Dynamic Helper Threaded Prefetching on the Sun UltraSPARC® CMP Processor

Jiwei Lu, Abhinav Das, Wei-Chung Hsu
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
{jiwei,adas,hsu}@cs.umn.edu

Khoa Nguyen, Santosh G. Abraham
Scalable Systems Group
Sun Microsystems Inc.
{khoa.nguyen,santosh.abraham}@sun.com

Abstract

Data prefetching via helper threading has been extensively investigated on Simultaneous Multi-Threading (SMT) or Virtual Multi-Threading (VMT) architectures. Although reportedly large cache latency can be hidden by helper threads at runtime, most techniques rely on hardware support to reduce context switch overhead between the main thread and helper thread as well as rely on static profile feedback to construct the help thread code. This paper develops a new solution by exploiting helper threaded pre-fetching through dynamic optimization on the latest UltraSPARC Chip-Multiprocessing (CMP) processor. Our experiments show that by utilizing the otherwise idle processor core, a single user-level helper thread is sufficient to improve the runtime performance of the main thread without triggering multiple thread slices. Moreover, since the multiple cores are physically decoupled in the CMP, contention introduced by helper threading is minimal. This paper also discusses several key technical challenges of building a light-weight dynamic optimization/software scouting system on the UltraSPARC/Solaris platform.

1. Introduction

Modern processors spend a significant fraction of overall execution time waiting for the memory systems to deliver cache lines. This observation has motivated copious research on hardware and software data prefetching schemes. Execution-based prefetching is a promising approach that aims to provide high prefetching coverage and accuracy. These schemes exploit the abundant execution resources that are severely underutilized following an L2 or L3 cache miss on contemporary processors supporting Simultaneous Multi-Threading (SMT) [8] or Virtual Multi-Threading (VMT) [24]. In hardware pre-execution or scouting [3], [15], [18], [22], [23], [25], [26], [28], the processor checkpoints the architectural state and continues speculative execution that prefetches subsequent misses in the shadow of the initial triggering missing load. When the initial load arrives, the processor resumes execution from the checkpointed state. In software pre-execution (also referred to as helper threads or software scouting) [2], [4], [7], [10], [14], [24], [29], [35], a distilled version of the forward slice starting from the missing load is executed, minimizing the utilization of execution resources. Helper threads utilizing run-time compilation techniques may also be effectively deployed on processors that do not have the necessary hardware support for hardware scouting (such as checkpointing and resuming regular execution).

Initial research on software helper threads developed the underlying run-time compiler algorithms or evaluated them using simulation. With the advent of processors supporting SMT and VMT, helper threading has been shown to be effective on the Intel Pentium-4 SMT processor and the Itanium-2 processor [7], [13], [24], [25], [29]. In an SMT processor such as Pentium-4, many of the processor core resources such as L1 caches, issue queues are either partitioned or shared. Helper threads need to be constructed, deployed and monitored carefully so that the negative resource contention effects do not outweigh the gains due to prefetching. The VMT method used a novel combination of performance monitoring and debugging features of the Itanium-2 to toggle between the main thread and helper thread execution. However, on Itanium-2, the large overhead in toggling between these modes limits the number of cycles available for actual helper code execution to a couple of hundred cycles. Only a few missing loads can be launched in this short time interval.

Almost all general-purpose processor chips are moving to Chip Multi-Processors (CMP) [19], [20], including the *Gemini*, *Niagara*, *Panther* chips from Sun, the *Power4* and *Power5* chips from IBM [16] and recent announcements from AMD/Intel. The IBM and

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

Sun CMPs have a single L2 cache shared by all the cores and private L1 caches for each of the cores. Since the cores do not share any execution or L1 resources, helper thread execution on one core has minimal negative impact on main thread execution on another core. However, since the L2 cache is shared, the helper thread may prefetch data into the L2 cache on behalf of the main thread. Since such CMPs are soon going to be almost universal in the general-purpose arena and since many single-thread applications are dominated by off-chip stall cycles, helper thread prefetching on CMPs is an attractive proposition that needs further investigation.

The minimal sharing of resources between cores gives rise to unique issues that are not present in an SMT or VMT implementation. First, how does the main thread initiate helper thread execution for a particular L2 cache miss? In an SMT system, both threads are co-located on the same core enabling fast synchronization. Second, how does the main thread communicate register values to the helper thread? In the Itanium-2 VMT system, the register file is effectively shared between the main and helper threads.

We have devised innovative mechanisms to address these issues, implemented a complete dynamic optimization system for helper thread based prefetching, and measured actual speedups on an existing Sun UltraSPARC IV+ CMP chip [27]. In our system, the main thread is bound to one core and the runtime performance monitoring code, the runtime optimizer and the dynamically generated helper code execute on the other core. Runtime performance monitoring selects program regions that have delinquent loads. The helper code generated for these regions is optimized to prefetch for delinquent loads. The main thread uses a mailbox in shared memory to communicate and initiate helper thread execution. The normal caching mechanism maintains this mailbox in the L2 cache and also in the L1 cache of the helper thread's core. We address many other implementation issues in this first evaluation of helper thread prefetching on a physical Chip-Multiprocessor and measure significant performance gains on several SPEC benchmarks and a real-world application.

The remainder of this paper is organized as follows. Section 2 provides the background and related work. Section 3 discusses the helper thread model in our optimization framework, including code selection, helper thread dispatching, communication and synchronization. Section 4 introduces the dynamic optimization framework on the UltraSPARC system. Section 5 evaluates the performance of dynamic helper threading and Section 6 draws the conclusion.

2. Background and Related Works

Many researchers have proposed to use one or more speculative threads to warm up the shared resources, such as the caches and the branch prediction tables, to reduce the penalty of cache misses and branch mis-predictions. These helper threads (also called scout threads) usually execute a code segment pre-constructed at compile time by identifying the instructions on the execution path leading to the performance bottleneck [5].

2.1. SMT/VMT vs. CMP

Pre-computation based helper threads have been evaluated on Pentium-4 with hyper-threading and on Itanium-2 system with special hardware support for VMT [7], [13], [15], [24], [29]. Other helper threading works such as Data-Driven Multi-Threading (DDMT) [3], Simultaneous Subordinate Micro-threading (SSMT) [28] and Transparent Threads [10] target at achieving effective helper threaded prefetching on SMT processors.

Unlike SMT and VMT, which share many critical resources, Chip Multi-processing (CMP) processors limit sharing, for example, to only the L2/L3 cache. While the restricted resource sharing moderates the benefit of helper threading to only L2/L3 cache prefetching, it also avoids the drawback of hard-to-control resource contention encountered by helper threading on SMT. The impact of different resource sharing levels to thread communication cost on CMP as well as the corresponding performance margin for pre-execution have been quantitatively assessed by Brown and et al. on a research Itanium CMP [14].

2.2. Dynamic vs. Static Helper Threading

Software-based dynamic optimization [6], [17], [30], [31], [33] adapts to the runtime behavior of a program due to the change of input data, and/or the underlying micro-architecture. Current dynamic optimizations include data prefetching, procedure inlining, partial dead code elimination, and code layout, which have been proven to be useful complements to static optimizations. Optimizations such as data cache prefetching and branch mis-prediction reduction are usually difficult to perform at compile time since cache miss and branch mis-prediction information may not be available. Furthermore, program hot spots may as well change under different input data sets or

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

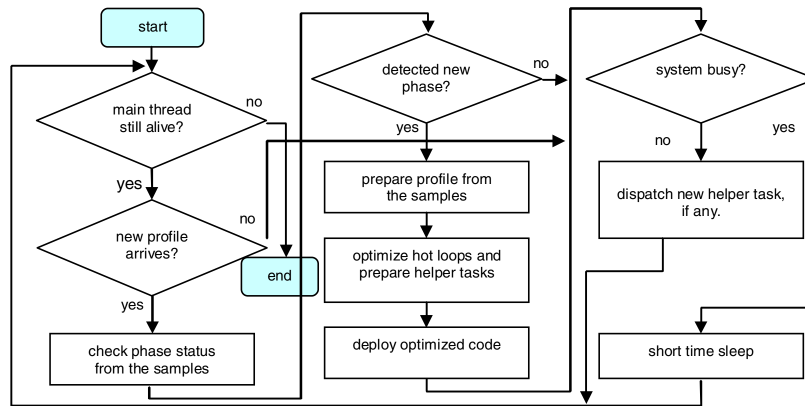

Figure 1. Control flow of runtime optimization for helper threading on UltraSPARC CMP.

on processors with different micro-architectures. For example, statically pre-constructed helper threads may incur performance degradation if the generated binary is executed under different inputs or on a non-CMP processor. For this reason, it is more desirable to generate helper threads dynamically.

3. Helper Thread Model for CMP

This paper focuses on the aspects of helper-threaded prefetching through dynamic optimization on a real CMP processor, UltraSPARC IV+ (code named "Panther"). Since each Panther processor encapsulates two identical cores, helper threads can run on the often under-utilized secondary core to prefetch data into the shared cache so as to improve the main thread performance. Unlike other helper threading schemes discussed for SMT [4], [7], [10], [13], [15], [29], thread communication for Panther has to be conducted through the L2 cache since the L2 cache is the closet level of shared caching. On the other hand, construction of helper threads on Panther is less constrained by resource contention as each core is a complete processing unit with its own functional units, TLB, L1 caches and register files; execution of the helper thread would have less negative impact on the main thread performance.

3.1. Piggyback on the Optimizer Thread

In Kim's helper threading work [7], the main thread activates the helper thread in a master-slave fashion. Where there is no work for the helper thread, the SMT processor runs in Single-Threading (ST) mode to avoid performance degradation in the main thread. The overhead of switching from ST to MT, or vice versa, becomes a major concern if no hardware support is provided. To alleviate such cost, they prototyped several lightweight instructions for thread synchronization. Other research work on helper threading also faces the same issue.

Our dynamic optimization system is based on the ADORE dynamic optimization framework [17], which spawns a secondary thread to collect runtime performance profile, detect phase changes, select hot regions and deploy optimizations to the original binary. This secondary thread is in sleep mode most of the time and it consumes very little system resources. To employ software scouting in our runtime optimization system, we piggy-back the helper thread on the dynamic optimization thread, meaning that the dynamic optimization thread communicates with the main thread to prefetch data in the post-optimizing stage. This design minimizes the cost of dynamically triggering helper threads and maintains a better control of helper threading at runtime.

3.2. Task Dispatching

Figure 1 shows the control flow of the merged dynamic optimization and helper thread. The control loop first waits for a new *profile-window* [17], upon which the phase detector is called to check for major changes in the behavior of the main program's execution. Once a new phase becomes stable (Section 4.3), the dynamic optimizer selects the hot regions as candidates for software scouting. The optimizer generates two code segments in the code cache, one for the main thread and the other for the helper thread. The code segment for the main thread communicates register values and synchronizes with the helper thread. It is patched to the binary of the main thread so that this code segment is executed prior to entering each hot region. The code segment for the helper thread prefetches data that may likely be needed in the hot region by the main thread. Once a helper thread code is generated, the merged

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

optimization/helper thread enters a "scouting" mode. It checks a dispatching table (the mailbox) to see whether there are outstanding requests for scouting. Such requests are dynamically registered by the newly generated regions/loops. These requests carry the location of the code segment in the code cache that the helper thread should execute.

In Dorai et al.'s work [10], each scouting task is a function. The optimized loop executed in the main thread passes the corresponding function pointer to the helper thread. We use a different approach by passing only the function ID to the helper thread while the function pointer is maintained in the dispatching table at entry <ID>. This gives more freedom to the runtime optimizer -- whenever the dynamic optimizer learns that the helper task for a certain region/loop is not helping (e.g. cache stalls are not reduced), it can simply change the addresses in the corresponding entry to point to a newly generated helper task.

Figure 2 illustrates how the dispatching table works. Each of the three optimized hot loops passes its ID to the helper thread when executed. The helper thread will call *Helper_Func_<ID>* to perform its helper task. The first task on the list is an idle function that will be called when there is no task to perform. Finally, in a throughput computing environment, if all processor cores are busy running useful jobs, the scouting job may be skipped so that all the cores are available for increasing throughput.

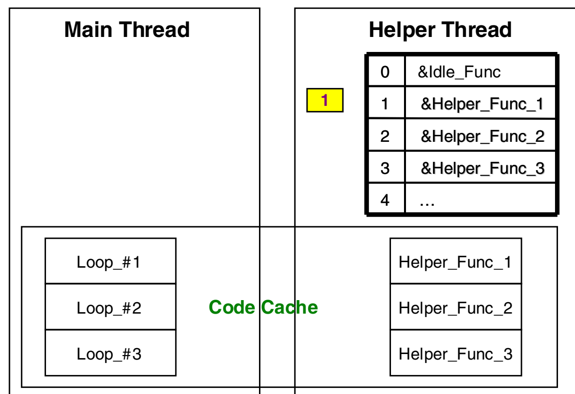

Figure 2. Dispatching Table.

3.3. Loop-based Region Selection

Most helper thread schemes for data cache prefetching focus on selecting hot loops as candidates because 1). For data cache prefetching, delinquent loads are likely found in frequently executed loops, and 2). Loops can amortize the cost of communication and minimize the cost for synchronization. Consequently, our runtime optimization system uses runtime profile to select hot loops and determine whether helper threading may be beneficial based on the information sampled from hardware performance counters. However, there are cases where loops may have been transformed into various shapes by compiler optimizations such as loop peeling and software pipelining; hence the loop structures observed at runtime can be obscure. To deal with this problem, our runtime loop selector searches in a broader context for loop structures to reduce the risk of missing important candidates[1].

3.3.1. Loop Processing

Once a stable phase is detected, the code selector scans a hot region for backward branches to delimit the rough boundary of a hot loop. The Sun UltraSPARC architecture provides a branch prediction bit for branch instructions. The compiler sets such prediction bits based on its best knowledge of the program structure or from profile feedback. Accordingly, our loop selector skips backward branches with a "not taken" hint to prevent the selected code from exiting the scouting loop prematurely that reduces the effectiveness of scouting. Each PC sample is a PC location associated with a sampled performance event. A loop is considered hot when there are a large number of PC samples occurring in it. To lower the cost incurred on optimizing less important regions, we select loops in sorted order, starting from the hottest PC, one by one until the accumulated PC samples of all selected loops exceed 90% of all samples.

3.3.2. Region Selection

Next, the code selector performs a quick profile-assisted control flow analysis on each loop to check whether the critical path in the loop covers the majority of the loop's PC samples. This is to reassure the quality of the selected loops. Additionally, some loops can overlap with each other as they are essentially portions of a large loop. Such loops will be merged. Selecting nested loop is another challenge. For most inner loops, fewer extra registers are required to perform inter-thread communication, whereas the cost of doing so might be unbearable if the inner loop iterates only a small number of times. Contrarily, such overhead can be well amortized in outer loops, yet getting extra registers for outer loops is often more costly. To make appropriate choice, our code selector estimates the relative execution time between the inner and the outer loop by computing the total number of PC samples and the *minimal execution cycles* of each loop. The *minimum execution cycles* is

[1] Some architectures support branch history information that simplifies this work.

0-7695-2440-0/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

the un-stalled cycles of a single iteration plus cycle stalls from delinquent loads. We divide the PC sample count by minimum execution cycles to roughly figure out the relative trip count of each loop (Before that, the values of both metrics of the inner loop are subtracted from the outer loop). If the inner loop's trip count is more than Δ_T times (currently Δ_T is 10) of that of the outer loop, the code selector would favor the inner loop, otherwise, it selects the outer loop.

In the final step, loops selected from the above operations are decoded and assigned initial values, such as structural and performance properties.

3.3.3. Delinquent Load Identification

Delinquent loads refer to load operations causing heavy cache miss stalls. The *libcpc* library in Solaris (similar to *pfmon* [12] library in Linux/Itanium) enables the signal handler to collect cache miss events when the hardware performance counter overflows at the user-defined sampling rate. However, due to the delay in the pipeline, the event address acquired from the interrupts is often a few instructions past the actual event, which is called "interrupt skid". Fortunately, this lag rarely spans across taken branches. Thus a simple back-scan is sufficient to find the real delinquent load, except when the interrupt hits on a branch, then the instruction in the branch delay slot is also examined.

3.4. Helper-Thread Code Generation

The scouting code in each helper task is constructed by selecting the *backward slices* [5] of the delinquent loads plus instructions that change the control flow. The main thread needs to pass *live-in* variables to the helper thread and maintain proper synchronizations with the helper thread.

3.4.1. Stores, Prefetches and Non-Faulting Loads

In the scouting code, memory stores are ignored as the helper thread is speculative in nature and should not modify the architecture state of the main thread. However, some computations may involve local variables on the main thread's stack. Such memory dependences can be easily detected from their "[%SP + off]" or "[%FP + off]" addressing patterns. These variables can be treated as registers and their corresponding references (load/store) can be selected as well. However, such references need to be converted into register references or private memory references since the stack pointer of the helper thread is different.

On Panther, a cache missing load instruction blocks the pipeline whereas a prefetch instruction does not. Accordingly, delinquent loads in scout code should always be replaced by prefetches unless their result is used by subsequent computations. In the latter case, a bit in the IR (Intermediate Representation) of that load is set indicating its result is used by other selected instructions, so this delinquent load will be converted into a non-faulting load. It is quite often that load operations in the helper thread can be replaced by non-blocking prefetches. This conversion usually enables the helper thread to run ahead of the main thread even if the distilled slice is not much smaller than the original code in the main thread.

Machine Configuration	Cost in cycles
Dual-core UltraSPARC IV+ with shared L2 Cache	60 ~ 65
2-way single-core UltraSPARC III	450 ~ 500

Table 1. The cost of inter-threading communication via memory operation.

3.4.2. Inter-Thread Communication

Inter-thread communication refers to the operations of activating the helper thread and passing live-in variables. The latency of communication is important to the effectiveness of the prefetching because it takes time for the helper thread to run ahead of the main thread. The later the helper thread starts, the longer it takes to catch up with the main thread. It is difficult to schedule communication code far ahead of the loop entry due to the complexity of control and data dependences. Therefore, prior work propose hardware support for light-weight thread switching to trigger the helper thread and use shadow registers for passing live-in variables. Unfortunately, such support is not available on the Panther processor. In our system, we attach a shared memory buffer to each entry in the dispatching table to perform inter-thread communication. The live-in variables to be passed from the main thread to the helper thread include loop invariants and variants. Invariants only need to be passed once rather than at every synchronization point (see Section 3.4.3).

The Solaris operating system randomly schedules user threads to any core/chip on a regular basis by default. To avoid losing the benefit from having the helper thread warm up the shared cache, the helper thread and main thread must be bound to different cores on the same chip. This is enforced by using the system call *processor_bind* at the beginning of each thread's execution.

The cost of inter-thread communication through memory can be minimized by the shared cache. On

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

the Panther CMP, the one way communication latency is ~60 cycles. If the cache is not shared, however, it can be as high as 500 cycles (shown in Table 1). This low cost enables a large window for effective helper threaded prefetching as the helper thread can catch up and run ahead of the main thread easily.

3.4.3. Synchronization

Since the merged optimization/helper thread takes care of both dynamic optimization and helper thread prefetching, a well-managed scheduling scheme is needed. Once a helper task starts, it must finish within a certain time limit so that critical dynamic optimization tasks such as phase change detection can take place in a timely manner. Additionally, the dynamic optimizer must ensure that no incorrect control flow leads to run-away loops. Such run-away loops not only thwarts the merged optimizer/helper thread from gaining control to perform its continuous profiling and optimization jobs but also pollutes the data cache with useless data. To address this issue, the helper thread loop synchronizes with the main thread loop after a certain number of iterations [7]. In our system, we use an asynchronous protocol between the two threads to avoid cross-checking each other's progress. In this protocol the helper thread maintains a single counter while every main thread loop maintains a private counter. All counters consist of three components: a task ID, a synchronization index and an iteration counter. The "iteration counter" is used in each loop to track the number of loop iterations. When it overflows a preset threshold, say 128, the main thread and helper thread take different actions. The helper thread will terminate the current scouting work while the main thread passes updated values of live-in variables and increments the synchronization index to indicate the start of a new synchronization interval, before continuing normal execution. When the helper thread enters the helper function again, it compares the first two parts of its counter with the main thread loop's counter. Equality of the comparison indicates this scouting task has been done, hence is skipped. Inequality means this is either a new task requested from a different loop or the old loop has entered a new synchronization interval so the helper thread should synchronize the counter and start the scouting work. This protocol minimizes the synchronization overhead in the main thread as it never waits for the helper thread.

3.4.4. Qualification Test

Helper threaded prefetching is meant to be applied to all loops experiencing external cache misses. However, some loops are particularly difficult for effective prefetching since the results of delinquent loads are needed on the loop's critical path. The helper thread hence may not run faster than the main thread, making the prefetching useless. As a result, we apply a qualification test to rule out such loops at the end of code generation lest these loops degrade the performance. In this test, a set of metrics are used to assess the risk of applying helper threading, such as the loop size, the PC sample coverage and the lengths of misses' dependence chains, etc.

4. Dynamic Optimization Framework

In this section, we briefly describe the effort of implementing a dynamic optimization system on the UltraSPARC architecture [2] and discuss architectural features that are important to runtime optimization; in particular, the new helper threaded prefetching.

4.1. Startup of Runtime Optimizer

Although the way to start-up a runtime optimization system varies across different implementations [6], [17], [30], [31], [33], it is commonly accepted that the runtime optimizer should be activated at the beginning of programs' execution. A straightforward solution is to make the optimizer a dynamically linked library loaded by the runtime loader. On Sun's Solaris operating system, an environment variable called *LD_PRELOAD* can be explicitly set to run library code right before program execution. We use this interface to start our dynamic optimizer as it is easy to control.

4.2. Hardware Performance Monitoring

Modern micro-processors provide hardware performance monitoring (HPM) capabilities and powerful counters that facilitate performance monitoring and profiling. For example, ADORE [17] takes advantage of the comprehensive Itanium PMU to achieve low-overhead runtime optimization and in turn sped up many programs, including large applications.

The *libcpc* library in Solaris is used to access the performance counters. Since the current UltraSPARC only implements two physical counters, monitoring multiple performance events must go through interleaving counters. The drawback of interleaving, however, is that the profiler receives fewer samples for each event if the sampling rate remains unchanged. To maintain the quality in dynamic profiling, we choose to dedicate one physical counter to the most

[2] Many of the issues are also applicable to other architectures.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

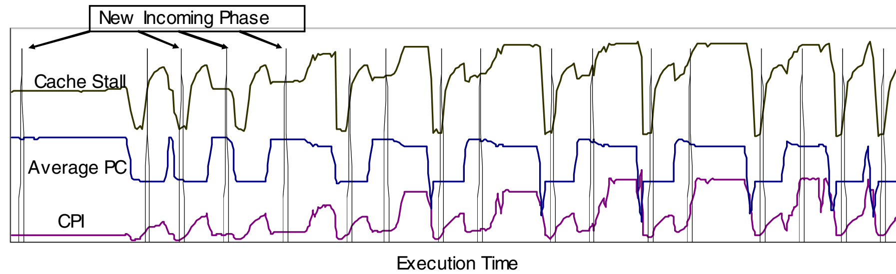

Figure 3. Three performance characteristics (L2-Stall, Average PC, and CPI) exhibit the runtime behavioral change of SPECINT benchmark *mcf*. The vertical lines represent new phases detected by the PC_Centroid approach.

important event (e.g. CYCLE_CNT), and use the second counter to interleave other important events. Additionally, a slight change to the Solaris kernel has been implemented to enable buffering the interrupts generated from event overflows. This buffering scheme significantly reduces the overhead of signal handling from 8-10% down to 1-2%. With the greatly reduced signal handling cost, the dynamic optimization system is able to afford a higher sampling rate for continuous performance monitoring.

4.3. Phase Detection

Running programs often exhibit time-varying behavior called *phases*, as the example shows in Figure 3. A phase-sensitive optimizer that adapts to phase changes is able to identify more optimization opportunities. Several phase detection schemes have been proposed [1], [32], yet most of them tend to capture fine-grain phases that may incur excessive overhead to software optimizations. Consequently, we decide to use ADORE's low-cost *PC-Centroid* [17] approach to detect phase changes.

First, the algorithm calculates the average PC address of all samples collected in a fixed time interval, called *PC-Centroid*, as a depiction of phase to reveal the pivot of code mass that the main program has executed. In actual computation, a few high bits in the PC address must be masked out to deal with the situations where frequent switches from the user code to the shared library code (shared library code often locate in high address range) may generate misleading *centroid*s.

Next, the expectation value E and the standard deviation D of n (n is a fixed value, e.g. 7) most recent PC-Centroids are calculated to form a tolerant space [E–D, E+D]. If the newly arrived PC-Centroid is shifting beyond the current tolerant space by some extent, a signal of *phase change* will be raised. To improve this scheme, a state machine has also been introduced to help capture important phases and ignore transient insignificant changes.

4.4. Optimization

4.4.1 Code Cache and Reachability Issue

The code cache stores the optimized code of each running program, where the execution of the program can be re-directed to by patching the entry point to a code region with a branch. To be accessible within the same process, the code cache must be allocated in the same virtual address space. Direct branching on some platforms has limited branch range. For example, the largest range of branch distance on UltraSPARC is ±8MB only. Therefore, the reachability issue arises if the code cache sits beyond the range of a single branch instruction.

```
00010000     320K    r x    /home/a123948/a.out
0006E000     24K     rwx    /home/a123948/a.out
00074000     616K    rwx    [ heap ]
FF100000     688K    r x    /usr/lib/libc.so.1
FF1BC000     32K     rwx    /usr/lib/libc.so.1
FF340000     16K     r x    /usr/lib/libmp.so.2
FF354000     8K      rwx    /usr/lib/libmp.so.2
FF370000     8K      rwx    [ anon ]
FF3B0000     160K    r x    /usr/lib/ld.so.1
FF3E6000     16K     rwx    /usr/lib/ld.so.1
FFBD2000     120K    rw     [ stack ]
```

Figure 4. An application's runtime virtual address mapping on Solaris 10.

Figure 4 shows an application's virtual address mapping on the UltraSPARC/Solaris system. Close investigation reveals many "holes" in this address space where the code cache can be safely allocated to maintain reachability. For instance, the space from address "00060000" to "0006E000" is a good candidate. There are other holes in the higher address space but all beyond the reachability of one direct branch from the text segment. If the code cache has to be positioned in those addresses, trampoline code that uses indirect branch is needed to assure reachability, although the trampoline code requires some unused

0-7695-2440-0/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

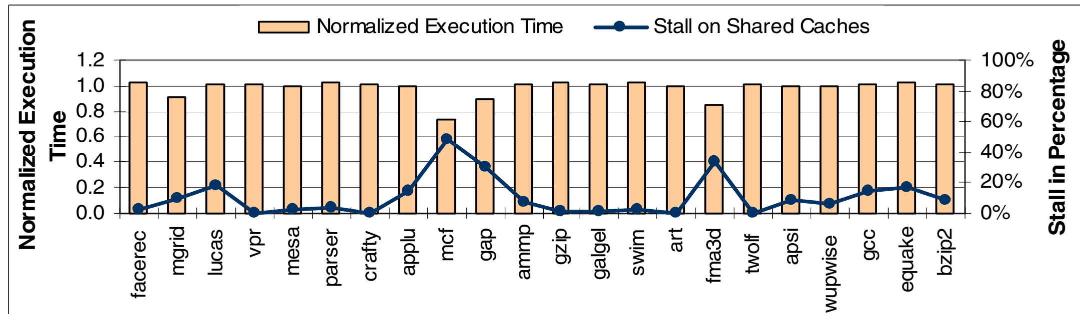

Figure 5. Helper threaded prefetching on CPU2000 benchmarks compiled with base option and profile feedback. 6 programs have L2/L3 cache miss stall no less than 20%.

bytes in the text segment that are reachable by direct branches from the original code. Such code sequence is costly and should be avoided as much as possible.

4.4.2 Weak/Strong Prefetch

The Panther processor supports two variations of prefetch instructions: *weak* prefetch and *strong* prefetch. Although their semantics are implementation dependent, the key difference is that the *weak* prefetch can be dropped when the prefetch queue is full, or when a TLB miss is encountered. In general, *strong* prefetch should be used if many prefetches are needed to bring in useful data, and *weak* prefetch should be used when the prefetch is more speculative.

Initially, we replaced delinquent loads with *weak* prefetches in the helper thread, especially when control dependent instructions are not included (as suggested in some earlier research). However, as the two cores do not share TLB, a *weak* prefetch can get dropped in case of a TLB miss. Unlike in the main thread, where the regular loads will eventually bring in the missed TLB entries, the helper task may end up having all prefetches dropped due to TLB misses (since the regular loads are replaced by prefetches). We therefore decide to use *strong* prefetch in the helper thread code. Although this is generally a better code generation strategy, there are cases when *weak* prefetch may yield better performance (Section 5.6).

4.4.3 Register Allocation

Runtime binary optimization often requires extra registers to perform computations. For helper threading on CMP processors, the main thread only needs to free up two to three registers for communication and synchronization; there is no other computation needing additional registers. Extra registers can be obtained by compiler through register pre-reservation or annotations indicating which registers are not used in a procedure or a loop nest. In ADORE [17], free registers are obtained by using the new IA64 instruction *alloc* to dynamically allocate registers from the register stack. On UltraSPARC, we decide to collect the register liveness information and spill unused or less-frequently used registers to memory. For instance, according to the SPARC ABI, some spaces are unused in the function's activation record where a small number of registers can be safely spilled. Spilling and restoring registers inevitably adds overhead. Yet such overhead is normally negligible since the optimization for helper threading is targeted at hot loops. Finally, register allocation is not an issue for the helper functions as registers in the helper thread context are always available.

4.5. Atomic Patching and Cache Coherence

In a binary optimization system, *patching* refers to redirecting the original code execution to the optimized code in the code cache. This operation must be performed atomically in a multi-threading environment to ensure the execution will see either old code or new code, but never half-way patched instructions. In our runtime optimization system, *patching* involves using a 32-bit store to place a single branch instruction in the original code, which is atomic. In addition, the Panther CMP's I-Cache snoops on stores issued from all processor cores so that the patched instruction will be brought into the I-Cache, making it unnecessary to explicitly execute an *iflush* instruction. Other solutions about the cache coherence can be found in Brown et al.'s work [14].

5. Performance Evaluation

5.1. Machine Configuration

We implement our dynamic optimization system on a 4-way Sun UltraSPARC IV+ (code named *Panther*) machine [27] running Solaris 10. Each processor is a dual-core CMP sharing a 2MB on-chip L2-cache. The

0-7695-2440-0/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

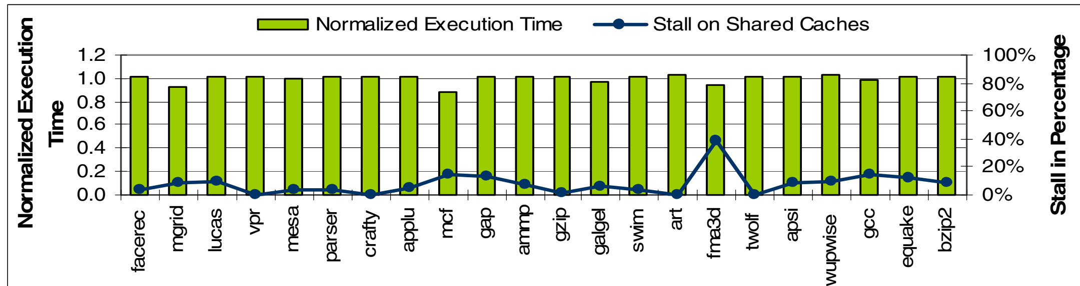

Figure 6. Helper threaded prefetching on CPU2000 benchmarks compiled with peak option.

per-core L1-I/D Caches are 64 KB, 4-way set associative with 64B line size. Many other architectural enhancements have been added onto this new chip including a 32MB shared external cache, enhanced Branch Prediction Unit and support for *weak/strong* prefetches.

5.2. Benchmarks

To examine the effectiveness of helper threaded prefetching on programs encountering cache misses, we studied the CPU2000 benchmark suite (we believe CPU2006 would be more appropriate to study cache performance, but they are not available yet), as well as one real world application, *Fluent* (version 6.1.9), which is a large computational fluid dynamics software. All CPU2000 programs are compiled by Sun Forte Compiler using two options: 1) base + profile-feedback, 2) peak. They all run reference input (the first input set if there are multiple sets). For *Fluent*, we compiled it using the best option we could find and fed it with 9 different data input sets: 3 small (*fl5s*1-3), 3 medium (*fl5m*1-3), and 3 large (*fl5l*1-3), obtained from the *Fluent* website [9].

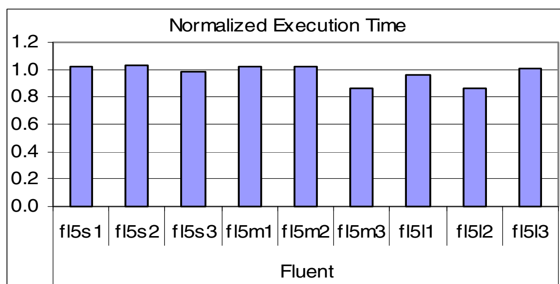

Figure 7. Helper threaded prefetching on *fluent*.

5.3. Performance Results

As shown in Figure 5 and Figure 6, stall on L2 and L3 cache still dominates the performance of a few CPU2000 binaries[3]. However, with the 32MB L3 cache, most CPU2000 programs suffer from only L2 cache misses. Our runtime helper threaded prefetching was able to speed up four base binaries by as much as 35% (*mcf*) and three peak binaries. Similarly, Figure 7 shows that performance gain from 2% to 16% can be achieved from 4 out of the 9 input sets for the application *fluent* (with input data *fl5s3*, *fl5m3*, *fl5l1*, and *fl5l2*).

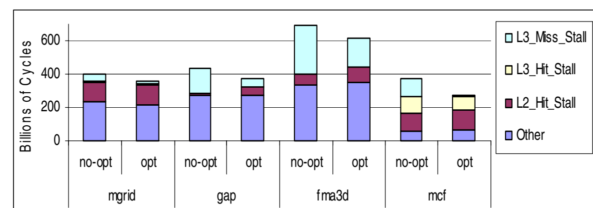

Figure 8. Cycle breakdown before and after prefetching. "no-opt" is for original code and "opt" is for dynamic helper threaded prefetching.

Figure 8 shows the change of cycle breakdown before/after applying help threaded prefetching. Helper threaded prefetching is particularly effective on hiding L3 cache miss latency, as shown in the figure. Since the helper threaded prefetching does not decrease L1 data cache miss penalty (shown as *L2_hit_stall* in Figure 8), we need to consider CMT type helper threading or in-thread prefetching if L1 cache miss stall dominate the performance. Note that the working set sizes of CPU2000 programs may be less suitable to evaluate the effectiveness of a cache hierarchy of latest processors. The yet to be announced CPU2006 programs would have a working set size more representative of current and future applications. For those programs that do not benefit from helper threaded prefetching, there is only 1-2% extra overhead caused by the runtime optimizer.

---

[3] On the CMP processor, L1 cache is per-core, whose misses cannot be pre-fetched from other cores.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

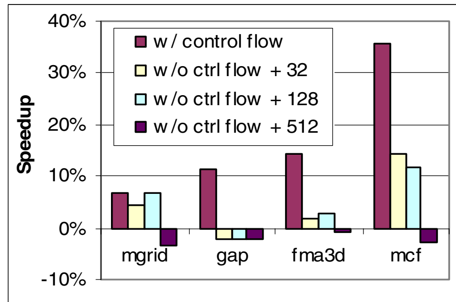

Figure 9. Effect of excluding control flows in helper tasks. *32*, *128* and *512* represent the synchronization interval in loop iterations.

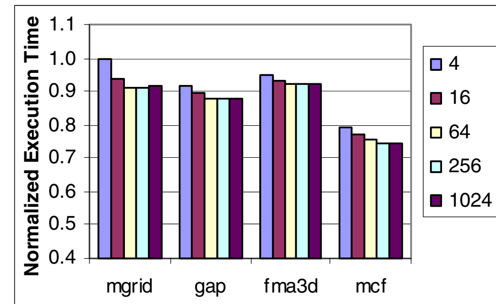

Figure 10. Effect of different synchronization intervals.

In Figure 5 and Figure 6, our dynamic helper thread optimizer did not speed up some programs with high cache miss stalls. As an example, our optimizer failed to obtain sufficient registers for the transformation for *wupwise* (to avoid excessive spilling). For other programs, the tight data dependence from linked-list chasing prevents us from performing effective scouting.

5.4. Effect of Control Flow Structures

To ensure the prefetching accuracy of the helper thread, the scout code for each loop includes instructions that change the control flow, as discussed in Section 3.4. However, it would be interesting to understand the impact of not including such instructions. Would the removal of these instructions make the helper thread run faster and hide miss latency more effectively for the main thread?

Figure 9 studies the effect measured on the four base binaries that we speedup using helper thread prefetching. Without the control flow instructions, only one program completely lost its performance gain (*gap*) while the other three still benefit from the software scouting, although the speedups now become less. This is because the helper thread missed some early exits of the loop by not computing the control flow, which causes a delay in reaching or entirely missing the next synchronization point. Additionally, without control flow computation the helper task runs the maximum number of iterations that is set as synchronization interval (e.g. 32, 128 and 512). If the actual iteration count in the main thread loop is much fewer, the helper thread might severely pollute the L2 and L3 cache (as shown in Figure 9, *w/o loop control + 512* always renders the worst performance). As a result, our runtime optimizer by default selects control flow computations into the helper thread code for prefetching.

5.5. Evaluation of Synchronization

Since helper threads do not include store instructions, and may skip control dependent instructions, there is a risk that the helper thread loop runs out of control. Therefore, synchronization intervals are set between the main thread and the helper thread to keep run-away loops under control.

To find appropriate values, Figure 10 evaluates five intervals, at which the helper thread synchronizes with the main thread every: 4, 16, 64, 256, or 1024 iterations. Since the instructions computing control flow are included and the external cache is large enough for the CPU2000 programs, there is not much performance degradation when large intervals like 1024 are used. However, it is quite interesting that helper threading is still very effective even at a small interval of 4. At such a small interval size, one might expect that the helper thread cannot run considerably ahead of the main thread and initiate prefetches sufficiently early to hide the miss latency. In fact, while the main thread is stalled on the first cache miss in the interval, the helper thread uses prefetch instructions in place of some regular loads to initiate multiple misses. Thus, the helper threading scheme overlaps multiple misses achieving high Memory-Level Parallelism (MLP) even when it is unable to run sufficiently ahead of the main thread to hide miss latency. Therefore, significant speed-up is achievable even with a smaller interval size.

5.6. Weak vs. Strong Prefetches

The relative benefits of using *strong/weak* prefetches on the UltraSPARC CMP processor are evaluated here. As mentioned in Section 4.4.2, *strong* prefetch is generally preferable in helper threading since the TLB is private per core and *weak* prefetches do not ensure that data is brought into the shared cache. Figure 11 demonstrates that when *weak*

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

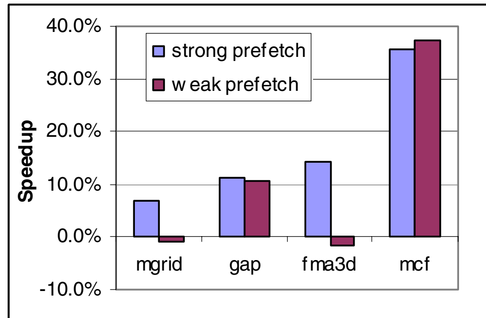

Figure 11. Strong vs. weak prefetch.

prefetches replace *strong* prefetches, performance gains of some programs (*mgrid* and *fma3d*) are diminished. Nevertheless, *weak* prefetch works better for some cases, such as *mcf*. There are two reasons for this. First, in *mcf*, the traversal of the linked-lists must use non-faulting loads, not prefetch instructions. As a result, TLB misses can be resolved by non-faulting loads as well, regardless the *weak* prefetches are dropped or not. Second, the *strong* prefetch could cause delay when the prefetch queue is full. For this reason, when TLB misses is less a concern and prefetch is more speculative, using *weak* prefetches may have an edge over strong prefetches in the helper thread to allow it to run faster.

6. Conclusion and Future Work

This paper presents the design and implementation of a dynamic optimization system capable of helper threaded prefetching on a state-of-art UltraSPARC CMP processor, where two on-chip processor cores share an on-chip L2 cache and an off-chip L3 cache. We have shown that effective helper threaded prefetches can be generated dynamically using runtime profiling based on hardware performance monitoring. By utilizing the otherwise idle processor core, the dynamic optimizer has a great potential to speed up single-threaded user applications, particularly those suffering significantly from L2 or external cache misses. For programs that helper threaded prefetching does not help, our system introduces negligible slowdown (< 2%) due to the light weight runtime profiling mechanism. This paper also discusses the critical issues of implementing efficient helper threaded prefetching, which include efficient synchonization/communication, whether control flow should be kept in the helper thread, and the impact of using different prefetch instruction in the helper thread. We believe this dynamic optimizer would have greater performance impact on the next generation benchmarks such as the CPU2006 with larger working set sizes.

In the near future, we will focus on finding an arbitrator to decide whether to select helper threaded or in-thread cache prefetching optimizations based on profitability analysis. Specifically, since some programs sped up by helper threading can also be sped up by in-thread prefetching, the dynamic optimizer may favor in-thread optimization when the other on-chip cores could be used to run other jobs. In-thread optimization should also be considered when the performance is dominated by private L1 cache misses. Although it seems that helper threaded prefetching on CMP may be inadequate in a throughput oriented computing environment, a number of transaction processing benchmarks show that some threads (such as log writer and DB writer) could be more time critical than others in attributing to the total performance. Hence helper threaded prefetching can be selectively applied to the time critical threads, even in a throughput computing environment.

The thread synchronization mechanism requires further evaluation on the future CMP processors as well. The reason for this is that a faster synchronization mechanism (e.g. through shared L1-Cache or hardware assist) will help utilizing the idle core more efficiently according to our current scheme, particularly when multiple main threads are involved. Other aspects that need enhancement include register allocation, region selection and undoing ineffective optimization to maximize the performance gain of dynamic help threaded prefetching.

References

[1] A. Dhodapkar and J. E. Smith. “Comparing Program Phase Detection Techniques”, In *Proc. of Micro-36*, Dec. 2003.
[2] A. Bhowmik and Manoj Franklin. “A General Compiler Framework for Speculative Multithreading,” in *Proc. of SPAA’02*, Aug 2002.
[3] A. Roth and G. Sohi, “Speculative Data-Driven Multi-Threading,” in *Proc. of HPCA-7*, Jan 2001.
[4] C.-K. Luk, “Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors,” in *Proc. of ISCA-28*, July 2001.
[5] C. Zilles and G. Sohi, “Understanding the Backward Slices of Performance Degrading Instructions,” in *Proc. of ISCA-27*, 2000.
[6] D. Bruening, T. Garnett, and S. Amarasinghe. “An Infrastructure for Adaptive Dynamic Optimization”, In *Proc. of CGO’03*, March 2003.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.

[7] D. Kim, S. S. Liao, P. H. Wang, J. d. Cuvillo, X. Tian, X. Zhou, H. Wang, D. Yeung, M. Girkar, J. P. Shen, "Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors," In *Proc. of CGO'04*, March 2004.

[8] D. M. Tullsen, S. J. Eggers, and H. M. Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism," In *Proc. of ISCA-22*, June 1995.

[9] Fluent Benchmarks. http://www.fluent.com/software/fluent/fl5bench/intro.htm.

[10] G. K. Dorai, D. Yeung. "Transparent Threads: Resource Sharing in SMT Processors for High Single Thread Performance," in *Proc. of PACT-2002*, Sept 2002.

[11] G. Sohi, S. Breach, and T.N. Vijaykummar, "Multiscalar Processors," in *Proc. of ISCA-22*, Jun 1995.

[12] Hewlett-Packard Corp. Perfmon Project. http://www.hpl.hp.com/research/linux/perfmon.

[13] H. Wang, P. Wang, R. D. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, and J. Shen. "Speculative Precomputation: Exploring Use of Multithreading Technology for Latency," *Intel Technology Journal*, Feb 2002.

[14] J. A. Brown, H. Wang, G. Chrysos, P. H. Wang, and J. P. Shen. "Speculative Precomputation on Chip Multiprocessors," in *Proc. of MTEAC-6*, Nov 2002.

[15] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, J. Shen, "Speculative Precompuation: Long-range Prefetching of Delinquent Loads," in *Proc. of ISCA-28*, July 2001.

[16] J. Kahle. "The IBM Power4 Processor," in *Microprocessor Report*, Oct 1999.

[17] J. Lu, H. Chen, P-C Yew, W-C. Hsu. "Design and Implementation of a Lightweight Dynamic Optimization System," in *The Journal of Instruction-Level Parallelism*, CFP-1, 2004.

[18] J. Dundas and T. Mudge. "Improving data cache performance by preexecuting instructions under a cache miss," in *Proc. of ICS-11*, 1997.

[19] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen and K. Olukotun. "The Stanford Hydra CMP," in *IEEE Micro*, Mar-Apr 2000.

[20] L. Spracklen, S. G. Abraham. "Chip Multithreading: Opportunities and Challenges," in *Proc. of HPCA-11*, Feb 2005.

[21] M. Tremblay. "The MAJC Architecture: A Synthesis of Parallelism and Scalability," in *IEEE Micro.*, Vol. 20, No. 6, pp. 12-25, 2000

[22] M. Dubois. "Fighting the Memory Wall with Assisted Execution," in *2004 ACM Computing Frontiers Conference*, Ischia Italy.

[23] 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. of HPCA-9*, 2003.

[24] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych and J. P. Shen. "Helper Threads via Virtual Multithreading On an Experimental Itanium 2 Machine", in *Proc. of ASPLOS-XI*, Oct 2004.

[25] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen. "Memory Latency-tolerance Approaches for Itanium Processors: Out-of-order Execution vs. Speculative Precomputation," in *Proc.* of *HPCA-8*, Feb 2002.

[26] P. Marcuello, A. Gonzalez, and J. Tubella. "Speculative Multi-threaded Processors," in *Proc. of ICS-12*, Jul 1998.

[27] Q. Jacobson, "UltraSPARC IV Processors," in *Microprocessor Forum 2003*, 2003.

[28] R. S. Chappell, S. P. Kim, S. K. Reinhardt, Y. N. Patt. "Simultaneous Subordinate Microthreading (SSMT)," in *Proc. of ISCA-26*, May 1999.

[29] 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 *Proc. of PLDI'02*, Jun 2002.

[30] T. Kistler, M. Franz, "Continuous Program Optimization: Design and Evaluation", in *IEEE Transaction on Computers*, vol. 50, No. 6, June 2001.

[31] T. M. Chilimbi and M. Hirzel. "Dynamic Hot Data Stream Prefetching for General-Purpose Programs," In *Proc. of PLDI'02*, June 2002.

[32] T. Sherwood, S. Sair, B. Calder. "Phase Tracking and Prediction", In *Proc. of the 30th Symposium on Computer Architecture*, June 2003.

[33] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Runtime Optimization System," in *Proc. of PLDI'00*, 2002.

[34] V. Krishman and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," in *IEEE Trans. on Computers*, pp. 866-880, 1999.

[35] Y. Solihin, J. Lee, and J. Torrellas. "Using a User-Level Memory Thread for Correlating Prefetching," in *Proc. of ISCA-29*, May 2002.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 8, 2009 at 08:55 from IEEE Xplore. Restrictions apply.