;
    continue;
  }
  ...
}
```

Figure 1. Example of EmonLite code instrumentation - price_out_impl() of MCF.

program code. One of the routines, called `emonlite_begin()`, initializes and programs a set of EMON-related *Machine Specific Registers* (MSRs) to specify the performance monitoring events for which the profiles are collected. This library routine is inserted at the beginning of the user program and is executed only once for the entire program execution. The other intrinsic, called `emonlite_sample()`, reads the counter values from the PMCs and is inserted in the user code of interest. Since the PMC provides performance event information for each logical processor, to ensure accuracy of profiling, the target application is pinned to a specific logical processor via OS affinity API, such as `SetThreadAffinityMask()` in Win32 API.

2.2.3. Implementation issues. Automatic instrumentation of EmonLite library routines is also implemented in the Intel preproduction compiler. For a selected code region, the following steps are taken to generate the chronology of performance monitoring events.

Step 1. Same as previously described compiler analysis phases, the delinquent loads are first identified and appropriate loops are selected.

Step 2. The compiler inserts the instrumentation codes into the user program. In the `main()` function of the user program, it inserts `emonlite_begin()` to initialize EmonLite and set up corresponding PMCs.

Step 3. For each loop that is identified in Step 1, the compiler inserts codes to read the PMC values once every few iterations. Figure 1 shows how instrumentation codes are inserted in a loop in `price_out_impl()` of MCF, a benchmark in the SPEC CINT2000 suite. By varying the profiling interval, PROFILE_PERIOD, the granularity and sensitivity of profiling can be easily adjusted.

2.2.4. Example usage of EmonLite: Chronology of L2 cache miss event. Figure 2 depicts the chronology of the L2 cache miss events for the same loop used in Figure 1. The graph illustrates the dynamic behavior of the L2 cache at a very fine granularity, i.e., 10 loop iterations per sample (around 2K cycles on average). In the figure, it is obvious that there are time fragments where the main thread incurs small or even no L2 cache misses. If a helper thread were launched for a period without any cache misses, it could potentially degrade the performance during this time interval due to hardware resource contention. This observation motivates certain mechanisms to control helper threads dynamically, which is further discussed in Section 5.2.

Figure 3 illustrates another example of chronology at a much coarser granularity for the same loop. The number of L2 cache misses is collected for every 100,000 iterations; each profiling period taking about 14M cycles on average. This figure shows the EmonLite profile for the entire program execution and a phase behavior of L2 cache misses for the program is evident.

Table 1. System configuration

CPU	2.66GHz Intel Pentium 4 with Hyper-Threading Technology
L1 Trace cache	12K micro-ops, 8-way set associative 6 micro-ops per line
L1 Data cache	16KB, 4-way set associative 64B line size, write-through 2-cycle Integer, 4-cycle FP
L2 Unified cache	512KB, 8-way set associative 64B line size, 7-cycle access latency
DTLB	64 entries, fully associative, map 4K page
Load buffer	48 entries
Store buffer	24 entries
Reorder buffer	128 entries
OS	Windows XP Professional, Service Pack 1

3. Experimental framework

Our experiments are performed on a real physical system using a Pentium 4 Processor with Hyper-Threading Technology. In this section, we present the hardware configuration and the benchmarks used in the experiments.

3.1. Hardware System

3.1.1. System configuration. As shown in Table 1, we use a system with a 2.66GHz Intel Pentium 4 processor with Hyper-Threading Technology [9, 13]. The system contains one physical processor, which supports two logical processors simultaneously. For the memory subsystem, the processor contains a trace cache for instructions and a 16KB data cache at the first level, and a 512KB unified cache at the second level. The processor implements a hardware stride prefetcher for prefetching data cache blocks.

3.1.2. Hyper-threaded processors. Unlike a proposed SMT processor [21] where most, if not all, micro-architectural structures are shared between the logical processors, the micro-architectural resources in the Intel hyper-threaded processors are managed differently. As detailed in [13], a hyper-threaded pro-

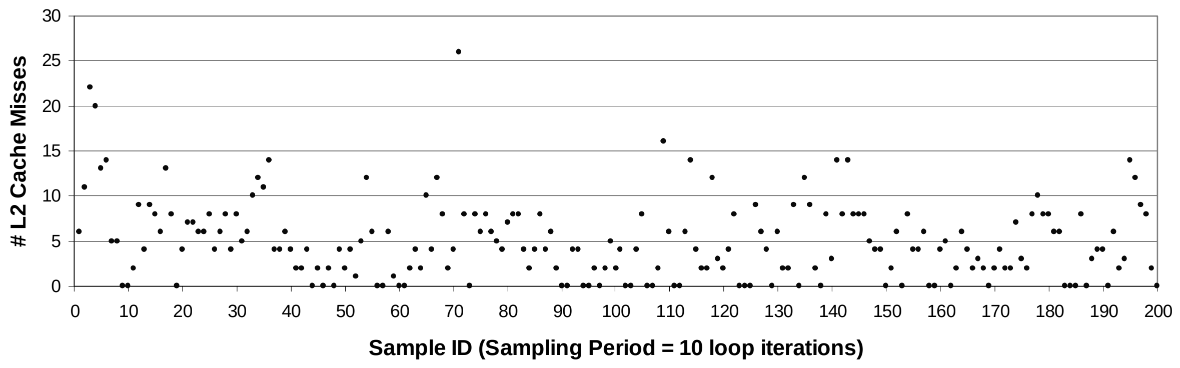


Figure 2. Chronology of L2 cache miss events at a fine granularity - price_out_impl() of MCF

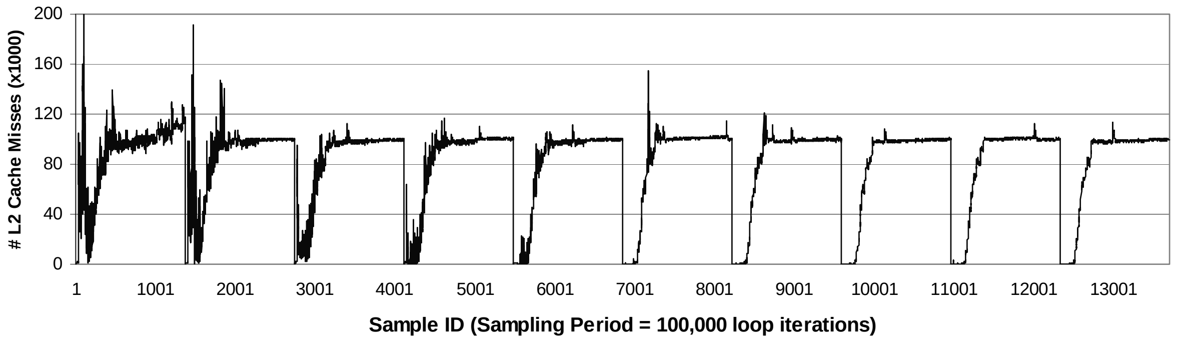


Figure 3. Coarse-grain phase behavior of L2 cache miss events - price_out_impl() of MCF

cessor dynamically operates in one of two modes; in ST (Single Threading) mode, all on-chip resources in Table 1 are given to a single application thread whereas, in MT (Multi-Threading) mode, these resources are shared, duplicated or partitioned between the two logical processors. As shown in Table 2, structures like caches and execution units are shared between the two logical processors, very much like resource sharing on a research SMT processor [21]. On the other hand, structures like the reorder buffer are evenly hard-partitioned to prevent one logical processor from taking up the whole resource. In addition, micro-architectural resources like the ITLB and the return stack buffer are replicated for each logical processor. The transition between ST and MT modes occurs automatically when an application thread on a given logical processor either suspends or gets reactivated. In particular, such transition is primarily due to synchronization between application threads running on the logical processors.

3.1.3. Thread synchronization mechanisms.

In this paper, we compare two mechanisms for invoking and suspending helper threads. First, the Win32 API, `SetEvent()` and `WaitForSingleObject()`, can be used for thread management. When an active thread calls `WaitForSingleObject()`, the Windows scheduler waits until the CPU utilization of the corresponding logical processor falls below 10%. Only then does the OS deschedule the suspended thread and trigger a mode change from MT mode to ST mode. The latency between the moments the thread calls for suspension to the occurrence of the mode transition is non-deterministic and it is between 10K to 30K cycles.

Table 2. Hardware management in Intel hyper-threaded processors

Shared	Trace cache, L1 D-cache, Execution units L2 cache, Global history array, Allocator Microcode ROM, Uop retirement logic IA-32 instruction decode, DTLB Instruction scheduler, Instruction fetch logic
Duplicated	Per logical processor architecture state Instruction pointers, Rename logic, ITLB Streaming buffers, Return stack buffer Branch history buffer
Partitioned	Uop queue, Memory instruction queue Reorder buffer, General instruction queue

To lower the thread switching and synchronization cost, we prototype a hardware mechanism that implements user-level light-weight thread synchronization instructions similar to the lockbox primitives described in [22]. This hardware mechanism is actually implemented in real silicon as an experimental feature. In our hardware synchronization mechanism, a thread can issue a single

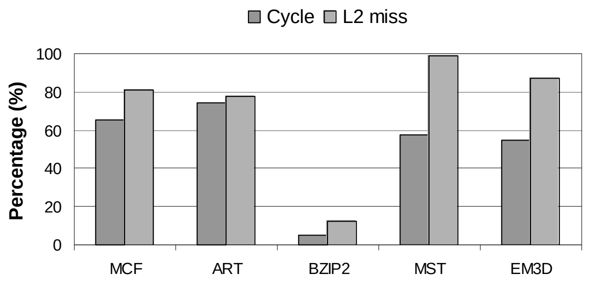


Figure 4. VTune profiles: cycles and L2 cache misses associated with delinquent loads

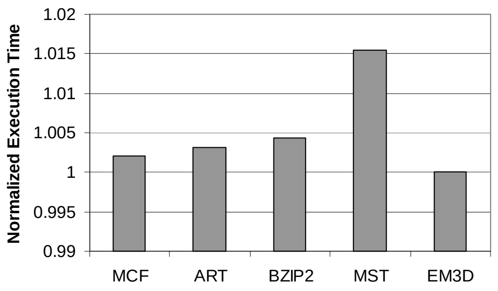


Figure 5. Normalized execution time without thread pinning (no helper threading)

instruction to suspend and can directly cause the mode transition from MT mode to ST mode. Conversely, another thread can issue a single instruction to wake up a suspended thread and cause ST to MT mode transition. Using these direct hardware synchronization primitives, thread suspension takes approximately 1,500 cycles, which achieves one order of magnitude reduction as compared to the cost of OS API. Moreover, this hardware mechanism is entirely transparent to the OS. In Section 5, we evaluate the impact of thread synchronization cost for a number of helper threading scenarios.

3.2. Benchmarks

To decide the benchmarks used in our experiments, we first run entire SPEC CPU2000 benchmark suite [8] on VTune and collect the cycle and L2 cache miss profiles. Then, we select those applications that have significant number of cycles attributed to the L2 cache misses, including MCF and BZIP2 from SPEC CINT2000 suite, and ART from SPEC CFP2000 suite. The reference input sets are used for both profile runs and experiments. In addition, we also pick MST and EM3D from Olden benchmarks [2] for the same reason.

Memory latency tolerance has long been tackled by both micro-architecture techniques and advanced compiler optimizations. For example, the Intel Pentium 4 processor employs a hardware stride prefetcher. In addition, the production compiler can perform various optimizations to reduce the number of cache misses such as cache-conscious code or data layout and access optimizations [5]. Our objective is to tackle those cache misses that remain even after these hardware and compiler techniques are applied, and to achieve additional speedup by using helper threads. The benchmarks are compiled with the best compiler options (-O3 -Qipo -QxW) in the most recent version of the Intel compiler. Then, VTune is used to identify the candidate loads. Figure 4 shows the percentage of the L2 cache misses that are associated with the targeted loads over the total L2 cache misses. It also shows the percentage of the exposed memory stall cycles due to the cache misses over the entire execution time. For every application in the figure, we observed fewer than five loads contribute to a large fraction of the total L2 cache misses, i.e., 83.5% on average for top five delinquent loads. The percentage of memory stall cycles indicates an upper bound on the performance improvement that is achievable with perfect data prefetching for the targeted delinquent loads.

3.3. Baseline

In the hyper-threaded processors, the Windows OS periodically reschedules a user thread on different logical processors. This involves the overhead of OS job-scheduling and can incur more cache misses if multiple physical processors are employed in one system. In the context of helper threading, a user thread and its helper thread, as two OS threads, could potentially compete with each other to be scheduled on the same *logical* processor. In addition, on a multiprocessor configuration with multiple physical processors, an application thread and its helper thread could also be scheduled to run on different *physical* processors. In this case, without shared cache, the prefetches from a helper thread will not be beneficial to the application thread. In order to avoid these undesirable situations, the compiler adds a call to the Win32 API, `SetThreadAffinityMask()`, to manage thread affinity at the beginning of the application thread to pin the main computation thread to a particular logical processor. This is our baseline configuration to evaluate helper threading. Similarly, the helper thread, if present, is pinned to the other logical processor.

Figure 5 shows the effect of thread pinning on the performance of the baseline (no helper threading) binaries. Each bar represents the execution time without thread pinning, which is normalized on the execution time with thread pinning. Clearly, thread pinning slightly improves the single thread performance in the hyper-threaded processors. This is because thread pinning eliminates some overhead of OS job scheduling. Throughout the rest of the paper, we use the execution time *with* thread pinning as the baseline reference performance numbers.

4. Helper threading scenarios

We introduce three helper threading scenarios in this paper. They differ from each other with regard to how the trigger placement is tailored in the code and when the helper threads are triggered at run-time. In this section, we illustrate the interesting tradeoffs and the insights.

4.1. Static: Loop-based trigger

The first helper threading scenario is called *Loop-based Static Trigger*. In this scenario, a helper thread is activated at the entrance point of the targeted loop. Once the helper thread is woken up, it runs through all iterations of the loop without any further intermediate synchronization. In other words, the inter-thread synchronization only occurs *once for every instance of the targeted loop*.

Since this approach implements the trigger placement at the well-defined program structures, it is simple to implement. It can be useful if the thread synchronization cost is high. However, due to the lack of intermediate synchronization between the main thread and the helper thread throughout the computation of the targeted loop, the helper thread can either run too far ahead of the main thread and pollute the cache, or run behind the main thread and waste computation resources which could be more effectively used by the main thread.

4.2. Static: Sample-based trigger

To avoid run-away helper threads or run-behind helper threads, it is necessary to perform inter-thread synchronization at finer granularity than the size of the targeted loop. This motivates the second helper threading scenario, called *Sample-based Static Trigger*, where a helper thread is invoked *once for every few iterations of the targeted loop*. The number of loop iterations between two consecutive helper thread invocations becomes the sampling period. In this approach, once the helper thread is activated, it executes either for the number of loop iterations equal to the size of the sampling period, or until it reaches the termination of the loop.

Since the helper threads in this scenario are invoked more frequently than the loop-based static trigger, the effectiveness is more sensitive to the thread synchronization cost. Furthermore, this scenario requires additional code to be instrumented in the targeted loop to check how far the main thread or the helper thread has executed within the sampling period iterations. In effect, this approach relies upon the sampling period as the synchronization boundary to frequently cross-check relative progress between the helper thread and the main thread. The size of the sampling period binds the distance by which a helper thread can run ahead of or behind the main thread. Therefore, the effectiveness of this approach depends on the choice of the sampling period at compile-time.

4.3. Dynamic trigger

As illustrated in Figure 2, a program's dynamic behavior varies at different chronological phases. Consequently, helper threads may not always be beneficial. Even when the main thread suffers long latency cache misses, the effectiveness of helper threads still depends on a variety of resource related issues such as the availability of execution units, occupancy of the reorder buffer, the number of cache misses, memory bus utilization, or fill buffer (i.e., Miss Status Holding Register or MSHR) usage.

To adapt to the dynamic program behavior and avoid activating a helper thread when it is not needed, we evaluate a simple dynamic helper threading scenario, which is based on the sample-based trigger presented in Section 4.2. However, rather than invoking a helper thread for every sample instance, the main thread dynamically decides whether or not to invoke a helper thread for a particular sample period. Effectively, this is a dynamic throttling scheme which dynamically monitors the relative progress and resource contentions between the main thread and its helper thread and then applies judicious control on both activation and termination of the helper thread.

Table 3. Statistics for sample-based trigger

Application	Procedure Name	Sampling Period	# Samples
MCF	refresh_potential	100	2422827
MCF	price_out_impl	1000	1370258
ART	match	1000	1672740
BZIP2	sortIt	1000	118201
MST	BlueRule	100	44985
EM3D	all_compute	200	20000

5. Performance evaluation

We experiment with compiler-generated helper threads on real silicon, and present the performance results and analysis from the experiments. Section 5.1 evaluates the two static trigger scenarios and Section 5.2 investigates the performance potential of the dynamic trigger scenario. Section 5.3 gauges the impact of thread synchronization cost and discusses the need for lighter-weight mechanisms.

5.1. Evaluation of static trigger

5.1.1. Statistics for sample-based trigger. To evaluate the sample-based static trigger scenario, the sampling period should be determined a priori. The compiler instruments each targeted loop with EmonLite library routines to profile the chronology of cycles and L2 cache misses. The sampling period is adjusted such that each sample takes between 100K and 200K cycles on average. Recall that the Windows API-based thread synchronization mechanisms cost between 10K and 30K cycles, whereas the prototype hardware-based synchronization mechanism takes about 1,500 cycles. Table 3 lists the procedure name that contains the targeted loop, the sampling period in loop iterations, and the number of samples over the entire program execution for each selected loop. In each benchmark except MCF, a loop that accounts for the largest fraction of the memory stall time is selected. In MCF, two loops that largely suffer from the cache misses are chosen.

5.1.2. Speedup results. Figure 6 reports the speedup results of the two static trigger scenarios. For each scenario, we compare the

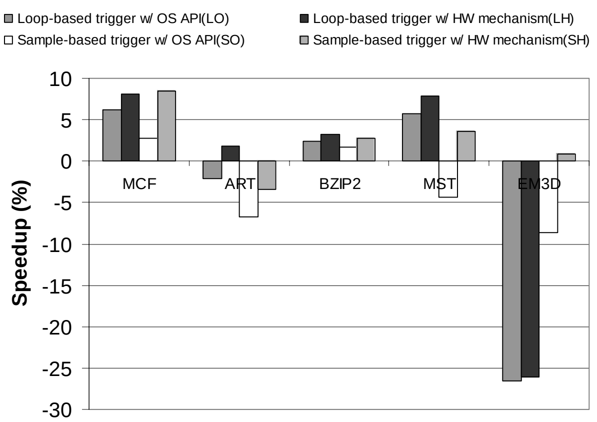


Figure 6. Speedup of static trigger

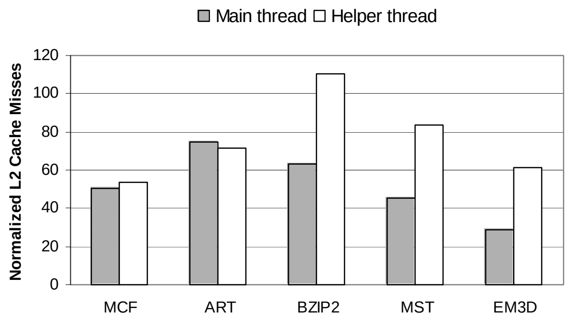


Figure 7. Cache miss coverage: Loop-based trigger with HW synchronization mechanism

performance of two thread synchronization mechanisms, heavy-weight Windows API and light-weight hardware mechanism. The speedup is for the entire program execution, not just for the targeted loop. For each application, we show speedups in percentage for four different configurations, LO, LH, SO, and SH as shown in the figure.

The difference in performance impact by the two thread synchronization mechanisms is rather pronounced. On one hand, for the loop-based trigger, i.e., LO vs. LH, the light-weight hardware thread synchronization mechanism only provides 1.8%, on average, more speedup than OS API. This is primarily due to the targeted loops and their corresponding helper threads run for many iterations before the next synchronization at loop boundary, thus the startup synchronization cost is much less significant, even assuming the cost of OS API. On the other hand, for the sample-based trigger, i.e., SO vs. SH, the hardware thread synchronization mechanism achieves 5.5% additional gain on average. Since the helper threads are activated more frequently in sample-based trigger scenario, the effectiveness is much more sensitive to the thread synchronization overhead. The heavy-weight OS API introduces significant overhead on the main thread and potentially causes helper thread to be activated out of phase, thus resulting in ineffectual pre-computation which not only runs behind but also takes away critical processor resources from the main thread. This explains the slowdown in SO for most benchmarks except MCF, which suffers from lots of long latency cache misses.

Comparing the performance of loop-based trigger and sample-based trigger, the loop-based trigger performs slightly better than sample-based trigger for the current sampling period and thread synchronization cost. Using OS API for thread synchronization, i.e., LO vs. SO, the loop-based trigger outperforms the sample-based trigger for all applications except EM3D. In EM3D, with the loop-based trigger, the helper thread runs away from the main thread due to the lack of synchronization, and the memory accesses in the run-away helper thread causes excessive cache pollution. However, the sample-based trigger can effectively prevent cache thrashing, thereby providing better performance for EM3D. For the other applications, since the sample-based trigger invokes helper threads more frequently, the heavy-weight overhead of calling OS API significantly affects the main threads performance.

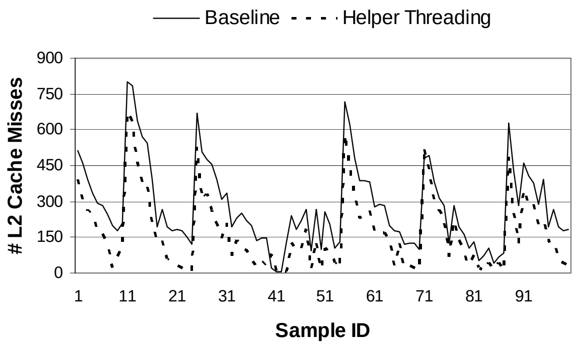
On the other hand, if the light-weight hardware thread synchronization mechanism is employed, i.e., LH vs. SH, the performance with the sample-based trigger is comparable to that of the loop-based trigger. In MCF and BZIP2, there is little difference between these two scenarios. On the other hand, since the targeted loop in ART consists of only 12 instructions, instrumentation code accounts for a relatively large portion of the loop, resulting in the performance degradation with the sample-based trigger. In MST, the sample-based trigger performs worse due to both the thread synchronization cost and the code instrumentation overhead. Though a reduction by an order of magnitude from OS API's overhead, even at 1,500 cycles, the hardware synchronization still takes more than 2 times as long as the latency to serve a cache miss to the main memory. As the thread synchronization cost becomes even lower, the sample-based trigger is expected to be more effective.

5.1.3. Dynamic behavior of helper threading.

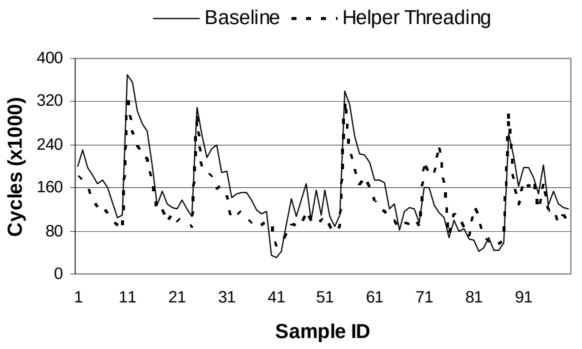
In order to further shed insights on the tradeoffs of helper thread, we investigate the dynamic behaviors of helper thread effectiveness, in terms of reduction in both cache miss coverage and cycle count improvement.

Figure 7 illustrates the L2 cache miss coverage based on the VTune profile for the loop-based static trigger with hardware thread synchronization mechanism (configuration LH from Figure 6). In the graph, we show, within the targeted loop, the cache miss counts incurred by both the delinquent loads in the main thread and prefetches in the helper thread for those loads. Those cache miss counts are normalized on that of the baseline for the same set of delinquent loads. The data clearly indicates that helper threading can achieve significant reduction in cache misses in the main thread, ranging from 25.3% in ART to 60.4% in EM3D. In EM3D, helper threads eliminate a large portion of L2 cache misses for the targeted loads. However, they significantly increase the number of cache misses for the non-targeted loads, which is not shown in Figure 7, thereby degrading the overall performance.

In addition, this graph also reveals some inefficiency of the



(a) L2 cache miss event



(b) Cycle event

Figure 8. Dynamic behavior of performance events with and without helper threading

static trigger helper threading scenarios. First, in all benchmarks except for BZIP2, the percentage of the cache misses covered by the helper thread is not close to 100%. This indicates that the helper thread sometimes runs behind the main thread and thus, the helper thread should not be activated for certain period. Second, in ART, BZIP2, MST, and EM3D, the sum of the two bars exceeds 100%. This indicates that, for certain time phases, the helper thread runs too far ahead of the main thread and incurs cache thrashing. During these time phases when the helper threads are not effective, dynamic throttling can be introduced to either suspend an on-going helper thread or prevent activating the next helper thread instance.

Figure 8 shows the EmonLite-based chronology of the L2 cache miss events and the cycle events of BZIP2 for 100 samples using a sample-based static trigger scenario. Each graph depicts two sets of data, one without helper thread (Baseline), and the other with helper thread. Comparing the patterns in Figure 8(a) and 8(b), there exists strong correlation between the L2 cache miss event and the cycle event, which implies that those targeted loads triggering the L2 cache misses are likely critical. However, when helper threading is applied, there are some sample phases when L2 cache miss reductions do not convert to similar reductions in cycle counts. For instance, between sample ID of 71 and

Table 4. Percentage SD statistics

Application	% SD(cycle)	% SD(L2 miss)
MCF (refresh)	44.80%	39.45%
MCF (implicit)	44.50%	50.41%
ART	17.14%	3.79%
BZIP2	61.91%	96.83%
MST	30.59%	30.86%
EM3D	46.05%	44.72%

88, even though the number of L2 cache misses is reduced with helper threads, the cycle counts actually increase. Therefore, it would be helpful to detect the time phases when the application performance is degraded so that helper threads do not get activated during those phases. This observation leads us to consider certain run-time mechanisms to dynamically throttle helper threads, a topic to be discussed in the next section.

5.2. Evaluation of dynamic trigger

In this section, we explore the potential of dynamic throttling of helper threads assuming perfect throttling mechanisms.

5.2.1. Quantifying dynamic behavior. To quantify the dynamic behavior of the targeted loops, using the same sampling period shown in Table 3, we profile cycle and L2 cache miss events for each sample without helper threading. Then, we compute the percentage standard deviation (SD) of the cycles and the L2 cache misses among all the samples as shown in the following equation, where PMC(i) is the PMC value for the i-th sample, A is the average PMC value per sample, and N is the total number of samples.

$$SD(\%) = \sqrt{\frac{\sum_i (PMC(i) - A)^2}{N}} * 100/A$$

Table 4 reports the percentage SD values for cycles and L2 cache misses. A large SD value implies the performance event is more time-variant dynamically. Again, there exists some correlation between the cycle event and L2 cache miss event on the SD. Or rather, if one performance monitoring event is dynamically variant, so is the other one. This is because the delinquent loads in our targeted loops are usually on the critical path and thus, the cache miss behavior directly affects the cycle count behavior.

5.2.2. Performance potential with perfect throttling. To evaluate the dynamic throttling mechanisms, it is essential to gauge when the helper thread improves or degrades the performance of the main thread. This can be done by comparing the cycles with and without helper threads. For a limit study, an ideal scenario is to activate the helper thread when it is beneficial and deactivate it when it degrades performance. Thus, the first step is to collect the EmonLite profiles, using the sample-based static trigger, for every sampling period with and without helper threads. Using the samples without helper threads as baseline, the cycle counts of the samples that show performance improvements with

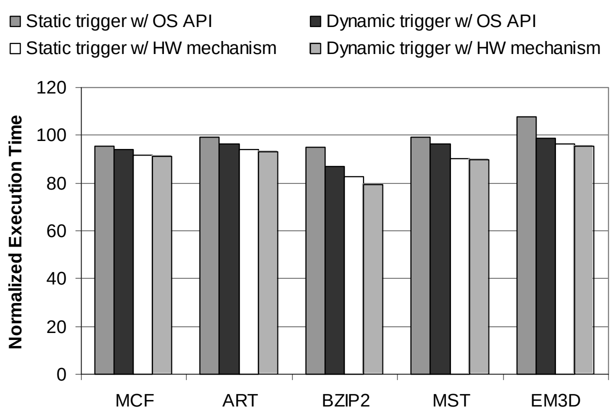


Figure 9. Performance comparison between static and dynamic trigger scenarios

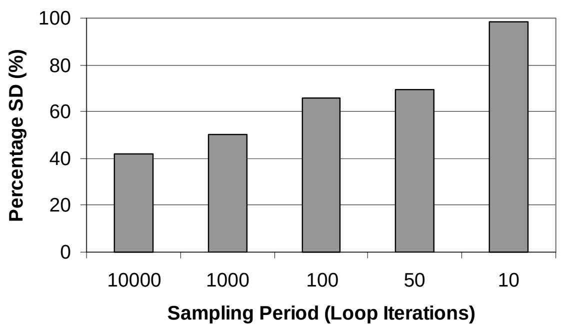


Figure 10. Percentage SD of L2 cache misses for various sampling periods

helper threads are recorded. For the remaining samples, the increased cycle counts due to detrimental effect of helper threads are discarded and the baseline cycle counts are recorded instead. The total number of the recorded cycles projects the performance of a perfect throttling scheme, where the throttling algorithm would activate a helper thread only for those samples with speedup.

Figure 9 depicts the total cycles for the targeted loops, not the entire program, for four different configurations; Static trigger with OS API, Dynamic trigger with OS API, Static trigger with HW mechanism, and Dynamic trigger with HW mechanism, where each bar is normalized on the cycles of the baseline. It is apparent that perfect throttling would provide non-trivial speedups beyond the static trigger performance. The figure shows 4.8% more gain with OS API and 1.2% gain with hardware mechanism. Interestingly, there exists correlation between the SD values in Table 4 and the impact of dynamic throttling in Figure 9. For instance, BZIP2 has the largest SD value for cycle counts showing the most dynamic behavior among the benchmarks, and the amount of execution time difference without and with dynamic throttling is also the largest. This implies applications with more dynamic behavior could benefit more from dynamic throttling. With perfect throttling algorithm and light-weight hardware synchronization mechanism, helper threads would provide as much as 20.6% wall-clock speedup for the targeted loop in BZIP2. This performance potential of dynamic throttling serves to motivate future efforts for optimization.

5.3. Sampling granularity and sensitivity of dynamic behaviors

Currently, the granularity of the sample-based trigger is limited by the cost of the thread synchronization mechanisms. Even at 1,500 cycles, the hardware synchronization cost remains to be more than twice the L2 cache miss latency. Consequently, the dynamic behavior of a program can be exploited at rather coarse-grain. In this section, we show that the time variance in the dynamic cache miss behavior becomes more pronounced as the resolution of sampling of the program execution increases, thus indicating much more room for exploring dynamic throttling at finer-granularity. In turn, this motivates further hardware optimization to reduce synchronization cost.

In Figure 10, the sampling period is varied from 10000, to 1000, 100, 50, and 10 loop iterations for the loop in `price_out_impl()` of MCF and the L2 cache misses are profiled for each sample using EmonLite. The graph shows the percentage SD for these different sampling periods. The SD values show a steady increase from 10000, to 1000, 100, and 50 iterations. Once the sampling period reaches 10 loop iterations, where each sample takes 2K cycles on average, the dynamic cache miss behavior fluctuates, and there is significant variation in L2 cache miss counts among different samples. Such dynamic cache behavior can only be captured at very high sampling resolution.

6. Key observations

6.1. Impediments to speedup

From our experience with helper threads, we have learned that to achieve significant speedup on a real machine, a number of issues that are correlated need to be addressed.

First, resource contentions in the hyper-threaded processor impose tricky tradeoffs regarding when to fire off a helper thread, how long to sustain the helper thread, and how frequent to reactivate the helper thread. In an SMT machine, for helper threads to be effective, potential resource contention with the main thread must be minimized so as not to degrade the performance of the main thread.

Second, program execution can exhibit dynamically varying behavior relative to cache misses and resource contentions. Therefore, there are some time phases where helper threads may not be helpful due to, for instance, lack of cache misses, MSHRs, cache ports, or bus bandwidth. We observe that indiscriminately running helper threads solely based on the compile-time placed static triggers is not always desirable. Dynamic throttling of helper thread invocation is important for achieving effective prefetching benefit without suffering potential slow down.

Third, to achieve more speedup with helper threads requires monitoring and exploiting dynamic behaviors of program execution. This requires hand-shaking between the main thread and the helper threads. To do this effectively, having very light-weight thread synchronization and switching mechanisms is crucial.

6.2. Essential mechanisms

To overcome these impediments, we first need better compiler algorithms to construct more judicious helper threads to ensure more timely activation and deactivation. In addition, the compiler can further optimize the helper thread to reduce the resource contention, e.g. by exploiting occasional stride-prefetch pattern, as a form of strength reduction, to accelerate helper thread execution, thus potentially minimizing resource occupancy time by the helper threads.

In addition, it is crucial to employ run-time mechanisms to capture the dynamic program behavior and throttle the helper threads, thereby filtering out helper thread activations, which lead to wasteful resource contention and unnecessary prefetches. Since the dynamic throttling mechanisms require very fine-grain thread synchronization, light-weight thread synchronization support in hardware is essential.

If provided with such compile-time and run-time support, helper threading can be a highly effective technique for dealing with the ever increasing memory latency problem in workloads that have large working sets and that suffer from significant cache misses, especially those misses that defy stride based prefetchers.

7. Related works

In pre-execution techniques, constructing effective helper threads is the key to achieve performance improvement. It can be done in either software or hardware. *Software-controlled pre-execution* extracts code for pre-execution from source code [10, 12] or compiled binaries [7, 11, 16, 25] using off-line analysis techniques. This approach reduces hardware complexity since the hardware is not involved in thread construction. In addition, off-line analysis can examine large regions of code, and can exploit information about program structure to aid in constructing effective pre-execution threads. In contrast, *hardware-controlled pre-execution* [1, 6] extracts code for pre-execution from dynamic instruction traces using trace-processing hardware. This approach is transparent, requiring no programmer or compiler intervention, and can examine run-time information such as delinquency of load instructions in an on-line fashion.

8. Conclusions

In this paper, we show that helper threads can indeed provide wall-clock speedup on real silicon. To achieve even more gain, however, there are acute challenges that can greatly affect the effectiveness of helper threads, such as hardware resource contention in the hyper-threaded processors, dynamic program behavior, and thread synchronization cost. In order to benefit from helper threads in a real system, certain run-time mechanisms are required to dynamically throttle the activation of the helper threads with very light-weight thread synchronization support. It is also beneficial for a compiler to generate efficient helper threads and judiciously place the static triggers.

We believe there is much headroom for helper threads in the future. The processors would spend more time on memory stalls due to the ever-increasing memory latency. Moreover, the compiler will generate highly optimized helper threads for the hyper-threaded processors and target more cache misses in the applications. For future work, our focus is on tackling the two important issues identified in this study; development of practical dynamic throttling framework and even lighter-weight user-level thread synchronization mechanisms.

9. Acknowledgments

The authors thank Gerolf Hoflehner and Dan Lavery for helpful comments on the compiler issues, and Shihjong Kuo for providing tools to implement EmonLite. We also thank Jamison Collins and the anonymous reviewers for their constructive comments on previous drafts of this paper. Donald Yeung was supported in part by NSF Computer Systems Architecture grant CCR-0093110 and in part by NSF CAREER Award CCR-0000988.

References

[1] M. Annavaram, J. M. Patel, and E. S. Davidson. Data Prefetching by Dependence Graph Precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 52–61, Goteborg, Sweden, June 2001. ACM.

[2] M. C. Carlisle. Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines. Technical Report PhD Thesis, Princeton University Department of Computer Science, June 1996.

[3] R. S. Chappell, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 186–195, Atlanta, GA, May 1999. ACM.

[4] T.-F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[5] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 1–12, Atlanta, GA, May 1999. ACM.

[6] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 306–317, Austin, TX, December 2001. ACM.

[7] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation:

Long-range Prefetching of Delinquent Loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, Goteborg, Sweden, June 2001. ACM.

[8] J. L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal, Issue on Pentium 4 Processor*, February 2001.

[10] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 159–170, San Jose, CA, October 2002. ACM.

[11] S. S. W. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, pages 117–128, Berlin, Germany, June 2002. ACM.

[12] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, Goteborg, Sweden, June 2001. ACM.

[13] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal, Volume 6, Issue on Hyper-Threading Technology*, February 2002.

[14] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-Processors: An Implementation of Operation-Based Prediction. In *Proceedings of the 15th International Conference on Supercomputing*, pages 321–334, Sorrento, Italy, June 2001. ACM.

[15] T. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *ACM Transactions on Computer Systems*, 16(1):55–92, February 1998.

[16] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the 7th International Conference on High Performance Computer Architecture*, pages 191–202, Monterrey, Mexico, January 2001. IEEE.

[17] M. Smith. Tracing with Pixie. Technical Report CSL-TR-91-497, Stanford University, Nov 1991.

[18] Y. Solihin, J. Lee, and J. Torrellas. Using a User-Level Memory Thread for Correlation Prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–182, Anchorage, AK, May 2002. ACM.

[19] Y. Song and M. Dubois. Assisted Execution. Technical Report CENG 98-25, Department of EE-Systems, University of Southern California, Oct 1998.

[20] 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*, pages 191–202, Cambridge, MA, May 2000. ACM.

[21] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995. ACM.

[22] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 54–58, Orlando, FL, January 1999. IEEE.

[23] Intel Corporation. VTune Performance Analyzer. http://developer.intel.com/software/products/VTune/index.html.

[24] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4), July 1984.

[25] C. B. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, Goteborg, Sweden, June 2001. ACM.

[26] C. B. Zilles and G. S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 172–181, Vancouver, Canada, June 2000. ACM.