In Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA 2007).

Accelerating and Adapting Precomputation Threads for Efficient Prefetching

Weifeng Zhang   Dean M. Tullsen   Brad Calder
Department of Computer Science and Engineering
University of California, San Diego

Abstract

Speculative precomputation enables effective cache prefetching for even irregular memory access behavior, by using an alternate thread on a multithreaded or multi-core architecture. This paper describes a system that constructs and runs precomputation based prefetching threads via event-driven dynamic optimization. Precomputation threads are dynamically constructed by a runtime compiler from the program's frequently executed hot traces, and are adapted to the memory behavior automatically. Both construction and execution of the prefetching threads happen in another thread, imposing little overhead on the main thread. This paper also presents several techniques to accelerate the precomputation threads, including colocation of p-threads with hot traces, dynamic stride prediction, and automatic adaptation of runahead and jumpstart distance. The adaptive prefetching achieves 42% speedup, a 17% improvement over existing p-thread prefetching schemes.

1 Introduction

Software-based prefetching [2, 15, 13, 12, 4, 17, 26] is a powerful technique to hide the increasing latency gap between processors and the memory subsystem. It significantly increases memory level parallelism by overlapping memory stalls with other useful computation.

Software prefetching can be enabled either by inserting prefetch instructions into the original code, which we call *inlined prefetching*, or by running prefetch instructions in a separate thread (i.e. precomputation thread, or *p-thread*), either on a multithreaded or multi-core architecture. While inlined prefetches are typically effective for simple addressing patterns (e.g., strided addresses), p-thread based prefetching has the potential to handle more complex address patterns (e.g. pointer chasing), or accesses embedded in more complex control flow. This is because the prefetching address is computed via actual code extracted from the main thread.

A successful precomputation-based prefetcher must address several challenges. It must be able to determine the proper distance by which the prefetching thread should lead the main thread, and it should have the ability to control that distance. It must create lightweight threads that can actually proceed faster than the main thread, so that they stay out in front. It must prevent p-threads from diverging from the address stream of the main thread, or at least detect when it has happened. This divergence may be the result of control flow or address value speculation in the p-thread. Runaway prefetching may unnecessarily displace useful data, resulting in more data cache misses in the main thread.

In this research, we exploit an event-driven dynamic optimization system to dynamically construct precomputation code, called *p-slices*. P-slices are created from the main thread's hot execution traces stored in the dynamic optimization system's code cache. To clarify, we refer to the prefetching code by the term *p-slice*, and an instantiation of that code running on the processor a *p-thread*. By embedding our p-slice generation in an event-driven dynamic optimization framework, we can overcome the key challenges of thread-based prefetching. We can adapt the same program differently depending on the program's data input and the underlying hardware architecture (e.g., the number of available hardware contexts, the size, organization, and latency of the cache, etc.), and adapt to changing behaviors at runtime (e.g., different loads become problematic, or control flow behavior changes).

Additionally, we employ several techniques to accelerate the p-thread ahead of the main thread. First, we exploit dynamic hardware load stride prediction to speculatively specialize p-slices, allowing for simpler p-slices with lower overhead. Second, we dynamically examine a p-slice to identify its loop induction variable(s) (either inherited from the original hot trace or due to hardware detected strides), allowing us to jump start the p-slice execution a few iterations ahead of the main thread. This technique works even if live-in values in a p-slice do not exhibit any predictable patterns. Third, we leverage the streamlined nature of the hot traces to minimize control flow related overhead. Fourth, we continuously monitor the success of prefetching, and have the ability to adapt and repair the p-slices along several dimensions until the memory latency is covered. Finally, we devise a low overhead mechanism for tracking prefetching addresses to determine when they become out of sync with the main thread. We use this to prevent a p-slice from running away from the main thread.

This dynamic precomputation-thread generation framework achieves 17% improvement over prior precomputation approaches for prefetching, and 11% improvement over our prior dynamic optimization approach for inlined prefetching. We show there is actually synergy between the dynamic p-thread and inlined prefetching, as we get even better results when the two techniques are combined.

The low monitoring and optimization overhead of the event-driven optimization framework allows p-slices to be continuously adapted and repaired. In addition to adapting jump start distance and runahead distance, we can also change whether we use speculative strides or extracted code and whether we use inlining or p-threads to target specific loads and stores.

2 Related Work

There is a large body of prior research in precomputation based prefetching. Precomputation threads may run in a separate thread [3, 6, 19, 5] or in a dedicated hardware engine [14, 1, 18], concurrently with the main thread. P-slice code can be constructed statically [7, 13, 10] or dynamically [6, 19, 12].

Chappell, et al. [3] propose the Simultaneous Subordinate Microthreading (SSMT) architecture which uses microthreads to do prefetching, branch prediction, or even hardware resource management. Roth and Sohi [19] exploit the Speculative Data-Driven Multithreading (DDMT) architecture to perform precomputation to target L1 misses and branch mispredictions. Branch results from helper threads are passed to the main thread via integration. DDMT statically constructs helper threads via offline analysis. Execution based prediction (EBP) [27] adds the ability to loop in the helper threads. Dependence Graph Precomputation by Annavaram, et al. [1] detects the backward slice of a load from among fetched instructions in the instruction queue, and executes those instructions in a separate context to precompute the address.

2.1 Static Precomputation Construction

Collins, et al. [7] manually construct precomputation code to prefetch delinquent loads based on offline profiling. Luk [13] proposes a software controlled precomputation scheme to generate p-slices from the manually annotated program code. A compiler algorithm is developed by Kim and Yeung [10] to automatically generate p-slices at the high level language. Quinones, et al. [16] develop a compiler framework, called *Mitosis*, to generate speculative thread code. Instead of prefetch addresses, Mitosis uses p-slices to compute live-in values for a speculative multithreading architecture. Liao, et al. [11] propose post-pass binary analysis to construct p-slices at the binary level. Kim, et al. [9] reduce p-thread impact on the main thread's performance by judiciously invoking p-threads. Rabbah, et al. [17] embed precomputation code into VLIW traces, using unused issue slots.

Static construction of p-slices fails to support legacy code, does not adapt to program behavior, and typically does not have sufficient information to determine the prefetch distance accurately. Also, it cannot adapt to different architectures without separate binaries.

Our work enables new levels of adaptability by generating and improving p-threads within a dynamic optimization framework. In addition, it also introduces new techniques to push the p-thread in front of the main thread, to further streamline the p-threads, and to detect and recover p-threads that get off track.

2.2 Dynamic Precomputation Construction

Collins, et al. [6] exploit Dynamic Speculative Precomputation (DSP) to identify delinquent loads and construct helper threads via hardware code slicing. Chaining threads, which prefetch delinquent loads in a loop, are created by hardware examination of a buffer containing committed instructions. Slice processors, proposed by Moshovos, et al. [14] similarly create slices for individual loads by examining traces of committed instructions. Our approach leverages the computational power of a separate software thread to construct slices, reducing hardware overhead, and increasing the potential sophistication of slice creation.

More recent work by Lu, et al. [12] dynamically constructs p-slices via a runtime optimizer running on an idle core. A single user-level thread is multiplexed to detect the program's phases, construct the p-thread code, and perform precomputation prefetching. Thus it often limits how long a p-thread can run without missing other critical events in the system.

Our research is distinct in the following ways: (1) Execution, p-slice construction, and p-thread execution all take place in parallel, minimizing overhead to the main thread. This allows more aggressive optimizations, as well as continuous monitoring, repair, and adaptation of the p-slices. (2) We devise an adaptive approach to discover runahead distances for more efficient prefetching. (3) We apply several more aggressive optimizations to accelerate the p-thread ahead of the main thread.

3 Precomputation Thread Construction and Optimization

Our precomputation based prefetching is built on the Trident event-driven dynamic optimization framework [25, 26]. This framework exploits hardware monitoring to detect the program's interesting behavior and trigger compiler optimizations that adapt to that behavior. Optimization and execution occur simultaneously in separate threads, minimizing the performance overhead of the adaptations. This allows much more aggressive optimization than traditional dynamic compilation systems. Typically, a monitoring event triggers an optimization thread, which either creates a hot trace that goes into the code cache to be executed by the main thread, or improves an existing hot trace. Prior optimizations within that framework include dynamic value specialization [25] and inlined software prefetching [26]. In this work, we show that precomputation-based prefetching enables benefits beyond that available to inlined software prefetching (and conventional hardware prefetching); in particular, it has the ability to prefetch more complex memory access patterns. We also demonstrate ways to manage the interaction of the two methods dynamically, in a system that supports both.

We extend the Trident framework by adding a runtime optimizer to perform p-slice construction, acceleration, and adaptation. In this section we describe how we form precomputation slices after the dynamic optimization system detects the hot execution traces in the main thread. We then apply our base optimizations on the p-slices.

3.1 P-Slice Construction

The basic function of the dynamic optimization system is to detect the program's hot execution traces (i.e. basic instruction blocks often executed together). Then further optimizations can be gradually applied to the trace upon new optimization events.

After a hot trace has been executed a number of times, a delinquent load (one which frequently misses in the cache) will generate a *delinquent event*. A load becomes delinquent if its cache miss rate is above a threshold and its average latency for the last *M* misses is also above a threshold [26]. Then the runtime optimizer is triggered to run and perform optimizations on this trace. In this research, the optimization is to construct a p-slice to prefetch delinquent loads within the trace. We choose to construct the p-slice only if the trace contains a self loop.

The goal of p-slice construction is to extract all instructions which are necessary to compute the memory address for a delinquent load, so that we can prefetch the load. To do so, the runtime optimizer first identifies the hot trace containing the event-triggering load. Then it scans the trace to record all delinquent loads inside the trace. The delinquent loads are looked up from the hardware *Delinquent Load Table* (DLT). The DLT is introduced in [26], and its format is shown in Table 2.

Because there is a delay between the event and when the optimizer thread reads the DLT, it is common to have multiple loads from the same trace tagged as delinquent by the time the trace is ready to be optimized. For each recorded load, the optimizer analyzes the hot trace in reverse order, beginning with the delinquent load, to build up a slice of instructions the load depends on, either directly or indirectly. This is called *back-slicing*. As back-slicing continues (going through the loop multiple times), instructions that have been examined in a previous traversal may need to be examined again for dependencies with new instructions in the slice. This process stops when the slice converges, similar to prior research [6, 12]. After back-slicing is done, we apply some base optimizations on the p-slice, such as hoisting loop-invariant instructions out of the loop, and converting matched local load/store instructions to single MOVE instructions. Here, we improve the quality of p-slices from prior research by adding the following new optimizations:

Loop Re-rolling – A hot trace may contain multiple copies of the same code due to loop unrolling done during static compilation. We perform loop *re-rolling* for the p-slice (i.e. removing the redundant loop copies) to reduce duplicated computation inside a p-slice. This optimization increases the granularity at which we can set the prefetch run-ahead distance, since the prefetch distance is always an integral number of iterations.

Object-Based Prefetching – We perform same-object based prefetching, as in our prior work on inline prefetching [26]. Same-object prefetching clusters prefetches falling into the same cache line into a single prefetch. This optimization helps reduce redundant prefetches and reduces the overhead of the p-slice.

Control Flow Removal – We remove control flow from p-slices to streamline the code. Note that a hot trace contains a single path, but multiple exit branches (and branch condition computation), which we can choose not to include in the p-slice. Skipping control flow helps reduce the instruction count inside the p-slice and allows us to further optimize the p-slice. This optimization works for two reasons. First, the prefetching thread often continues to prefetch effectively even when control flow does not match the main thread exactly. Second, we devise a mechanism in the p-slice to recover when the p-thread diverges too much from main thread, which we discuss in Section 3.5. Prior work typically includes all control flow or no control flow in the p-slice. We evaluate this as an optimization in this work, because prior compilation-based techniques [10, 12] are among the former, maintaining all or most control flow in the p-slice.

During p-slice construction, the live-in values used to start the precomputation thread are also identified. Instructions are inserted at the entrance of the p-slice to retrieve live-in values. Then, we lay out the p-slice code as shown in Figure 3.1 (a). Details of the code are explained next.

3.2 P-thread Startup and Termination

During construction, the optimizer also identifies precomputation spawning points and termination points. The hot trace in the main thread is then re-generated with software-based primitives to trigger the generation of a p-thread and to kill the p-thread. The new hot trace has its layout shown in Figure 3.1 (b).

To start up a p-thread, we need to communicate all live-in values from the main thread to the p-thread. Prior research uses a hardware mechanism to perform a fast copy of register state [6] or uses a memory based mailbox mechanism [12].

In this paper, we use a simple software memory buffer to communicate live-in values. The main thread (inside the hot trace) writes the value to the buffer, and the p-thread reads from it. This is similar to the mailbox mechanism, but we designed it in such a way that the communication buffer is relocatable (so that it can potentially be repaired). Thus we map the buffer to a *cold data cache block* to help retain these values in the cache. The cold cache block is one that is lightly accessed during a recent time window, as indicated by hard-

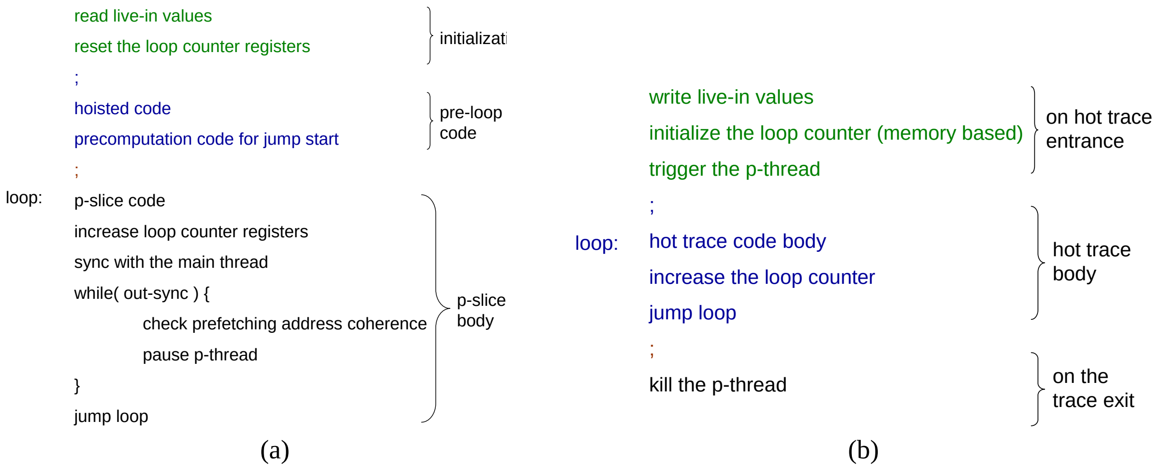

Figure 1. The code layout of (a) precomputation slices and (b) hot traces to trigger/terminate precomputation threads

ware proposed previously [25]. We found that most p-threads only need one or two live-in values. Thus, copying live-in values has minimal overhead for the main thread.

3.3 P-Thread Priorities

To minimize any negative performance impact on the main thread, a p-thread is always triggered to run at low priority during instruction fetching. This is different from some previous systems [6]. This works because in the two cases where you want the p-thread to run at high priority (p-thread startup, and when the p-thread lags behind the main thread) the main thread typically will experience load stalls that give the p-thread ample access. In addition, due to our acceleration, the p-thread often gets a head start from the main thread. These mechanisms are sufficient to allow the p-thread to stay ahead of the main thread, even at a lower priority.

We assume the ICOUNT fetch policy [24], and adjust priorities by imposing a constant bias.

3.4 P-Thread Synchronization

Prefetching too far ahead of the main thread often leads to performance degradation. This is because prefetching not only competes for memory bandwidth, but also may replace useful data. In addition, prefetched data may be overwritten by other loads before being consumed by the main thread.

To prevent this from happening, a p-thread is synchronized to set a limit to how far beyond the main thread it can get. In this research, we devise a new mechanism to let p-threads take the bulk of the responsibility for synchronization and let the main thread run unencumbered. The main thread's only responsibility is to update a loop counter in memory every iteration. The p-thread also keeps a loop counter (in a spare register), and compares it with the counter in memory. If the p-thread's counter exceeds the main thread's counter by more than the *prefetch distance* threshold, the p-thread blocks (or pauses) itself, until the main thread catches up.

To avoid any complicated wakeup scheme, we simply spin-wait for the main thread to update the counter. Since the p-thread is running at a low priority, spin-wait does not consume execution resources needed by the main thread.

Prior research [6, 12] used a fixed synchronization distance. In [12], it also terminates p-threads when the synchronization distance is reached. Our system automatically adapts this distance if the p-thread is not efficient.

3.5 P-Thread Address Coherency

In existing precomputation schemes, once a p-thread is spawned, it often runs along without further interference from the main thread until it is killed. The prefetching address stream is initiated based upon the live-in values.

However, we found that even before the p-thread terminates (i.e. the main thread exits the hot trace), it is often possible for the prefetch stream to diverge from the main thread's address stream. This may be due to store instructions that are left out during the p-slice formation. However, we also see this in our system because of our use of speculated strides (discussed in the next section). They may be correct for hundreds of accesses, but then a discontinuity in how memory was allocated is unaccounted for, and all future accesses are wrong. One hardware solution [5] uses the global history register to check control flow consistency between the main thread and the p-thread. We instead choose a reactive technique with low overhead.

In this study, we propose a low-overhead software solution where the p-thread checks if its address stream is coherent with the main thread. First, it consults the DLT for the last address of a delinquent load. Then it computes the load's expected address, knowing how far it is ahead of the main thread. If the expected address is off from the current prefetching address by more than a given threshold, it is treated as runaway prefetching. This check is only done when the p-thread is about to block because it is too far ahead; thus, in the common case there is no overhead for the check. Because the main thread will experience stalls once the prefetcher diverges, the p-thread will always get ahead and check for the divergence soon after it happens. This approach also prevents us from over-reacting to a quick temporary divergence.

When a divergence is discovered, we try to re-synchronize it with the main thread. This is easy when the base address of the load was obtained as a live-in, or as a constant from the DLT. P-threads can then reset their live-in values from these values, and execution continues after the p-slice initialization code. For more complex address computation in the p-slice, it may not be possible to re-synch with the main thread. In that case, the p-thread terminates itself.

4 Accelerating and Adapting P-Threads

It is critical that we minimize the overhead of the p-slice. This reduces interference with the main thread, but more importantly it is what enables the prefetching thread to stay in front of the main thread. This section describes several techniques to speed up the p-thread. These include co-locating p-slices with hot traces, using speculative strides to simplify complex load recurrences, and jump-starting p-threads.

4.1 P-Thread and Hot Trace Co-Location

To eliminate conflicts between the main thread and the p-thread, we co-locate the p-slice code with its hot trace. When a p-slice is constructed, the run-time optimizer needs to regenerate a new trace with the p-thread trigger and termination instructions inserted. Due to the cold color layout policy, the new trace will be located at code cache blocks which map to the least frequently used cache blocks. Thus, by appending the p-thread code to the end of its corresponding hot trace, we can reduce I-cache misses for the p-thread and often allow at least part of the p-thread code to be prefetched when the hot trace runs.

Additionally, this allows a hot trace and its p-thread to be invalidated in a single operation. If a p-slice needs to be modified or repaired to adjust the prefetch distance, we can often do it in place by just changing constants. If more significant repair is needed (e.g., new delinquent loads identified), we will re-generate both the hot trace and the p-slice, so that they can continue to be co-located.

In experiments not shown in the results section, we examined the interaction between co-location and the relative priority of prefetching threads and the main thread. Without co-location, we needed to give the p-threads high priority to allow them to stay ahead of the main thread. With co-location, keeping the p-threads at low priority gave the best results. This was due to the lack of interference between the two with the co-location optimization (in particular, the main thread not interfering with the p-thread).

4.2 P-slices with Speculative Strides

Using the delinquent load table (DLT) in the dynamic optimization framework, our runtime optimizer can detect data access patterns that occur during execution. We therefore leverage the hardware monitoring mechanism (the delinquent load table, or DLT) from [26] to detect speculative load strides. We found that some loads that have very complex recurrence (resulting in high-cost p-slices) can sometimes be prefetched with a simple strided recurrence instead. For example, a p-slice for a pointer chasing loop, where each load (prefetch) depends on prior delinquent loads can be replaced by a simple strided loop where the prefetches are all independent.

Following is a real code example from *mcf*. The original trace is a pointer chasing loop. After the hardware detects the strides from both pointer loads, we can simplify it as follows.

original p-slice			simplified p-slice with strides		
				LDQ	a2, 104(s2)
loop:	LDQ	a2, 104(s2)	loop:	PREF	(104-120)(s2)
	...			PREF	(0 - 192)(a2)
	LDQ	s2, 0(a2)		SUB	s2, 120
	...			SUB	a2, 192
	BNE	t0, loop		JMP	loop

Here, the first load and the second load in the original p-slice have speculative stride values of -120 and -192, respectively.

With the speculative stride optimization, the new p-slice code can run much faster than its original form. This optimization is made more effective with our new mechanism for address coherence detection and re-synchronization, discussed in Section 3.5.

4.3 P-Thread Jump Starting

Sometimes, the only way to get the prefetch thread ahead of the main thread is to give it a head start. Existing dynamic precomputation schemes (e.g. [12]) typically start p-threads from the same starting point (same iteration) as the main thread.

Our goal is to jump start p-threads multiple loop iterations ahead of the main thread. To do this, we scan the hot trace to identify its loop induction variables, which are also included in the p-slice. We peel them off and hoist them outside the p-slice loop (see Figure 3.1 (a)). Then we either duplicate the peeled code several times, or in many cases simplify it to a single instruction (e.g., if the induction is a constant add). For example, the following p-slice is extracted from an actual hot trace:

original	p-slice	p-slice with jump start
		LDA t1, 256(t1)
##– loop	starts –	##— loop starts —
LDQ	v0, 0(t2)	...
ADDQ	v0, t1, t3	...
PREF	zero, 8(t3)	...
PREF	zero, 72(t3)	...
ADDQ	t1, 128, t1	...

Here, *t1* is the induction variable which has a stride value of 128. Thus, we start prefetching on the third iteration by adding 256 to *t1* before entering the loop.

Prior work also used induction unrolling [19, 6], but only to target a single load `n` iterations ahead. They did not use it in conjunction with loop-based prefetching.

4.4 Self-adapting Runahead and Jump-Start

Prior proposed precomputation prefetchers use fixed runahead distances to synchronize with the main thread. Here, the runahead distance refers to how many loop iterations the p-thread can be in front of the main thread. Finding the correct runahead distance, however, is difficult. It depends on the architecture, on the behavior of all loads in the loop, and the data inputs. Prefetch effectiveness deteriorates rapidly when the prefetch distance is wrong.

In this paper, we use an adaptive approach to discover the runahead distance, and adjust it when the memory behavior changes. A good initial estimate is:

$$Runahead\ distance = \frac{MAX(average\ latency\ per\ load)}{minimal\ execution\ time\ of\ trace} \qquad (1)$$

We get the average load latency from the DLT table. The minimal execution cycles of the trace can be found in the trace performance monitoring hardware of Trident [25]. However, our framework adapts so efficiently that it is not important that our initial estimate be accurate, and we can likely avoid the overhead of computing these values.

The best indication that the p-thread is successfully running ahead of the main thread is that it is often blocked because it runs up against the runahead distance. Thus the p-thread keeps a log in memory of how many times it is blocked because it is too far ahead of the main thread. Since the p-thread has a loop counter (typically in a spare register), we can compute the ratio of the p-threads blocked count relative to how many iterations it runs freely. The jump-start distance may be too small if the ratio is too low (say, 25%). The low ratio means that the p-thread probably cannot run fast enough. Then we dynamically adjust the jump start distance until the prefetched loads are effectively covered. In [26], the repairing technique is directly applied to individual prefetch instructions to patch their prefetch distances. The prefetch distance can be limited by the ISA (e.g. 16-bit offset in the prefetch instruction). Here, we repair the precomputation thread's runahead distance relative to the main thread as well as its jump start distance. These parameters control all prefetch instructions in the p-slice.

One last axis of adaptability is our ability in this framework to do both inlined and p-thread prefetching. We may want to switch between one or the other when (a) no idle hardware contexts are being found available, (b) the overhead of inlined prefetching is affecting the main thread, or (c) the load recurrence is simplified because we detect a stride in hardware. None of these advantages of adaptability are reflected in our current results. Our heuristic for choosing inlined or p-thread prefetching is very simple.

5 Methodology and Background

The performance of precomputation based prefetching is evaluated on a simulated simultaneous multithreading (SMT) processor [24]. We use simulation due to the unavailability of the assumed performance monitoring mechanisms in existing processors. All simulations were run on SMTSIM [22], an emulation-based cycle-level simulator of an SMT processor. We developed a lightweight dynamic compiler to perform our proposed optimizations, which runs as a simulated thread on our simulator, alongside the main thread execution and the generated helper threads.

These techniques apply equally to a multi-core architecture, but we use an SMT platform for two reasons. First, it allows us to prefetch into the L1 cache. Second, it forces us to keep the thread overhead low to minimize interference.

5.1 Baseline Processor Architecture

Our baseline architecture is a 4-issue 20-staged SMT processor with four hardware contexts. The processor fetch policy is ICOUNT2.4, which allows up to four total instructions from up to two threads to be fetched per cycle [23]. The processor includes hardware prefetching stream buffers [21] for data accesses. The stream buffers are guided by a stride predictor, and buffers are allocated using a confidence scheme. The hardware stream prefetcher is configured aggressively, with 8 stream buffers each with 8 entries. This ensures that the easily prefetched loads are already handled in the baseline architecture, and our improvements come only from handling the more difficult ones. Because our prefetcher is dynamic, though, it automatically targets those loads not handled by the hardware prefetcher. For some of the applications in this study, the performance of writes was critically important. Therefore, our baseline processor has the ability to retire stores before they are performed to the cache (to not overstate the effect of store prefetching). We model an 8-entry post write buffer, and only stall when we have more than eight incomplete retired stores. The baseline processor configuration is shown in Table 1.

5.2 Benchmarks

Our results target the memory-intensive programs from the SPEC 2000 benchmarks. We profile all benchmarks and rank them using average miss penalty per load, choosing the top seven FP benchmarks and the top seven INT benchmarks. These benchmarks include *applu, equake, facerec, galgel, mgrid, swim, wupwise*, and *gap, gzip, mcf, parser, twolf, vortex, vpr*. All benchmarks are compiled on the Alpha platform (Digital Unix V4.0F) with the highest compiler optimization options. Note that for applications with fewer cache misses, our optimization provides less performance gain; however, it

Pipeline	20-stage, 256-entry ROB, 224 registers
	Four hardware contexts
	ICOUNT 2.4 fetch policy
Queue Sizes	64 entries each IQ, FQ, and MQ
	Post write buffer: 8 instructions deep
Fetch Bandwidth	4 total instructions
Issue Bandwidth	4 instructions per cycle
	up to 4 Integer, 2 FP, 2 loads/stores
Branch Predictor	2bcgskew, 64K entry Meta and gshare
	16K entry bimodal table
ICache size & latency	64 KB 2-way associative, 3 cycles
L1 size & latency	64 KB 2-way associative, 3 cycles
L2 size & latency	512 KB 8-way associative, 11 cycles
L3 size & latency	4 MB 16-way associative, 35 cycles
Memory Latency	350 cycles
Hardware stream buffers	8 stream buffers; each buffer 8 entries. Stride Predictor.

Table 1. The baseline SMT processor configuration.

Branch profiler	256-entry, 4-way associative. Each entry has a 4-bit counter. Three 16-bit bitmaps to catch branch traces
Delinquent Load Table (DLT)	2-way associative, total 1024 entries. Each entry keeps track of these parameters for each load in a given monitor period: *access count* *cache miss count* *miss latency* *last address* *address stride* Access counter threshold: 256 Miss counter threshold: 8

Table 2. Trident hardware monitoring structures

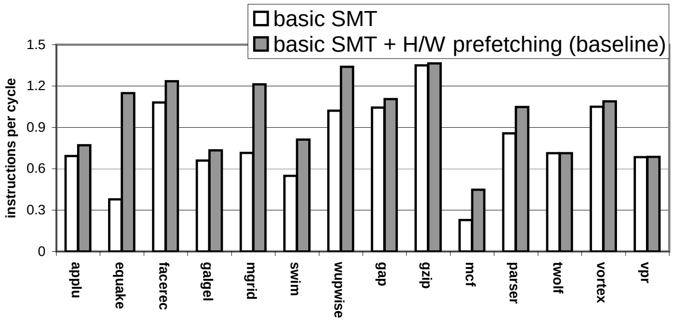

Figure 2. Performance of the baseline SMT processor

also does not introduce much overhead. The observed overhead is typically less than 1% (in Section 6.1), which is consistent with our prior results [26].

Each benchmark is simulated for 100 million instructions. The simulator is warmed up with 5 million instructions before the true simulation starts. During the warmup phase, dynamic optimization and related structures are not enabled. Simulation starts from the single simulation point chosen by SimPoint [20]. Figure 2 shows the base performance when these benchmarks are executed alone on the baseline architecture. On average, hardware stream prefetching achieves 37% speedup over the same baseline configuration without hardware prefetching.

Note that our final instruction throughput results (IPC) reflect only the number of instructions that the *original* code would have executed when running alone.

5.3 The Dynamic Optimizer

Our dynamic optimization system is based upon the Trident framework [25, 26]. Trident is a software/hardware solution to take advantage of the processor's abundant on-chip parallelism (SMT or CMP) to perform concurrent optimization on a thread while it is running. It introduces a conservative extension to existing processors' performance monitoring structure to perform low overhead profiling. The hardware structure detects the program's behavior and triggers optimization events. These events spawn a runtime optimizer in a separate hardware context. Table 2 shows two major hardware structures used in the framework – the branch profiler and the delinquent load table. These are the same structures used in previous research [26].

We extend the Trident dynamic compiler to construct the p-slice code from the program's hot execution traces. The dynamic compiler, running as an optimizing helper thread concurrently with the main thread, dynamically places the p-slice code in the code cache and patches the main thread to trigger and terminate p-threads.

In this research, the main tasks performed by the runtime optimizer are generating and optimizing hot traces, generating p-thread code, or inserting inlined software prefetches into hot traces if the p-thread code cannot be generated. A lightweight runtime optimizer performs optimizations on the streamlined instruction traces. The optimizer is written in *C* and compiled with *gcc -O5* on the Alpha platform. Special care is taken to make the runtime code thread safe.

Upon an optimization event, the runtime optimizer is spawned to run in a separate processor hardware context, concurrently with the main thread. This thread is initialized by our runtime system to set up its starting PC, stack pointer, global data pointer, and the priority according to the information stored in the thread's registration structure. We assume the total initialization requires 2000 cycles. Increasing this assumed latency, e.g. to 4000 cycles, had no observable effect on performance.

The runtime optimizer creates hot traces and the p-slice code, and stores them in a memory buffer (called *Code Cache*). In this study, we assume the code cache has unlimited size, even though Trident has the ability to invalidate individual traces. The code cache management policies are discussed in [8].

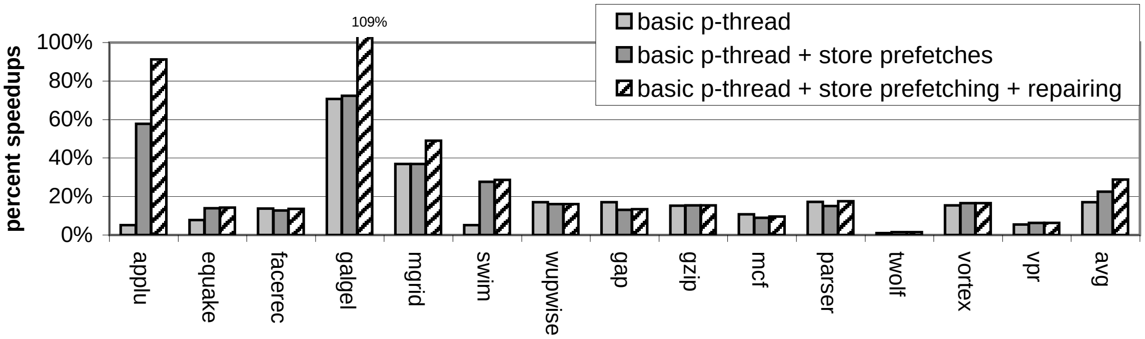

Figure 3. Comparison of basic p-threads with runahead distance repairing

5.4 Baseline Inline Prefetching

In this study, we start with the self-adapting prefetching approach built into the Trident framework in [26]. This approach uses a branch profiler to find hot traces and inserts them into the code cache. The architecture use the delinquent load table find delinquent loads. For these loads, the DLT may provide a stride prediction (if one exists) for the load's miss stream. This is used as part of the prefetch distance calculation.

This approach tracks only the loads executed in hot traces. Prefetch instructions are dynamically inserted into these frequently executed hot traces, using a prefetch distance of one predicted miss stride ahead of the current load's address calculation. The event-driven dynamic optimization system continuously monitors the performance of the software prefetches. If a load is repeatedly classified as delinquent, it's prefetch distance is gradually increased by just patching the prefetch address calculation in the hot trace until it is no longer delinquent or a maximum prefetch distance is used. This allows dynamic search for the appropriate prefetch distance. In Section 6, we compare the precomputation-based prefetcher against this prior approach to aggressive inline prefetching, as well as examining a combined approach.

6 Results

This section evaluates the cost and performance of our dynamically generated precomputation based prefetching technique. The baseline is the architecture described in the prior section. The baseline instruction throughput is shown in Figure 2. All performance improvements shown in this section represent performance gains over aggressive hardware prefetching. All performance improvement is measured as the relative change in effective IPC over the baseline.

6.1 Overhead of the Dynamic Prefetch Optimizer

The precomputation based prefetching technique incurs overhead from the runtime optimizer during the construction of hot traces and the generation of p-slices from the hot traces. The cost depends on how often the optimizer runs, and how much it interferes with the main thread. To measure these factors, we dynamically construct hot traces and p-slices without actually using them. We observe the total performance degradation for the main thread to be 0.6% on average. This low overhead is due to our combination of hardware monitoring of events, and trace compilation and p-slice construction and repairing occurring in separate threads from the main thread (running at a low priority).

6.2 Basic Precomputation Threads

Figure 3 presents the IPC improvement due to precomputation based prefetching. For these results, all the precomputation slices include control flow, as in [12, 10].

The first bar, labeled *basic p-thread*, represents the IPC improvement using existing approaches (e.g. [12]) to generate p-slices. This result assumes a static runahead distance, and does not include store prefetching, which we found critical for some applications, such as *applu* and *swim*. This result assumes a default prefetch distance for loads of 10 loop iterations for all loops, which was found experimentally to provide good results on average. The second bar is similar to the first one except that we enhance it with store miss prefetches. We observe store prefetches boost performance over 50% and 20% for *applu* and *swim*, respectively. The third bar shows the performance improvement when we then apply adaptive p-thread run-ahead distance. This technique continues to monitor and adapt the maximum runahead distance until the memory access is fully covered. We observe speedups from *applu*, *galgel*, *mgrid*, and *swim*. We see modest performance from this approach, but we do not experience the full gains from adaptation until we add further optimizations to our p-slices. This is for two reasons, (1) the maximum runahead distance does not matter unless the p-thread runs fast enough to get far ahead of the main thread, and (2) we have more parameters to adapt beyond runahead prefetch distance. The former is particularly a factor for the integer benchmarks, where heavy control flow slows down the p-threads.

6.3 Accelerating Precomputation Threads

In Figure 4, we show how our acceleration techniques improve the p-thread efficiency. We first remove the control flow from the p-thread. Removing control flow makes the p-slices much more lightweight, but increases the chances

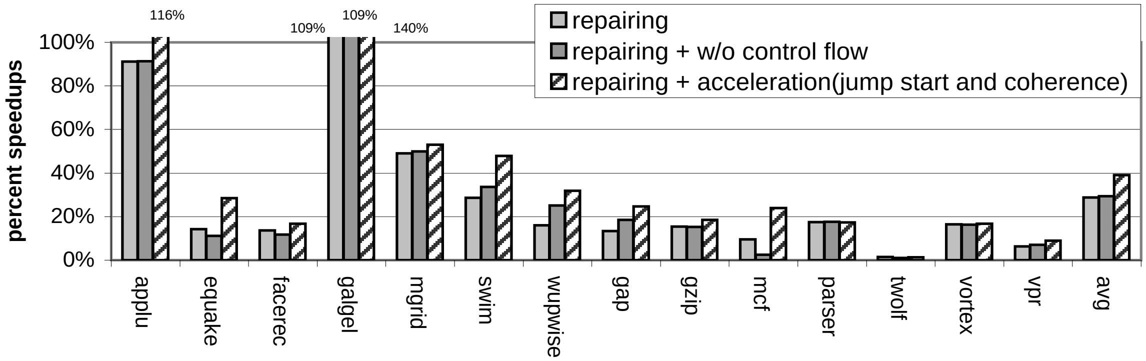

Figure 4. Accelerated p-thread with self-adaptation

that the p-thread will get out of synch with the main thread. This also opens up opportunities for us to exploit speculative strides from hardware prediction and accelerate the p-thread's execution by jump starting it. For comparison, the first bar in the figure is the basic p-thread approach with run-ahead repairing and store prefetching (the third bar in Figure 3). The second bar shows what happens when we remove control flow and their dependent instructions from p-threads. This simplification has some benefits on *swim*, *wupwise*, *gap*, and *vpr*, but causes slowdowns on *equake* and *mcf*. This is because those two programs have more complex memory access recurrences, and are more likely to get out of synch without control flow to guard against it, resulting in runaway p-threads. Because we do not yet have the address coherence mechanism (included in the next bar) to correct these runaway threads, the overall result of removing control flow is a wash.

The third bar adds to the previous result the address coherence detection and the ability to jump start the p-thread a few iterations ahead. The address coherence detection makes sure the p-thread does not get out of sync with the main thread, and if it does, we try to re-synchronize the live-in values for the p-thread. The jump start allows the p-thread to get out in front more quickly. Jump start distances are repaired when p-threads are frequently blocked (i.e., when their potential is not fully released). We observe as much as a 25% performance improvement from *applu*, 40% from galgel, 14% from *mcf*, and 11% from *gap*. The average speedup is 39%, which is 17% better than previous techniques (including store prefetches).

6.4 Synergy with Inline Prefetching

Here we compare and combine the performance of our precomputation thread acceleration and adaptation with inlined prefetching using our previous optimization framework [26]. The result is shown in Figure 5. For comparison, the first bar and the third bar in the figure are taken from Figure 3 and Figure 4, respectively.

The second bar (labeled *inlined prefetching*) in the graph shows the results for the inlined software prefetching from the prior work. This is an aggressive dynamic inline prefetching system that takes full advantage of the Trident framework, including dynamic detection of delinquent loads, stride prediction of pointer loads, and dynamic adaptation of the prefetch distance. It is called inlined prefetching, since the prefetches are inserted into the hot trace. No p-thread prefetching is performed, and all hot traces potentially have in-lined prefetching applied to them.

Although there is significant redundance between the two prefetching techniques, we find that neither subsumes the other, and there is advantage to using both in a system. That will be especially true in a dynamic optimization system like Trident, which can adapt the prefetching approach based on runtime behavior. The fourth bar (*repairing* + *acceleration* + *inlined*) in Figure 5 combines our precomputation prefetching with inlined prefetching.

For this result, we apply a very simple heuristic to decide which prefetch approach to apply. We use precomputation on all looped hot traces, and apply inlined prefetching when there is no loop. This exploits the low overhead (on the main thread) for p-thread prefetching, but does not incur the p-thread startup cost when prefetching a single load later in the hot trace. Thus, for this result we create p-threads for all hot traces that contain a loop back edge to the start of the hot trace. For those hot traces no inline prefetches are inserted. We only insert inline prefetches for the remaining hot traces. The benchmark *gap* receives the most benefits from this combination, since two of the critical loads actually fall on a non-looped trace. Overall, the combination gains 11% better performance than our previous aggressive inlined prefetching alone.

There are times, however, when inlined prefetching may still outperform precomputation. One such case is when the average loop count is low. In that case, the startup cost of the p-thread does not allow it to do useful prefetching in time. Because we find it easier to detect these cases once the hot trace and p-slice are optimized and in place, we initially target all looped accesses with p-threads, then convert selected hot traces to inlined prefetching. The last bar (*repairing* + *acceleration* + *conversion*) shows performance variation when we convert qualified p-thread prefetching to inlined prefetching. We do this conversion when all delinquent

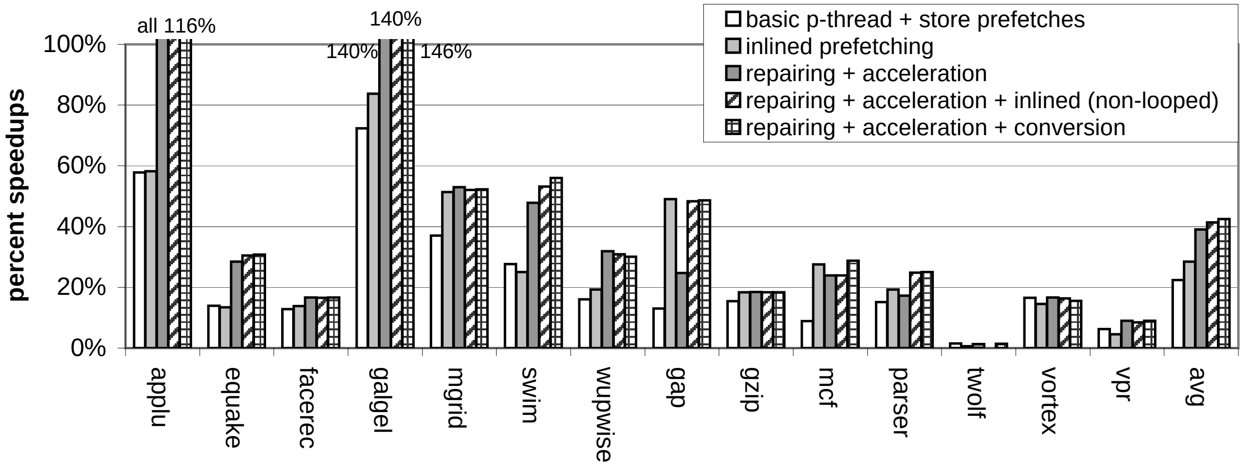

Figure 5. Comparison of accelerated p-thread prefetching with inlined prefetching

loads within a trace have strided patterns and the trace has a relatively small loop count (e.g. less than 5). In this case, we terminate the p-thread, and instead do inlined prefetching for the hot trace. Most benchmarks do not show any slow down, and a few benefit. We notice that conversion does not occur in *applu* because its loop has a large loop count. For *mcf*, however, conversion gives much better performance. This is because one of the critical loops in *mcf* has a very small loop count, and p-threads are frequently spawned and killed before doing useful work. In this case, inlined prefetching does a better job.

Further work to adaptively arbitrate between inlined and precomputation prefetching is ongoing. Perhaps the most compelling reason to have this ability, though, is to adapt to changes in the availability of free hardware contexts. Already having demonstrated the ability to convert between the two, we need only add an event that would trigger when a particular p-thread spawn instruction consistently failed to start a p-thread due to the lack of an available context. In that case, that hot trace would be converted to use inlining. If hardware contexts become available at a later date, an aging mechanism that forces hot traces to be regenerated on a regular basis would ensure that the system moves back to aggressive use of p-threads. We do not model this advantage, though, as our measurement environment does not reproduce the dynamic nature (jobs coming and going) of a real system.

One limitation of our use of the Trident framework is that we are limited to accesses that occur within hot traces. Figure 6 shows the distribution of load misses in and out of hot traces. The difference between the height of the bar and 100% represents cache misses that occur outside hot traces. The percent of cache misses within a hot trace are broken up into those that were found in loops (looped) by the dynamic optimizer, and those not in loops. The results show that some programs have most of their misses within hot trace loops, whereas others do not. Applying both p-thread prefetching for hot trace loops and in-lined prefetching for the other hot traces allows us to efficiently address both kinds of misses.

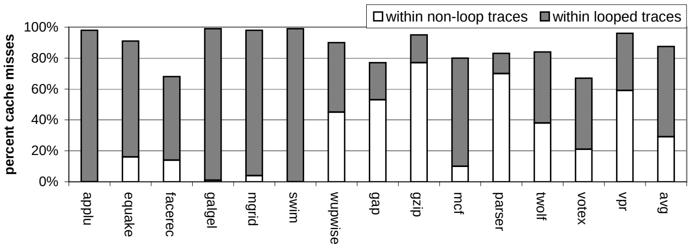

Figure 6. Dynamic load misses within hot traces

We observe that over 85% of load misses are within dynamically generated hot traces. Among them, nearly 58% of misses have the potential of being prefetched by using precomputation. Note that *vortex* and *gap* have relatively low miss coverage. This is in large part because of the low dynamic coverage of the hot traces. However, most critical cache misses from *gap* are covered by hot traces. This allows us to prefetch them via precomputation, inlined prefetching, or a combination.

7 Conclusion

Precomputation based prefetching is a powerful technique to hide long memory latencies, especially for complex load behavior. The goal is to create a precomputation approach that allows the p-threads to run far enough ahead of the main thread to hide all of the memory latency.

In this paper, we extend the event-driven, multithreaded dynamic optimization framework, *Trident*, to enable precomputation based prefetching by dynamically constructing p-thread code from hot traces and accelerating p-threads for efficient execution. We extract the hot trace loop induction variables and duplicate the induction computation ahead of the p-slice loop so that we can jump start the p-thread multiple loop iterations ahead. We adapt the prefetch runahead distance and the jump start distance dynamically, until each memory access is most effectively covered. We also propose a new mechanism to keep the p-threads in sync with the main thread of execution by comparing the prefetched address stream with the main thread's address stream – this is done with virtually no cost in the common case.

Our acceleration technique combines software code analysis with hardware performance monitoring to improve the efficiency of p-threads. Thus, we can exploit some patterns static software systems cannot, and can adapt to the actual runtime behavior of individual loads. Overall, we achieve an average 42% speedup relative to the hardware stride based prefetcher, and it is 17% better than previous dynamic precomputation approaches. In addition, using precomputation for loops and dynamic prefetching for non-loops achieves 11% speedups over our prior dynamic prefetching technique.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported in part by NSF grant CCF-0541434 and Semiconductor Research Corporation Grant 2005-HJ-1313.

References

[1] M. Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precompuation. In *Annual International Symposium on Computer Architecture*, July 2001.

[2] B. Callahan, K. Kennedy, and A. Porterfield. Software prefetching linked data structures in java. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[3] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *Annual International Symposium on Computer Architecture*, 1999.

[4] T. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general purpose programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[5] J. Collins, S. Sair, B. Calder, and D. Tullsen. Pointer-cache assisted prefetching. In *35th International Symposium on Microarchitecture*, 2002.

[6] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic speculative precompuation. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[7] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[8] K. Hazelwood and J. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *International Symposium on Code Generation and Optimization*, March 2004.

[9] D. Kim, S. Liao, P. Wang, J. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. Shen. Physical experiment with prefetching helper threads on intel's hyper-threaded processors. In *International Symposium on Code Generation and Optimization*, 2004.

[10] D. Kim and D. Yeung. Design and evaluation of compiler algorithm for pre-execution. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[11] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[12] J. Lu, A. Das, W. Hsu, K. Nguyen, and S. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. *38th International Symposium on Microarchitecture*, 2005.

[13] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[14] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice processors: An implementation of operation-based prediction. In *International Conference on Supercomputing*, June 2001.

[15] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[16] C. Quinones, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzales, and D. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

[17] R. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W. Wong. Compiler orchestrated prefetching via speculation and prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[18] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[19] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, July 2001.

[20] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[21] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, 2000.

[22] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[23] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[24] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[25] W. Zhang, B. Calder, and D. Tullsen. An event-driven multithreaded dynamic optimization framework. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2005.

[26] W. Zhang, B. Calder, and D. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *International Symposium on Code Generation and Optimization*, March 2006.

[27] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, July 2001.