UNIVERSITY OF CALIFORNIA, SAN DIEGO

Event-Driven Multithreaded Dynamic Optimization

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Weifeng Zhang

Committee in charge:

Professor Bradley Calder, Chair
Professor Dean Tullsen, Co-Chair
Professor Paul Chau
Professor Ranjit Jhala
Professor Chandra Krintz

2006

The dissertation of Weifeng Zhang is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

Co-Chair

University of California, San Diego

2006

DEDICATIONS

I dedicate this work to my family, the most important people in my life. To my wife, Xuejun, for her love, patience, and support throughout my study at UCSD. To my son, Brandon, and my daughter, Sophia, for bringing love and joy to my life. To my parents, my brother, and my sisters for their support and encouragement.

I owe sincere thanks to my advisors, Dr. Bradley Calder and Dr. Dean Tullsen, for their excellent guidance, inspiration, and support. I would also like to thank the other committee members, Dr. Paul Chau, Dr. Ranjit Jhala, and Dr. Chandra Krintz for their suggestions and final approval to my thesis.

I also owe significant thanks to all members in the Processor Architecture and Compilation Lab, Gilles, Erez, Jeremy, Satish, Rakesh, Jeff, Weihaw, Michael, Subhra, Jack, Ganesh, Matt, Allen, Anthony, Steve, Leo, and Cristiano, for fruitful discussions and making my time at UCSD enjoyable. Needless to say, this is the best time of my life.

高者抑之, 下者举之;
有余者损之, 不足者补之.

It is lowered if too high; It is raised if too low.

It is shortened if too long; It is lengthened if too short.

– Lao Tsu (500 BC), ancient Chinese philosopher

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Portions of Chapters IV and V reproduce the material from the paper, *Weifeng Zhang, Brad Calder, and D.M. Tullsen, "An Event-Driven Multithreaded Dynamic Optimization Framework"*, in the proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005), September 2005, Saint Louis, MO. The dissertation author was the primary researcher and author and the co-authors involved in the publication [142] directed, supervised, and assisted in the research which forms the basis for that material.

The text of Chapter VI is in part a reprint of the material from the paper, *Weifeng Zhang, Brad Calder, and D.M. Tullsen, "A Self-Repairing Prefetcher in an Event-Driven Dynamic Optimization Framework"*, in the proceedings of the 4th International Symposium on Code Generation and Optimization (CGO 2006), March 2006, New York, NY. The dissertation author was the primary researcher and author and the co-authors involved in the publication [143] directed, supervised, and assisted in the research which forms the basis for that material.

VITA

1968	Born, Henan, China
1987	B.S. in Electronics and Information Science Wuhan University, China
1990	M.S. in Computer Science and Application Chinese Academy of Science, Beijing, China
1996	M.S. in Physics University of California, Irvine
1996-1997	Software Engineer Phoenix Technologies, Ltd San Jose, California
1997-1999	Senior Software Engineer Phoenix Technologies, Ltd San Jose, California
1999-2002	Principal Software Engineer Phoenix Technologies, Ltd San Jose, California
2002-2006	Research Assistant Department of Computer Science and Engineering University of California, San Diego
2006	Ph.D. in Computer Science University of California, San Diego

PUBLICATIONS

Weifeng Zhang, Brad Calder, and D.M. Tullsen, "An Event-Driven Multithreaded Dynamic Optimization Framework", *in the 14th International Conference on Parallel Architectures and Compilation Techniques*, (PACT 2005), September 2005, Saint Louis, MO

Weifeng Zhang, Brad Calder, and D.M. Tullsen, "A Self-Repairing Prefetcher in an Event-Driven Dynamic Optimization Framework", *in the 4th International Symposium on Code Generation and Optimization*, (CGO 2006), March 2006, New York, NY

ABSTRACT OF THE DISSERTATION

Event-Driven Multithreaded Dynamic Optimization

by

Weifeng Zhang
Doctor of Philosophy in Computer Science
University of California, San Diego, 2006
Professor Bradley Calder, Chair
Professor Dean M. Tullsen, Co-Chair

Dynamic optimization has been proposed to overcome many limitations of static optimization, such as inaccurate assumptions about the underlying processor architecture and lack of adaption to the program's runtime behavior. However, existing dynamic optimization systems often impose high runtime overhead (software systems) or great hardware complexity (hardware systems), and only have limited runtime adaptability.

This thesis proposes a new model of optimization, where optimization is triggered by hardware optimization events and is performed concurrently on an application while it is running. We introduce an event-driven multithreaded dynamic optimization architecture, called *Trident*. Trident is a software/hardware solution which strives to reduce software runtime overhead as well as reduce hardware complexity and inflexibility. Trident takes advantage of two key features in modern processors, abundant chip-level parallelism (through Simultaneous Multithreading, Chip Multithreading, or a combination) and increasing hardware support for runtime performance monitoring. Trident proposes generic, lightweight extensions of the hardware monitoring mechanisms to profile the program's execution behavior. Hardware triggers an event for optimization upon detection of

any interesting behavior. Lightweight helper threads are spawned to process these events, in parallel with the main thread. The combination of event-driven and concurrent optimization makes Trident extremely low overhead in both profiling and optimization. This enables Trident to perform more expensive optimizations than existing dynamic systems, and perform continuous recurrent optimizations without fear of performance loss.

The power and flexibility of Trident enable many types of optimizations. In this thesis, we demonstrate it with an aggressive optimization, called speculative dynamic value specialization. We also demonstrate Trident's power of continuous, gradual optimization by improving traditional software prefetching to better attack the classical memory wall problem. These approaches include adaptive dynamic software prefetching via self-repairing and accelerating precomputation based prefetching.

I

Introduction

This dissertation explores Event-Driven Multithreaded Dynamic Optimization, a new optimization model, to enable continuous recurrent optimizations which are difficult to perform in traditional dynamic optimization systems. Static compilation and optimization techniques have been well studied in the past few decades [75, 2, 137, 1, 47, 96]. Static optimization and code generation often rely on assumed knowledge of the underlying processor architecture, such as the number of physical registers [22, 38], the number of functional units, cache hierarchy [80, 52, 17], and processing element (*PE*) organizations [51, 79, 69]. Static compilation has the advantage of doing sophisticated and global optimization without suffering runtime overhead. However, it is increasingly difficult for static compilers to make accurate assumptions about the underlying processor architecture. Inaccurate assumptions produce sub-optimal performance. Furthermore, the uniform assumptions made during static compilation do not always work well. This is because a program not only has different execution behavior with different input, but also exhibits phased behavior even with the same input [124]. A particular optimization performed by the static compiler may be optimal in one phase, but suboptimal in another phase of execution.

Dynamic optimization [7, 29, 39, 85, 10, 32, 104, 112, 13, 92, 121] has

emerged as an alternative solution to the above problems. Since a program is dynamically optimized while it runs, optimizations can be specifically customized to the underlying machine architecture, and they can automatically adapt to the program's changing behavior at runtime. While dynamic optimization has many advantages over static optimization, it also imposes several new challenges.

- **Runtime Overhead:** Dynamic optimization suffers three types of overhead, which are absent in static optimization. First, dynamic optimization requires runtime profiling to identify the program's execution behavior and runtime analysis to detect any behavior patterns. The accuracy of runtime profiling often determines the ultimate performance of dynamic optimization. Second, the runtime optimizer competes for execution resources with the main thread of the program. Thus, optimizing the program on the fly may unnecessarily impact the performance of the main program. To minimize this overhead, dynamic optimization is often judiciously performed on the common cases. It is often known that a program spends 90% (or more) of its execution time in 10% (or less) of its code. A common technique for reducing overhead is to classify the dynamic instruction streams into frequently-executed (hot) streams and infrequently-executed (cold) streams. When dynamic optimization is applied on the hot traces, the highest benefits are achieved and the cost is amortized if the hot traces are executed many times. Thus, a companion challenge for dynamic optimization is how to select hot traces with high dynamic coverage. Finally, because only a part of the program is dynamically re-optimized, overhead also occurs when switching between the optimized and un-optimized code during execution.

- **Program control:** It is also important to efficiently maintain control over the running program, in order to dynamically alter its code, and switch between the optimized code and un-optimized code. In the purely software-

controlled dynamic optimization systems, this is usually difficult to achieve without severely penalizing the performance.

- **Optimization Adaptation:** The optimized code in a program may need to be re-optimized as the program's behavior changes over time. Optimizing the program dynamically, but staying unchanged afterwards, has the same pitfall as static optimization. Adapting the optimization to the program's varying behavior requires continuous profiling and recurrent optimization, which can be very expensive.

  The optimized code is either stored in a memory buffer (in software systems) or in a hardware cache (in hardware systems). Managing the optimized code imposes new software overhead or hardware complexity. The management policy may also reflect how fast the dynamic optimization adapts to the program's behavior.

However, existing dynamic optimization systems essentially have the same limitation as the static systems. In these dynamic systems, profiling, optimization, and execution are interleaved, but still done one at a time. Thus, execution stalls when profiling or optimization occurs. Especially context switches are incurred when switching from one to another.

I.A Event-Driven Multithreaded Dynamic Optimization

This thesis proposes a new model of optimization, where optimization is triggered by hardware optimization events and is performed concurrently with the program's execution. By allowing profiling, optimization, and execution to occur in parallel, this model eliminates the above overhead, and enables continuous and adaptive optimization.

We implement this concurrent model as an event-driven multithreaded

dynamic optimization framework, called *Trident*. Trident is a software/hardware solution to enable low overhead and fast optimization with efficient hardware support. Trident exploits two features present in many modern processor architectures: increasing hardware support for runtime monitoring of execution and the ability to execute multiple threads, either through chip multiprocessing [56], hardware multithreading [133], or a combination [68]. Trident exploits a hardware multithreading processor, using an otherwise idle hardware thread to concurrently optimize a thread that is running.

This thesis also proposes conservative extensions to the existing hardware support to identify performance-critical events to trigger dynamic optimization. It provides efficient support to enable Trident's optimization without introducing much complexity to the processor's performance monitoring mechanism (e.g. Itanium). Thus, optimization that runs concurrently with execution and monitoring provides an extremely low-overhead dynamic optimization system. It significantly reduces overhead inherent to most prior systems, and can enable very aggressive optimizations.

The two key features of our optimization system are:

- **Performance and event monitoring can be done with no software overhead**. This allows more frequent monitoring and higher coverage of the executable. It also allows monitoring to continue with no overhead during and after optimization, allowing more opportunities to repair or back out of bad optimizations.

- **Event-driven low-overhead optimization**. Because optimization happens in response to hardware optimization events, optimizations are easily handled by spawning a lightweight helper thread, which does not interrupt the main thread's execution. Trident avoids the profile polling and associated software overhead (as in ADORE [84]). Thus, concurrent execution

and optimization allow the framework to explore much more aggressive optimizations without significant fear of performance loss, even if the code being optimized is short lived. Low overhead profiling and low overhead optimization also make it possible to enable continuous and gradual optimization, which is difficult to do in the existing systems.

I.B Optimizations with the Trident Framework

Our framework focuses on efficient dynamic optimization for a multi-threaded processor – in this thesis, the hardware platform we examine is the Simultaneous Multithreaded (SMT) processor [133].

Low overhead profiling and optimization provide Trident more freedom to re-consider some design tradeoffs in traditional dynamic optimization systems. In this thesis, we examine the benefit of using Trident to perform basic compiler optimizations, to guide code layout for the instruction cache conflict reduction, and to enable an architectural specific optimization (reduction of the Return Address Stack misprediction).

Trident is flexible enough to enable a variety of optimizations at once, and we demonstrate it in this thesis with dynamic value specialization combined with these basic dynamic optimizations. In addition, we demonstrate Trident's ability to enable continuous and gradual optimizations by re-examining the classical *Memory Wall* problem [139, 89]. The memory wall problem has been studied via pure hardware approaches [125]. Conventional dynamic systems (e.g. ADORE [84]) also try to solve this problem by dynamically inserting software prefetches according to its runtime memory behavior. In this thesis we show that Trident's gradual re-optimization can further improve these systems to achieve significantly better performance.

I.B.1 Dynamic Value Specialization

Value specialization is a compiler technique which makes a special version of code if a procedure (or an execution trace, in general) frequently takes the same parameter values during each invocation. Depending on the actual value used for specialization, the specialized code could be dramatically reduced from its original version. However, the compiler has to insert extra code to recover when the parameters does not take the predicted value. This is called *misspecialization*.

In this thesis, we study a form of speculative dynamic code specialization. The program's execution traces are speculatively specialized with semi-invariant runtime values, which are identified via runtime profiling. Values are dynamically verified to ensure the correctness of the program. Recovery is automatically performed using the existing mis-speculation hardware. The advantage of this technique over compiler-based value specialization is the ability to specialize on values identified dynamically during execution, to adapt value specialization as the application changes behavior, and to recover from mis-speculation quickly in hardware.

Speculative value specialization benefits from two factors - value prediction and value specialization. Traditional hardware value prediction takes advantage of value locality, and breaks true data dependence chains by directly predicting load values [83, 136, 145, 18]. With load value prediction, instructions that depend on load values are executed speculatively. The benefit of value prediction comes from reducing the program's critical dependence path of execution, which is of critical importance when loads that miss in the cache are part of the dependence chain. However, a significant missed opportunity for dynamic value prediction techniques is that unnecessary, or unnecessarily complex, instructions are executed after the value is "known" (speculatively). These are instructions

the compiler could have eliminated or reduced if the value had been a static constant.

Thus, by doing speculative value specialization in software, we end up expanding benefits beyond the very conservative use of value locality. Dynamic value specialization allows compiler-like optimizations to be applied to semi-invariant runtime values. Anytime value locality is able to make value prediction useful, value specialization significantly enhances its effectiveness.

I.B.2 Adaptive Dynamic Software Prefetching via Self-Repairing

The performance of modern processors is increasingly dominated by the widening latency gap between processors and memory subsystems. This is called the *Memory Wall* problem. As the microarchitecture pipeline is getting deeper and the clock frequency is getting higher, it puts even more pressure on the memory system [129]. Prior research also shows that growing cache size does not always improve the program performance[118]. This is especially true for the data cache due to diversified program memory behaviors.

One way to bridge the latency gap is to prefetch load values into the cache, which attempts to overlap the memory latency with the execution of other useful instructions in the same program. This essentially decreases the observed latency, increases memory level parallelism, and allows cache-hit dominated performance even when the working set is larger than the cache. Software based prefetching [20, 95, 87, 138, 85, 31, 62, 109] has been shown to be a promising technique to address the memory wall problem, and all modern high-performance instruction set architectures provide support for software prefetching.

Software prefetching should meet these criteria in order to be effective:

- Prefetches should be accurate. That is, prefetching should target the loads that are actually missing in the cache. Unnecessary prefetches may reduce

the effective memory bandwidth.

- Prefetching should be timely and far enough ahead of time to fully cover the miss latency. Prefetching a load too late will prevent the prefetch from hiding the entire memory latency. However, prefetching too far in advance may unnecessarily displace useful data, increases the likelihood that prefetched data will be replaced, and also increases the likelihood of prefetching unneeded data due to unexpected intervening control flow. Therefore, we want to prefetch a load just in time, so its value appears in the cache right before it is needed. This is the goal behind our study to enable adaptive (self-repairing) prefetching.

- Prefetching address computation should have low overhead. Determining where to prefetch should be fast and easy.

Static compilation does not always provide efficient prefetching, even with offline profiling. For the first criterion, which loads are critical and whether they will miss in the cache cannot be determined accurately offline, since what misses in the cache largely depends on the underlying cache size and organization. For the second criterion, the average latencies of cache-missing loads will vary across different data inputs. Additionally if the code runs on different machines, how far ahead to prefetch a load for one machine will be inappropriate for the other.

Existing dynamic optimization systems (e.g. ADORE [84]) overcome the above limitations by dynamically inserting software prefetches to target true cache misses. Prefetching distances (i.e. *how far to prefetch ahead*) are estimated using average memory latency. However, due to their high runtime overhead, these prefetch instructions stay unchanged during a very long stable phase after being inserted. There are two main limitations with this approach: (1) Due to

heavy interaction between prefetches and neighboring load instructions, prefetching distances are not necessarily correct through one profile estimation. Incorrect prefetching distances may only lead to *partial prefetching*. That is, the load latency is only partially hidden by prefetches. (2) Prefetching does not adapt to the runtime behavior as quickly as the program execution changes phases.

The adaptive prefetching proposed in this thesis overcomes these limitations of both static and dynamic approaches, by re-evaluating the effectiveness of the inserted prefetches through continuous hardware monitoring. Prefetches may be re-adjusted, or may be removed altogether, according to the program's runtime behavior. Like hardware prefetching, our technique operates on dynamic information rather than static information to initiate prefetching, and it works on legacy code without sacrificing software compatibility with past and future processors.

I.B.3 Accelerating Precomputation Based Prefetching

Software-based prefetching can be enabled either by inlining prefetches inserted into the dynamically-generated code as described in Section I.B.2, or by running prefetch instructions in a separate thread (e.g., precomputation thread, or *p-thread*). While inlined prefetching is 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).

Precomputation is a technique to speculatively execute small code traces to compute load addresses and then prefetch these loads [36, 86] or determine branch directions [116] before the main thread reaches those instructions. Precomputation code traces, called *p-slices* [36], can be constructed statically [37, 87, 71, 109], or dynamically [36, 116, 86]. The p-slice is a distilled version of the main thread code. When a p-slice is instantiated to run on the processor, it is

referred as a *p-thread*.

P-threads usually run in a separate hardware thread, in parallel with the main thread. In this research, we focus on precomputation based prefetching to hide memory latency.

A common problem in existing precomputation based prefetching schemes is that a p-thread often cannot run sufficiently ahead of the main thread. Though the p-slice is extracted from the main thread's code, it may be no simpler than the main thread when the load behavior is complex. Thus, the p-thread may run no faster than the main thread. At the same time, cache misses occurring inside the p-thread also impede the p-thread from running further ahead. This is especially true when there is pointer chasing behavior inside the p-thread. Another problem associated with precomputation based prefetching is that decoupling of the prefetching thread from the main thread allows the prefetching thread's address stream to possibly diverge from the main thread, if the prefetcher is based on control flow or address speculation. Runaway prefetching may unnecessarily displace useful data, resulting in more data cache misses in the main thread. Both problems above can dramatically reduce the effectiveness of precomputation based prefetching.

In this research, we dynamically construct p-slices from the main thread's hot execution traces. By embedding our p-slice generation in an event-driven dynamic optimization framework, we can overcome several key challenges of thread-based prefetching. We can adapt the same program differently depending on the input and the underlying hardware architecture, we can adapt to changing behaviors at runtime (different loads become problematic, control flow behavior changes).

Additionally, our low overhead, multithreaded optimization enables a few sophisticated techniques to accelerate the p-thread ahead of the main thread.

For example, we exploit dynamic hardware load stride prediction to speculatively specialize p-slices, allowing for simpler p-slices with lower overhead. Furthermore, we can dynamically perform code analysis on a p-slice to extract precomputation code from the p-slice, allowing us to jump start the p-slice execution a few iterations ahead of the main thread. Finally, during dynamic p-slice construction, we can insert code to enable a low overhead mechanism for tracking prefetching addresses to determine when they become out of sync with the main thread. This prevents a p-slice from running away from the main thread.

I.C Thesis Organization

The reminder of this thesis is organized as follows. Chapter II discusses prior research related to dynamic optimization. Chapter III describes details of our even-driven multithreaded dynamic optimization architecture. Chapter IV presents the base optimizations with our optimization architecture. These include hot trace formation, classical compiler optimizations, code relayout, and branch misprediction reduction due to the return address stack misalignment. Chapter V shows the ability of our framework for more aggressive optimizations by speculatively specializing hot traces using semi-invariant runtime values. Chapter VI demonstrates its ability of continuous and recurrent optimization via adaptive, self-repairing prefetching to target the memory wall problem. Chapter VII extends its ability to accelerate the precomputation based prefetching. We summarize this thesis in Chapter VIII and present a few research directions with our dynamic optimization architecture.

II

Background

This thesis proposes an event-driven multithreaded dynamic optimization framework, called *Trident*. It enables low overhead, continuous, and recurrent optimizations on applications. Our framework is based in part on a large body of prior research in dynamic optimization. In this chapter, we focus on summarizing prior research on dynamic optimization in both software and hardware based systems, and comparing it against our proposed framework. The related work in the areas where we apply the Trident framework is discussed in individual chapters to demonstrate how Trident improves these techniques. Related work in these applications includes value specialization, hardware and software inlined prefetching, and precomputation based prefetching.

II.A A Brief History of Dynamic Optimization

Dynamic optimization, as often embedded in binary translation [4], can be traced back to the middle 60's when the IBM/360 series was first introduced. Binary translators often optimize the code for new underlying architectures using the program's dynamic behavior during translation. Early forms of dynamic optimization include ISA remapping, basic block reordering, limited memory color-

ing, and code specialization (specialized with the procedure's parameter values). As processor technologies evolve, more sophisticated binary translation and dynamic optimization techniques are developed. Examples of these systems include Digital FX!32 [30], HP Aries [141], IBM DAISY [42], Transmeta Code Morphing System (CMS) [41], and the Intel IA-32 execution layer [10]. In 1996, Sun released the Java JDK. Java just-in-time (JIT) compilation [39, 131, 15], which delays all compilations until runtime, gradually gained popularity. On a different path, selective dynamic compilation, such as DyC [93], was proposed to restrict dynamic compilation to only selected portions of code. The code, identified by the user annotation or source language extension, is optimized by a static compiler as much as possible to generate templates that are invoked at runtime by a specialized dynamic compiler. A truly dynamic optimization system, called *Dynamo*, was developed at HP labs in 2000. This is one of the first such systems which demonstrated that a statically optimized native program could still benefit from runtime optimization.

II.B Software Based Dynamic Optimization

We will first summarize prior research on the software dynamic optimization systems. We divide these systems into two groups – those that optimize native binaries and those that perform optimization while doing ISA translation.

II.B.1 Native Binary Optimization Systems

There have been several software dynamic optimization systems proposed in prior research, such as Dynamo [7], DynamoRIO [13], and Mojo [29]. Dynamo intends to offer a client-side performance delivery mechanism, and mainly targets single-threaded applications on the HP PA-RISC architecture. Mojo, developed at Microsoft, targets the x86 architecture and desktop computing en-

vironment, which has rich multithreaded applications.

Dynamo provides transparent dynamic optimization on binaries without any support from user annotation, operating system, or underlying hardware architecture. Dynamo detects a program's frequent instruction execution streams (called *hot traces*) by directly interpreting the statically compiled binary. Here, a hot trace is a single-entry-multiple-exit instruction sequence. Dynamo exploits optimistic detection strategies to quickly identify hot traces in order to reduce the costly overhead of interpretation. After a hot trace is identified, Dynamo stops interpreting the current application and starts optimizing the hot trace. Interpretation resumes after the optimization is done. When Dynamo encounters the same instructions during future interpretation as the optimized trace, it does a user-level context switch so that the optimized code can be executed natively (instead of interpreted). Dynamo interleaves code interpretation and native execution, as shown in Figure II.1.

Mojo [29] has a similar optimization flow as Dynamo, but it targets multithreaded Windows applications. Mojo makes two design improvements over Dynamo. (1) Mojo uses the Just-in-Time direct translation approach to simulate a basic block, instead of code interpretation in Dynamo. Direct translation involves copying a basic block into a buffer, augmenting the buffer with control instructions at the end, and executing the instruction buffer. The control instructions will ensure the control returns to the Mojo dispatcher after the buffer is executed. (2) Mojo partitions the *Code Cache*, where optimized traces are stored, into several sections. It fills the sections one at a time. Optimized traces are invalidated gradually by flushing out the oldest section. This policy incurs lower trace re-optimization overhead and has quicker adaptation to the program's behavior than the global flushing in Dynamo.

A common attribute of these systems is that optimization is typically

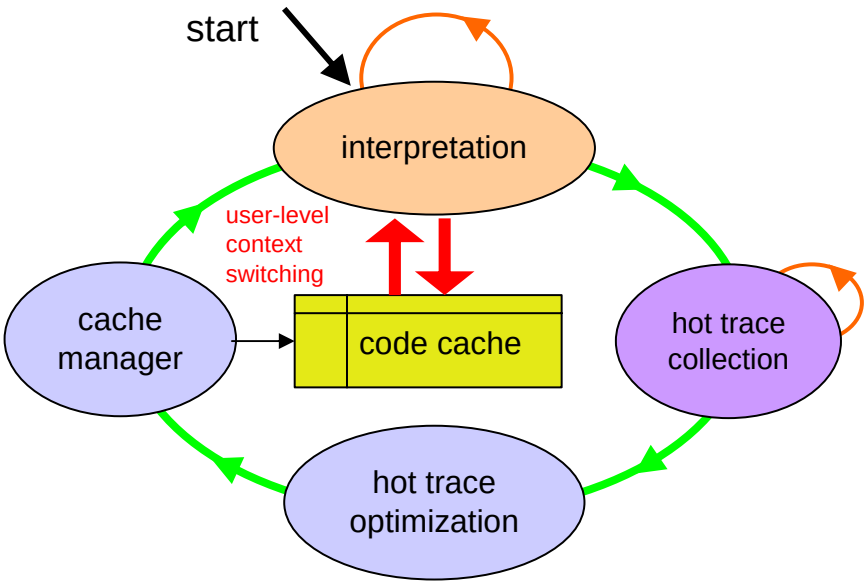

Figure II.1: Dynamo interpretation and optimization flow. Dynamo starts with interpretation of the program's binary. The program is profiled while being interpreted. When Dynamo detects a hot path, it switches to the collecting phase where the trace is formed. The trace is then being optimized and inserted into the code cache. So Dynamo interleaves interpretation/profiling, trace optimization, and native execution of optimized traces.

performed in the same thread as the main execution within a single hardware context. Sharing the same hardware context requires the system to pause the current program's execution in order to perform optimization. This also introduces additional runtime overhead due to heavyweight user-level context switching between execution, profiling, and optimization. So, these systems often adopt simple and optimistic strategies to reduce profiling and hot trace detection cycles. But these strategies can result in poor quality of hot traces. Similarly, these systems also perform less sophisticated optimizations to reduce optimization overhead.

The binary optimization framework proposed by Ootsu, et al. [101], focuses on detecting parallelizable loops in a single-threaded binary to speed up loop execution on multithreaded processors. The framework uses two phases of translation and optimization. Static translation and optimization (*STO*) analyzes the binary to identify control and data flow information, and instruments the binary to collect profiles at runtime. Dynamic translation and optimization (*DTO*) performs further optimizations partially done by STO. Trident, as described in this thesis, differs in that it performs all of its analysis on a code fragment in a parallel helper thread, so it requires no static analysis and instrumentation. In addition, rather than relying on statically identified events for optimization, Trident triggers optimization from dynamically identified hardware events.

The *ADORE* framework [85, 26] is the closest runtime optimization system to the Trident framework. ADORE is a two-thread model, as shown in Figure II.2 (a). It lets the application run in one thread, and uses a separate OS level thread to perform profiling and optimization (i.e., prefetching). ADORE takes advantage of Intel Itanium hardware counters [63] to collect a sequence of raw profile segments. The profiling thread periodically wakes up to mine through the raw profiles to detect any meaningful behavior patterns. These patterns identify the instruction PC's inside the program's hot traces. After

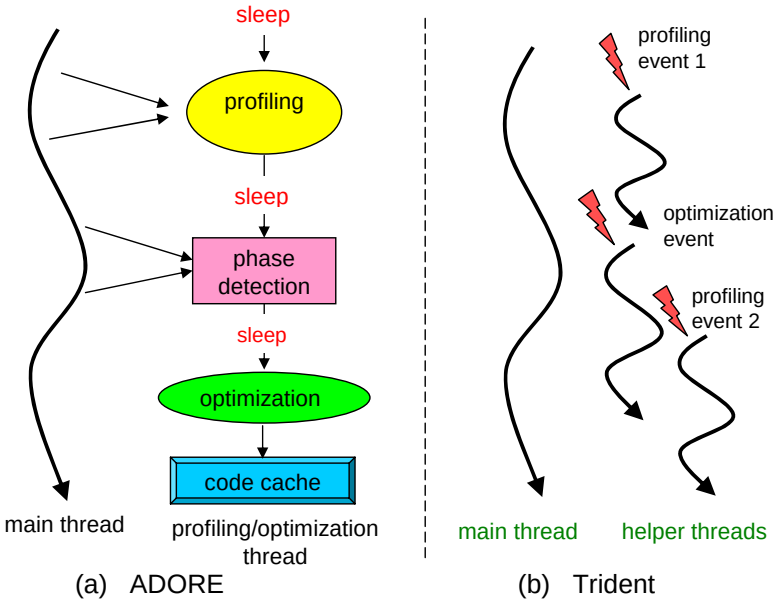

Figure II.2: ADORE and Trident threading models. ADORE interleaves profiling/analysis, phase detection, and optimization in a dedicated stand-alone thread. Trident allows multiple lightweight helper threads to be active concurrently. One helper thread is triggered to form a hot trace upon the profiling event, while the other thread can perform further optimizations on an optimized trace, which has been formed early.

being identified, hot traces are stored in a buffer and stay unused. The profiling thread begins to collect more profiles to detect the program's phase behavior. Only after a stable phase is detected, will the profiling thread begin to optimize hot traces. Otherwise, the thread will start over for both hot trace and phase detection. ADORE performs multitasking of profile collection, profile analysis, phase detection, and optimization inside a single thread. It is different from Dynamo in that ADORE performs these tasks in a separate thread and does not pause the main thread's execution.

Our Trident is built on the above ideas (from Dynamo and ADORE), but it dramatically reduces the profiling and optimization overhead. With the

removal of the overhead constraint, Trident is flexible enough to reconsider the optimization strategies, and has more freedom to re-optimize the code to take advantage of efficient hardware mechanisms available in new architectures. Trident makes these clear distinctions from previous work.

- The Trident framework focuses on reducing the overhead of profiling and optimization by proposing lightweight hardware support to perform all of the profiling needed to guide dynamic optimization. This avoids software profiling/analysis overhead and associated context switching overhead.
- The hardware interacts with the optimization framework by generating optimization events. These events are quickly handled by helper threads to make optimization decisions as well as to perform optimizations with very low overhead. Thus, Trident can react immediately to the program's behavior changes and adapt to shorter phases.
- Trident's event-driven nature makes it possible to process multiple optimization events via multiple helper threads simultaneously, as shown in Figure II.2 (b). All helper threads run concurrently with the main thread. Therefore, it allows continuous monitoring of more complex program behavior and allows continuous and recurrent optimizations. Trident naturally enables aggressive optimization by applying simple optimizations gradually, and provides the ability to recover from previous bad optimizations.

II.B.2 Translation Optimization Systems

Dynamic optimization is commonly seen in dynamic translation systems [41, 42, 10] through just-in-time compilation. Translation occurs from one ISA to another existing or proprietary ISA. These systems usually focus on compatibility or power efficiency issues. Optimization is often limited to the level of

the basic block. Binary translation is discussed more thoroughly by Altman, et al. [4].

In the Java Virtual Machine (JVM) [103, 15, 34], the JIT compiler interprets/compiles abstract Java byte-code and optimizes it to run on native machines. Optimization is usually tightly coupled with the virtual machine semantics. Runtime compilation overhead may be reduced with *Lazy Compilation* [77], where individual Java methods are compiled on demand upon their first invocation. The overhead may be further reduced using the *Background Compilation* technique, which uses an extra dedicated thread to perform just-in-time compilation on the background. The Java methods to be background compiled are prioritized according to static profiling.

The Jrpm system [27], with a similar motivation as [101], speculatively parallelizes a single-threaded Java program to run on a CMP processor with thread-level speculation (*TLS*) support [56]. Because CMPs have relative low sharing and communication cost, programs can be optimistically (via TLS) parallelized without violating correct sequential program order. Implemented as a Java virtual machine with dynamic compilation support, Jrpm is coupled with a hardware profiler to identify candidate loops to parallelize at runtime.

Our approach builds on all the research above, but uses helper threads to provide low overhead dynamic optimization of any binary running on the processor.

II.B.3 Selective Compilation

Selective compilation [53, 81] performs most optimizations at compile time, and only selects a portion of code to be optimized dynamically using runtime information. The code selection is usually identified by user annotations or source language extensions.

DyC [53] annotates the program's source code to identify static variables which have a single value or relatively few values, which are frequently used by many calculations. It automatically determines code regions, which depend on these variables, for dynamic optimization. These regions are partially evaluated during static compilation and are bound to the execution once values of those variables are known at runtime. DyC uses staged dynamic optimizations to minimize the dynamic compilation cost. This is done by performing much of the analysis and optimization during static compile time.

Selective compilation lacks runtime adaptability, and it also adds programming burden to programmers. In contrast, Trident works transparently on existing binaries.

II.B.4 Backend Support in Software Dynamic Optimization Systems

This section lists a few important issues related to software dynamic optimization systems: code cache management, multithreading supporting in the code cache, and exception handling.

- **Code Cache Management**: Because the code cache has a limited size, existing hot traces in the cache have to be frequently removed to make space for new traces. The code cache management policy should have low overhead, good temporal locality, and minimal fragmentation. Dynamo [7] resorts to global flushing to avoid complicated code cache management due to variable sized traces and trace chaining in the cache. Global flushing is triggered whenever Dynamo detects a burst of trace creation rate. Mojo [29] partitions the code cache as circular buffers (or sections), and flushes the oldest section when all sections are full. A trace is not allowed to straddle two sections. Simple policies used above incur high code cache miss rates, and are slow to adapt to the program's changing phases. More recent studies [59,

58] show that policies, such as grouping traces with similar lifetime and performing a medium-grained FIFO (*First-In First-Out*) eviction, result in an effective balance of cache management complexity and cache miss rates.

- **Multithreading Support**: There are two schemes to support multi-threading in the code cache. In general, the choice of these schemes is pretty much application domain specific. Systems like *DynamoRIO* [13] use a thread-private code cache. Private code caches can have redundant memory, which increases memory footprint and causes side effects on the instruction cache (*I-cache*). The biggest problem, however, is the additional overhead due to repetitious optimizations on the same instruction sequences. The shared code cache, as used in Mojo [29], avoids the redundancy problem, but introduces new challenges to the cache management, because it is difficult to determine if a given hot trace is in the middle of execution from any thread. Therefore, whenever cache flushing is needed, all threads must be forced out of the code cache. This is done by unchaining all hot traces to avoid self-loops in the code cache and preventing new threads from entering the cache. Similarly, thread synchronization is also required when adding a new trace to a non-full cache or when chaining a trace to other traces in the code cache. Relative to the cost incurred during cache flushing, the synchronization cost due to adding or chaining operations above is low, because the critical sections of these operations are small. More recent evaluation on the thread-shared code cache is in [14].

- **Precise Exceptions**: Another issue in the software optimization system is exception handling. Optimizations, such as dead code elimination, can cause problems for precise exception handling if an exception occurs while executing the optimized code. It is difficult, or sometimes impossible, to recreate the same signal context prior to the optimization [54]. Asynchronous signals

can be queued and handled after the currently optimized code finishes execution. For a synchronous signal, the software system attempts to construct the signal context by de-optimizing the code and re-executing the compensation code previously removed. Therefore, the software system needs to record the optimization logs, which may be simple because the optimizations are conducted on the straight-line traces.

Software systems [7, 29, 12] usually patch the signal handlers or call-back functions in the operating system with the pointer to a special routine so that the program's signal handlers can be invoked directly under the software system's control. However, Trident does not need to patch those handlers because Trident's optimization can be triggered directly via optimization events.

This thesis focuses on reducing profiling and optimization overhead, thus we will not study these issues in details. For example, we assume our code cache has unlimited size, though Trident has the ability to invalidate individual traces in the code cache. We also assume Trident uses a private code cache since we focus on improving the performance of single-threaded applications. Finally, we assume that precise exceptions are handled by microarchitectural support as in Crusoe processors [74, 41].

II.C Hardware Based Dynamic Optimization Systems

Hardware optimization systems typically store optimized traces in a hardware buffer, called the *Trace Cache* [105, 113, 98, 60], The trace cache and other hardware mechanisms conveniently reduce the overhead incurred in software based systems. In this section, we will discuss pure hardware based optimization systems. Hardware mechanisms to accelerate software optimization

systems will not be elaborated on here. These acceleration mechanisms include the hot spot detector[91, 92], fine-grain support for self-modified code [74, 41, 42], and hardware acceleration on control transfers in the code cache [72].

Hardware optimization systems usually perform lightweight optimizations such as constant propagation, register re-association, and *move* instruction elimination [48, 64]. Friendly, et al. [48] were among the first to perform trace optimizations via the trace cache fill unit. Their basic optimizations include constant propagation, register re-association, and scaled addition. They also propose marking the register move instruction as an explicit move so that it can be eliminated during register renaming without further execution. Jacobson and Smith [64] propose several specific optimization techniques based on the trace processor. One of these optimizations is to collapse a small chain of dependent instructions into a single operation using a new instruction not available to the external ISA. The instruction-path co-processor (*I-COP*) [32] is proposed as a programmable processor to enable the base optimizations above, but with much more flexibility. The *ROAR* architecture [100] greatly improves the effectiveness of dynamic optimization via hardware support of precise speculation.

The recent *rePlay* [104, 43, 127] and *PARROT* [3, 112] frameworks attempt to enable very aggressive hardware optimizations, by using a dynamically configurable optimization engine running in parallel with a high performance execution core. The key idea in these frameworks is the atomic execution of traces. Control dependencies are speculatively converted to form long, atomic traces upon which very aggressive code elimination can be applied. To construct a long atomic trace, rePlay exploits the *branch promotion* technique to convert highly biased branches into non-branch instructions (called *ASSERTs*), as shown in Figure II.3. With ASSERTs, multiple basic blocks can be packed into a trace while still maintaining its atomicity. During trace construction, a trace keeps growing

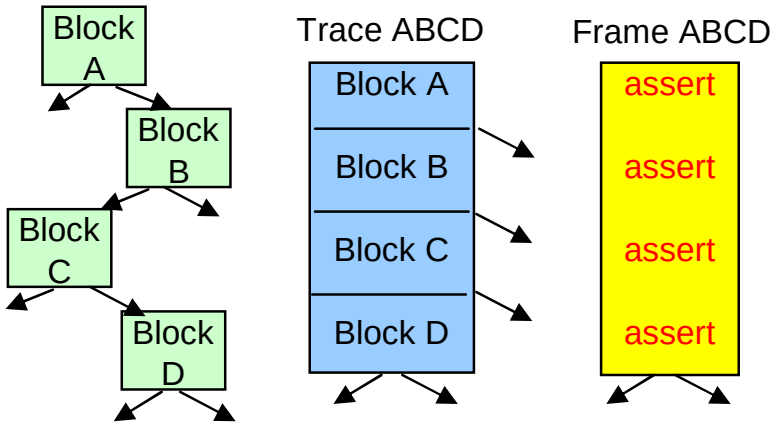

Figure II.3: The rePlay branch promotion. The biased, conditional branch is converted to an ASSERT. The trace keeps growing until the branch cannot be promoted. The final trace is an atomic, long sequence of blocks with ASSERTs. When a trace is executed, the ASSERT fires if the verification fails. Then it triggers a recovery process to re-start from the non-traced version of original instructions.

with promoted branches, and terminates on any non-promoted branches. Branch bias is tracked by a hardware branch bias table. PARROT [3, 112] has similarities to rePlay, but is more geared towards power efficiency. PARROT uses a two-level trace filtering scheme to find "hot" and "blazing" traces. The hotter a trace is, the more aggressive optimization it receives.

Since hardware optimization systems (e.g., rePlay and PARROT) store optimized traces in a dedicated hardware trace cache, this restricts how long the optimized traces are, or requires additional solutions to allow for longer traces. For example, variable-length traces may be truncated into fixed segments to store into different trace cache locations. These segments need to be chained together, but chaining introduces some complexity for instruction fetching and trace invalidation. Trident differs from these hardware optimization systems by storing optimized hot traces in the memory-based code cache, and the traces

can have arbitrary sizes. This avoids the difficulty of managing the traces in the hardware trace cache, and the optimized traces live across interrupts and context switches. In addition, Trident provides a more flexible, scalable optimization scheme than a hard-wired optimization engine, and enables user-level (i.e. user customized) code to perform the optimization in the program's address space.

More recent work on continuous hardware optimization, proposed by Fahs, et al [44], augments the renaming stage of the processor pipeline with a data-flow optimizer. The *in-pipeline* optimizer uses simple, table-based hardware to reduce the instruction data-flow height by performing basic optimizations, which include constant propagation, re-association, redundant load elimination, store forwarding, and silent store removal. The optimizer table maintains a *symbolic* representation of each architectural register value, which is converted to a *known value* after the execution result is fed back from latter pipeline stages. Thus the subsequent instructions using this register can be evaluated by the optimizer without being further executed in the latter pipeline stages.

This continuous optimization architecture has advantages of not requiring profiling of the instruction stream or caching the optimized traces. But it does impose problems of design complexity, power consumption, and the increased pipeline depth. For applications with a large instruction working set, but poor locality, it may need a large hardware table, which is impractical without severely impacting the microarchitectural timing. In contrast, Trident enables continuous yet flexible optimization by performing optimizations via software optimization in an available hardware thread. By storing the optimized traces in the memory buffer, benefits of optimization can last as long as needed without re-optimization.

II.D Program Profiling

Dynamic optimization relies on runtime profiling information to guide optimizations and to adapt to the program's changing behavior. Accurate and timely profiling is the key to enable efficient dynamic optimization.

II.D.1 Software Profiling

Traditional software profiling [76, 9, 19] is done statically by directly instrumenting the program. The metrics used to identify interesting behavior are typically based on event counting. Variational Path Profiling (*VPP*) [106] uses the execution time as a metric to identify paths with a large variation of execution time. VPP potentially identifies new optimization opportunities for paths, which are not heavily optimized under static profiling.

Software-based runtime profiling has also been well studied. Examples of these techniques are path profiling [8], targeted path profiling [66], and practical path profiling [11]. These software techniques often have high accuracy, but also have high runtime overhead.

Bursty tracing [61] samples sub-sequences (bursts) of the trace of runtime events to build a temporal program profile. Procedures are instrumented to collect profiles. This technique controls its overhead by switching back and forth between the instrumented code and the original code. Bursty tracing has relatively low runtime overhead, but it needs static instrumentation.

Due to runtime overhead, software dynamic optimization systems often fall back to inexpensive edge profiling [7, 6].

II.D.2 Hardware Profiling

Hardware based profiling schemes can range from the simple counter based approach [63, 40, 46], the hardware table based approach [91, 120, 99], to

profiling co-processors [147].

Counter based profiling takes advantage of the processor's performance counters for statistical sampling. Sampled raw profiles are accumulated in hardware buffers. When buffers are full, the software system is invoked to analyze profile samples, as in ADORE [84] and ProfileMe [40]. Shotgun profiling [46] uses sampled profiles to construct the dependence graph and identify the program's critical paths. Counter based profiling needs to collect many samples via a limited number of counters in order to detect any useful behavior patterns. It also needs software to analyze the profile samples. Thus, it often has high profiling overhead.

Table based profiling can collect profiles more quickly. The hot spot detector [91] uses a branch behavior buffer (*BBB*) to keep track of the execution history on a per-branch basis. The program's hot spot is detected when its working set converges to a set of hot branches currently in the BBB. Stratified Sampling [120] splits the input streams into multiple substreams, which are individually sampled by a random sampler. Because each substream is biased, stratified sampling may expect fast convergence with high accuracy. Multihash profiling [99] divides the program's execution into fixed intervals. It uses multiple hash tables to filter out profiling events in a given interval, and classifies events according to their importance relative to all other events. Multihash profiling can catch various profiling events (e.g. edge profiles and value profiles) with high accuracy.

Co-processor based profiling [147] uses a programmable co-processor to collect and analyze profile samples generated by a microprocessor. The co-processor is flexible enough to detect a broad range of information, such as correlations between instructions (e.g. memory dependence profiling) and different dynamic instances of the same instruction (e.g. value profiling). Information

stored in the co-processor buffer is transferred to the main processor by interrupt or explicit read from the main processor.

The hardware profiling schemes above often impose great hardware complexity. In this thesis, we extend the software path profiler in Dynamo [7] for fast and efficient hardware implementation. We augment this profiler with a few hardware bitmaps (e.g. three 16-bit bitmaps) to collect multiple execution paths after a hot branch. The final path is chosen by a voting scheme to get the longest common subsequence among bitmaps. This scheme improves the trace quality over Dynamo's optimistic selection, without introducing significant hardware complexity. Trident also proposes additional hardware to support specific optimizations.

II.E Helper Threading

The Trident dynamic optimization system exploits lightweight helper threads to perform runtime optimization. This is one of several mechanisms that allow a parallel machine to use idle thread execution resources to make a single-threaded application run faster, without actually parallelizing the application. This section summarizes related research on helper threading.

Helper threading is enabled by modern processors' on-chip parallelism, such as simultaneous multithreading (SMT) [133] or chip multiprocessing (CMP) [56]. The primary goal of the helper threading technology is to predict or speculate the program's behavior to speedup a single-threaded application. The helper thread code can be generated dynamically (e.g., [36]) or statically at the high level of source code [87, 71].

Simultaneous subordinate microthreading (SSMT) [23] uses microthreads to do prefetching, branch prediction, or even hardware resource management. Dynamic speculative precomputation (DSP) [36] targets long latency cache misses.

Helper threads are constructed dynamically by hardware, and are stored in the hardware buffer. DSP exploits the chaining trigger, which allows a helper thread to spawn a new helper thread, to prefetch cache-missing loads further down the instruction stream. Speculative data-driven multithreading (DDMT) [116] exploits helper threads 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) [146] is similar to DDMT except that (1) helper threads in EBP may loop multiple times, instead of one-time execution in DDMT. (2) EBP triggers helper threads at the fetch stage, slightly earlier than DDMT at the renaming stage. The slipstreaming processors [107] use the A-stream (speculative) for branch and value prediction, and passes all results to the R-stream (main) via a hardware FIFO.

Helper threading also serves other purposes, such as speculative code parallelization, or even software security checking. Examples of speculative code parallelization are the thread-level speculation (TLS) [130] and the master-slave speculative parallelization (MSSP) [148]. Jrpm [27] is a TLS-based system using Java virtual machine. A recent study, called *HeapMon* [126], extends helper threads to detect and pinpoint memory related bugs. In HeapMon, each heap location is associated with a state bit. The helper thread checks if the heap word is in an illegal state and logs the status. The helper thread receives the memory access address from the main thread via a hardware mechanism.

In this thesis, we use helper threads to dynamically optimize the main thread's code on the fly. We also dynamically generate precomputation code to prefetch data on behalf of the main thread.

III

Event-Driven Multithreaded Architecture

This thesis presents an event-driven multithreaded dynamic optimization framework, called *Trident*. The motivation behind Trident framework is to enable continuous adaptive optimization with low overhead and but high flexibility. Trident exploits three means to achieve these goals:

- **Simple hardware support for runtime profiling**. Trident conservatively extends the processor's performance monitoring mechanism to identify performance-critical events. These hardware structures monitor the program's execution behavior, and generate events to trigger optimizations. However, our proposed structures and the mechanism to trigger helper threads on events are mostly general-purpose – we anticipate future systems with a wider set of hardware-supported optimizations than are evaluated in this thesis.

- **Low overhead helper thread to achieve quick and fast response**. Trident takes advantage of a hardware multithreading processor, using an otherwise idle hardware thread to concurrently optimize the main thread

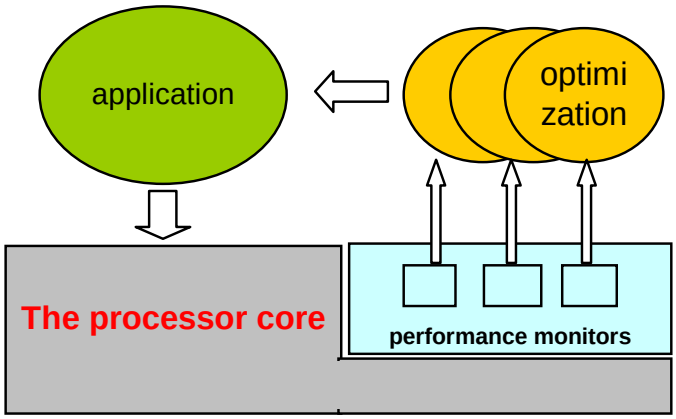

Figure III.1: Trident dynamic optimization architecture. Circles represent hardware threads. Hardware monitors the program's behavior, such as the number of branches executed or the number of data cache misses. The monitoring structures are off the processor's critical execution path. When the monitor detects an optimization event, the corresponding optimization thread (event handler) is triggered to run, in parallel with other threads.

that is running.

- **Concurrent optimization without interrupting the main program**. Hardware event-driven monitoring and concurrent optimization provide an extremely low-overhead dynamic optimization system that can support continuous recurrent optimization. Trident allows very aggressive or adaptive optimization by gradually performing simple optimizations.

In this chapter we give an overview of Trident's optimization architecture, and describe the performance monitoring extension to support the Trident framework.

III.A Overview of Trident Architecture

The Trident optimization architecture is shown in Figure III.1. Here, circles represent the hardware contexts (threads). The performance counters monitor the application's execution behavior, such as the number of branches executed or the number of data cache misses. The application runs in a thread while optimizations are performed concurrently in other hardware threads.

To achieve high flexibility without introducing too much hardware complexity, Trident is implemented as a software/hardware hybrid, as shown in Figure III.2. The architecture has two components, interacting with each other. The software component includes runtime support, a dynamic optimizer, a code cache, and associated software structures. The hardware component includes an event registration structure, the thread triggering mechanism, hardware event monitoring structures, and a hardware event queue. Note that most of these hardware structures already exist (or partially exist) in modern processors. We augment the hardware monitors to profile hot backward branches and watch over optimized hot traces. We also add a small registration structure. Details of these hardware and software structures are explained next.

Trident is a trace based optimization system. The foundation of our optimization is to identify the program's frequent execution paths to build hot traces. Then as we learn more about the hot traces we can re-optimize them for further performance gains. We will first examine how the Trident architecture builds and optimizes hot paths.

III.A.1 Definitions

First, we provide a few definitions.

- **Hot Traces**: A hot trace consists of a number of basic instruction blocks frequently running together. These blocks are often non-contiguous in the

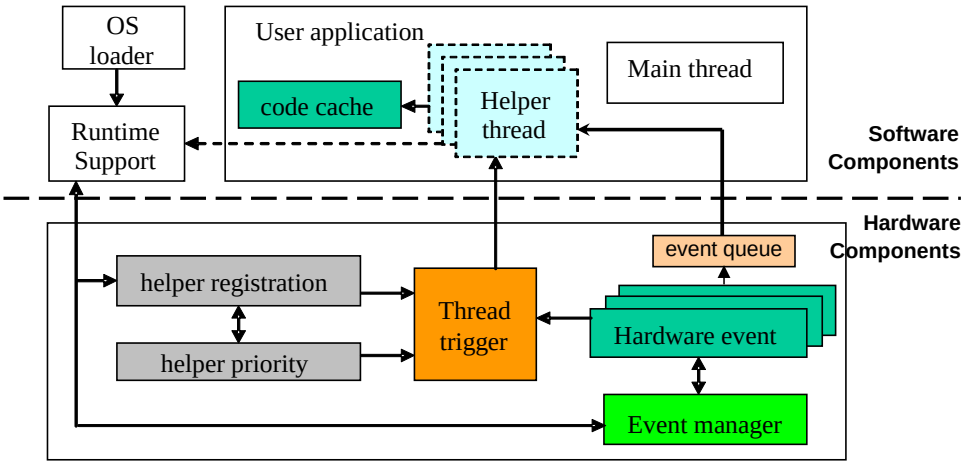

Figure III.2: An implementation of Trident architecture. Trident is a software/hardware hybrid. The software component is responsible for registering an optimization event, performing optimization, and managing optimized hot traces. The hardware component is responsible for monitoring the program's behavior and triggering optimization threads.

original binary layout. In Figure III.3 (a), blocks *A, B, D, H, J, K, G* are frequently executed together. So these blocks form a hot trace.

- **Trace Formation**: The goal of trace formation is to identify the basic blocks above and streamline them to get better execution locality. An example of the streamlined trace is shown in Figure III.3 (b). A hot trace usually has single entry and multiple exits.

- **The Code Cache**: It is a separate memory buffer to store hot traces. It is managed and maintained by the runtime optimizer.

- **Hot Events**: The occurrence of a particular type of runtime behavior. For example, a frequently executed branch may trigger a *Hot Branch* event when its execution count exceeds a predefined threshold.

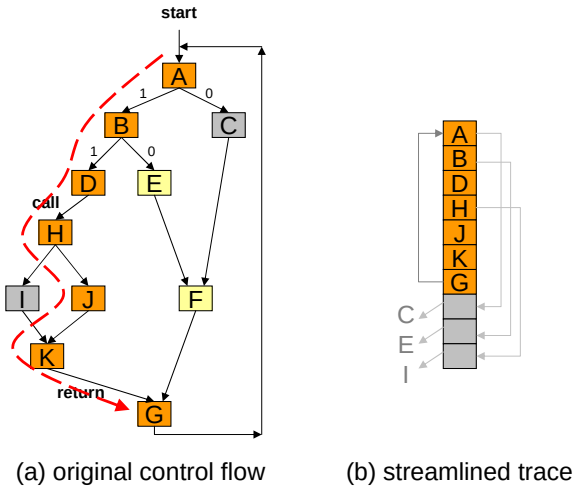

Figure III.3: An example of the hot trace. The dashed line indicates the frequent execution path in the program's control flow graph. Trace formation will form the streamlined trace in the right side of the figure.

III.A.2 Runtime Support

Trident's runtime support is needed for hot event registration and initialization. To apply the Trident optimization system, the OS loader calls runtime routines to specify that a given main thread is to be monitored. The monitor structures are programmed with the given thread ID to initiate monitoring. In our current simulation infrastructure, the hardware monitoring structures are assigned to one running thread at a time.

At the same time of event registration, runtime routines are also called to create a generic helper thread context. The runtime support allocates in the program's address space the dynamic optimizing compiler code, a stack, and some global data space. This is similar to loading a shared library into a program. This thread does not run unless it is triggered by future optimization events.

When registering an optimization helper thread, the runtime system creates a helper thread *Registration Structure* in the program's address space.

The registration structure contains a pointer to the starting code of the helper thread, as well as the stack pointer, global data pointer, a pointer to its code cache structure, and thread priority. The code cache structure keeps track of the free and allocated space for the code cache, since all of our optimizations interact with the code cache. The priority affects the helper thread's instruction fetch throughput, to control its impact on other threads. For example, Trident is simulated on a SMT processor. We use the ICOUNT policy [133] to fetch instructions from each thread. By giving the helper thread a low priority, instructions from the helper thread get fetched and executed primarily when the main thread is stalled due to long latency events, such as load misses in the data cache. Thus, Trident uses the priority scheme to reduce negative impact on the main thread.

III.A.3 The Dynamic Optimizer

The runtime optimizer is a set of functions which perform actual optimizations. Each function (or *event handler*) corresponds to a specific hot optimization event. The optimizer also includes supplementary functions to interact with the underlying machine (simulator). Examples of these supplementary functions are accessing the monitor structures and decoding instructions.

Upon occurrence of an optimization event, the corresponding event handler is dispatched to run as a helper thread. Event handlers are independent of each other, thus can run concurrently. Trident's basic event handlers include *Trace Construction* (with base optimizations) and *Trace Invalidation*. Other optimization specific handlers include *Speculative Value Specialization*, *Adaptive Dynamic Inlined Prefetching*, and *Precomputation Code Construction and Optimization*.

Table III.1: Trident hardware monitors and events. The first two structures will trigger optimization events. I-cache counters are used to place hot traces into the code cache. These counters are already available in modern processors.

Event	Source	Actions
Hot Branch	Branch profiler	helper thread spawned to construct a hot trace
Code Cache Invalidation	Watch table	invalidate the hot trace corresponding to the virtual address in the watch table
	I-cache access counters	Usage: to help allocate space in the trace cache to reduce I-cache conflict

III.A.4 Hardware Monitors and Events

Trident exploits some generic hardware monitoring structures to detect and construct hot traces, on which more optimizations can be further performed. Trident is flexible to extend its monitoring structures to enable specific optimizations. These specific monitoring structures will be discussed separately.

Trident's hardware monitors and associated events are listed in Table III.1. Some of these structures can trigger events, and others are just used by Trident during optimization.

- **Hot Branch Profiler**: The branch profiler identifies the frequently executed backward branches and generates *Hot Branch* events. The profiler includes two components: a set-associative cache used to identify hot branches, and B global history bitmaps used to find the dominant path for the hot branch. This structure is based on Dynamo's MRET structure [7] for ef-

ficient hardware implementation, but we improve it by adding bitmaps for better trace quality.

Each cache entry has a small counter to indicate how many times a branch has been executed. When a taken branch is committed, its PC is used to index into the cache and the counter in the cache entry is incremented. The least occurring branch is replaced when the cache is full. When the counter exceeds a predefined threshold T, a *hot* branch is detected.

Once a hot branch has been identified, we then start to keep track of the global history paths that occur after that branch. We do this by keeping track of the next B different global history bitmaps of length L that occur during execution for each hot branch. In our study, we set L to be 16 branches. This defines the maximum size of a hot trace. A 0 for not-taken and 1 for taken is stored into the global history bitmap for the 16 branches that occur after the hot branch. Once we have B executed paths for a hot branch in this bit history form, we then vote to identify the dominant path among these. The dominant path is the longest common subsequence across the different global history bitmaps. This is chosen by starting at the 1st branch after the hot branch, and voting across the different global history branch positions to see if the trace should follow the taken or not-taken path. This is done by a majority vote across the different histories. During this voting, as soon as a given path history disagrees with the majority it is no longer eligible to vote. In addition, once a majority can no longer be established, we stop expanding the hot path.

To illustrate how the voting scheme works, we assume the branch profiler keeps track of 3 bitmaps and a hot branch has following 3 global history paths of length 8 as shown in Table III.2. When picking the dominant path, the first two branch directions 10 are in agreement among the 3 sampled paths.

Table III.2: Trident hot trace voting scheme. The profiler collects multiple history path bitmaps after a hot branch. Voting starts off from the first bit in all bitmaps to choose the majority as the winner, which represents the *hot* outcome/direction of that branch. Then it moves to the second bit for the next round of voting, and so on. The losing bitmap will be dropped from next round of voting. The final result represents the longest common subsequence among all bitmaps.

hot branch	3 history path bitmaps; each bitmap length of 8
	1 0 0 1 0 1 0 0
	1 0 1 1 0 1 0 0
	1 0 1 1 0 0 0 0

The 3rd branch history has a majority vote of taken (1). At this point we have a dominant path of 101 and we only consider 10110100 and 10110000 for voting on the next branch, since 10010100 disagreed with the majority vote for the 3rd branch. In continuing down the path, these two histories differ at the 6th branch, and at this point a majority cannot be reached, so the hot branch is queued up as an event with the path of **10110**. This bitmap, along with the branch starting PC, will then be used by the helper thread to create optimized hot traces.

- **I-Cache Access Counters**: These counters are used when we place a new hot trace into the code cache. Trident polls I-Cache access counters that estimate which I-cache blocks are frequently accessed. This is used to make decisions as to where to place optimized traces in the code cache in order to reduce cache misses among the hot traces and between hot traces and the original code. Similar counters are already available on modern processors like the Intel Itanium [63].

- **The Watch Table**: This hardware table monitors the performance of optimized hot traces. The goal is to identify when an optimized trace is frequently deviating from the path for which it was optimized. Each table entry stores the hot trace's starting virtual address in the code cache and a completion threshold. The threshold specifies the number of *sequential* instructions that need to be used from the trace in order for the trace to be beneficial. The value of the threshold is different for every trace, since the traces can be of different lengths. Therefore, the threshold is set to be a percentage (e.g., 60%) of the optimized trace length, which is passed in when initializing the table entry. The watch table knows when a trace is prematurely exited if it sees a taken branch, which is a branch out of the trace, commit before the completion threshold is reached. Each table entry also contains an invalidation counter to identify when the trace should be invalidated. The counter starts out at 0, and each time the trace is exited before the completion threshold is met, the invalidation counter is incremented. Each time it executes that many instructions or more it is decremented. If the invalidation counter reaches a threshold, then a hardware trace invalidation event is inserted into the event queue.

  As noted in the following paragraph, on a context switch, all of the hardware structures are flushed, so we need a way to specify which optimized traces are to be monitored when the thread resumes. This is accomplished through a special instruction that is inserted at the start of each optimized trace. The instruction says to insert the current PC and the trace length into the Trace Watch Table, if it is not already there. We found that this instruction does not impact performance, since there are no dependencies on it. We expect only a small number of entries in the watch table because a typical application has a relatively small working set. The table may be replaced

via a simple FIFO policy.

On a context switch of the main thread, we do not save the current hardware profiling nor the optimization state of the helper thread. Instead, the hardware event queue and structures are flushed, and the helper thread's execution is stopped. When the main thread resumes execution, the hardware monitoring of events will start again, and the event queue will be populated, which will in turn trigger the execution of the optimizing helper thread on a new event. This is possible because the only thing we need to start executing the helper thread is a pointer to the registration structure. The thread registration structure provides a fast mechanism for spawning a thread to handle the associated event, and an efficient mechanism to keep track of state across context switches and helper thread invocations. The only state that remains from one run to the next of the helper thread is the code cache state.

III.B Trident Optimization Flow

In this section we describe the basic optimization flow in Trident. Optimization is driven by the hot events which are listed in Table III.1.

- **Program Monitoring**: After event registration is done and hardware monitoring structures are initialized, hardware begins monitoring the program as it executes. When hardware detects an event, the profile data associated with the event is put into the hardware event queue to be consumed by an optimizing helper thread. When the event queue fills up, new events overwrite older events. In the basic optimization, the hardware event is to find the starting PC and the subsequent hot path. The hot path is a series of bits indicating conditional branch outcomes. If the helper thread assigned to this queue is not running, the hardware signals the runtime sys-

tem, which consults the registration structure to identify the helper thread parameters corresponding to the hot event. This initiates the helper thread to run in a spare hardware context having access to the application's virtual address space for optimization.

If there are no hardware contexts available, the event will stay in the queue, or be overwritten by newer events later before being handled. In this thesis, we always assume we have one thread (in addition to the main thread) available to handle optimization events. The event queue is assumed to hold one event.

- **Trace Formation**: When a helper thread processes an event from the hardware queue it first determines what type of event it has, in order to invoke the correct routines. When a hot branch event is consumed, the helper thread will start running the *Trace Construction* code to build an optimized sequential trace for the hot path found.

  The hot path trace generated terminates at the beginning of an existing hot trace, a self loop, or any indirect jumps. However, a *return* instruction does not terminate the trace when its matching *call* instruction is within the trace. The helper thread performs basic optimizations on the trace as explained in the next chapter.

- **Linking Trace**: The helper thread inserts the new trace into the code cache and updates the code cache data structure. Finally, it links the new trace into the main thread's execution by patching the original binary with a jump instruction to the new trace. The location of patching is where the new trace starts. After patching, subsequent fetches to that hot path will execute from the code cache.

  When a helper thread is done processing an event, it checks to see if there is

an event in the queue and if so, it processes it. If the event queue is empty, then it stops running. Later, when a new event arises the event manager will enable the helper thread to process the new events.

- **Continuous Monitoring and Trace Invalidation**: Hardware monitoring continues. The hardware watch table also monitors hot traces executed in the code cache. The watch table will generate an invalidation event if a hot trace's completion degree is below the invalidation threshold.

  When the helper thread handles the code cache invalidation event, it will undo the patching by replacing the jump instruction with its original instruction. This is made possible since anytime we insert a trace, the instruction which is replaced by the jump and other information (i.e. the optimized trace's original source starting address, virtual address in the code cache, and length) are stored in the bookkeeping directory of the code cache. This allows us to invalidate a trace (i.e. backing up from a bad optimization) or to replace a trace with a more optimized version.

For the example in Figure III.3, the Trident optimization flow is shown in Figure III.4. In this example, we assume the hot branch threshold in the branch profiler is 8 and the profiler can hold three path history bitmaps. After branch A is executed 8 times, it becomes *hot*. The branch profiler begins to track the path history after A. The bitmaps collected are 1110111, 10111, and 1110111. Using the voting scheme described early, the hot path is 1110111. The profile data associated with this event is <*PC, 1110111*>.

The hot branch event triggers the helper thread to run in an idle hardware thread, in parallel with the main thread. After the trace is formed and optimized, it is inserted into the code cache.

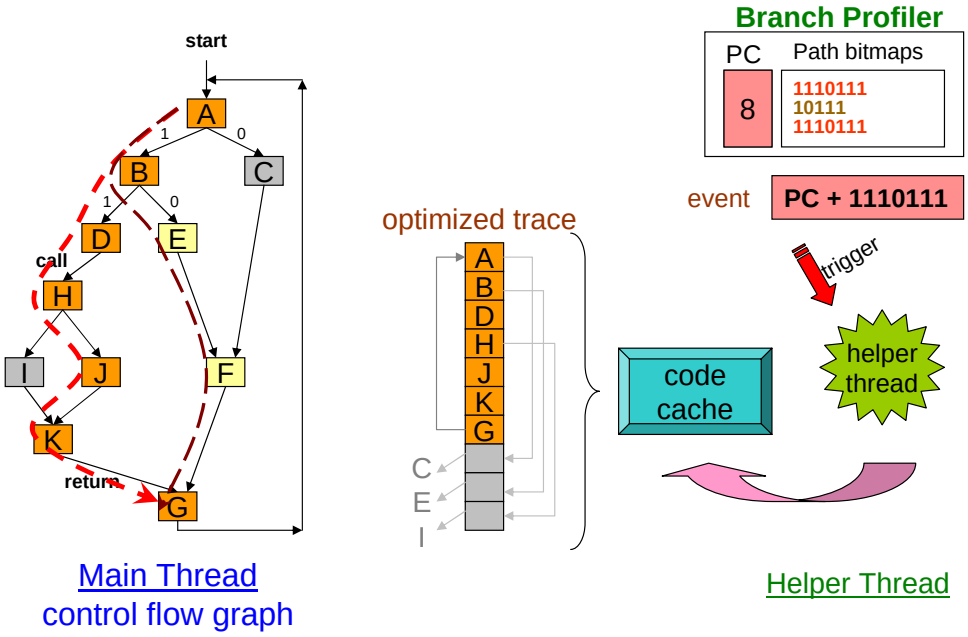

Figure III.4: Trident dynamic optimization flow. The path A-B-D-H-J-K-G is the frequently executed path (*hot*). The path A-B-E-F-G is a less frequent path (*warm*). Both paths end at the block G since it has a loop-back branch, which terminates a path during branch profiling. The path history bitmaps for these two paths are 1110111 and 10111, respectively. The voting scheme picks the winner path 1110111 inside the branch profiler. This generates the hot branch event: <PC, 1110111>. Then the Trace Construction thread is triggered to run. The hot trace is created, optimized, and inserted into the code cache.

IV

Trace Formation Based Optimization

Trident is designed to quickly respond to hardware events and perform dynamic optimizations with very low overhead. It is a trace based optimization system. Hot trace formation provides a fundamental vehicle to enable other optimizations. The performance of trace formation based optimization depends on design strategies in three major areas:

- Hot Trace Selection
- Trace Optimizations
- Code Cache Management

These strategies have been studied by many research groups [7, 29, 13, 85, 58]. Because of high overhead in these optimization systems, the design strategies are often chosen to be quick and optimistic (or simple). In this chapter, we re-examine some of these design strategies due to the removal of overhead constraints in Trident. At the same time, we also discuss how Trident addresses two important issues in dynamic optimization, which have not been studied in

current software systems. These two issues are I-cache usage based code cache placement, and branch (i.e. *return address*) misprediction reduction.

IV.A Trace Formation

When Trident detects a hot branch event, the *Trace Construction* helper thread is triggered to form a new trace. The branch event, which includes the trace starting PC and a dominant branch history bitmap, is passed to the helper thread through the event queue as described in Section III.A.4. The trace includes all of the basic blocks along the hot path found. The trace our optimizer creates is only terminated earlier than this if an indirect jump is encountered.

The base optimization performed by most dynamic optimization systems is to create optimized hot traces, which by itself helps improve fetch throughput and branch prediction accuracy. The base optimization includes streamlining the trace and performing classical compiler optimizations.

- **Streamlining the trace**: During the hot trace construction, conditional branches in the trace are adjusted to match their target addresses within the trace. All unconditional branches are removed, but their effects are preserved if needed. For example, if a pair of call/return instructions (i.e. the call instruction and its matched return instruction both on the trace) are removed. However, if there are any intermediate *conditional* branches within the paired instructions, then the return address is moved into the calling register. This allows the trace to branch out before the return is reached. Similarly, this is also done for any unmatched call instructions on the trace.

- **Classical optimizations**: On top of trace formation we perform classical compiler optimizations on the trace. These include

- Constant propagation
- Copy propagation (or register re-association)
- Redundant and dead instruction removal

Redundant loads are removed if there are no intermediate store instructions between them in the trace. A *move* (or *copy*) instruction is removed if it has been copy propagated and its destination register is redefined in the same basic block. This allows the hot trace to branch out at the end of any basic block. Trident does not perform register re-allocation during basic optimizations, so we scavenge free registers to be used only within the basic block in which they are redefined.

IV.B Methodology

Trident is simulated on a 20-stages simultaneous multithreading processor [133] with 2 hardware contexts. The baseline configuration of the SMT processor is shown in Table IV.1.

IV.B.1 The Configuration of Hardware Monitors

The Trident framework includes two small hardware structures which monitor the program's execution: the hot branch profiler and the trace watch table. The first structure will generate hot branch events to trigger the trace formation and optimization. The second structure will generate trace invalidation events to remove under-performing traces in the code cache. The configurations of these two hardware structures are shown in Table IV.2.

Table IV.1: The baseline SMT processor configuration. The SMT processor has two hardware contexts to run programs simultaneously.

Pipeline	20-stage, 256-entry ROB, 224 registers
	Two hardware thread contexts
Queue Sizes	64 entries each IQ, FQ, and MQ
Fetch Bandwidth	8 total instructions
Issue Bandwidth	8 instructions per cycle
	up to 6 Integer, 3 FP, 4 load/store
Branch Predictor	2bcgskew, 64K entry Meta and gshare
	16K entry bimodal table
ICache size & latency	64 KB 2-way associative, 2 cycles
D-Cache size & latency	64 KB 2-way associative, 2 cycles
L2 size & latency	512 KB 8-way associative, 20 cycles
L3 size & latency	4 MB 16-way associative, 50 cycles
Memory Latency	600 cycles

Table IV.2: Trident baseline monitor configurations. These hardware structures are needed to form basic hot traces.

Hot branch profiler	256-entry 4-way associative and each entry has a 4-bit counter. Three standalone 16-bit bitmaps
Watch table	128-entry; each entry has a trace tag, an invalidation threshold, and an invalidation counter.

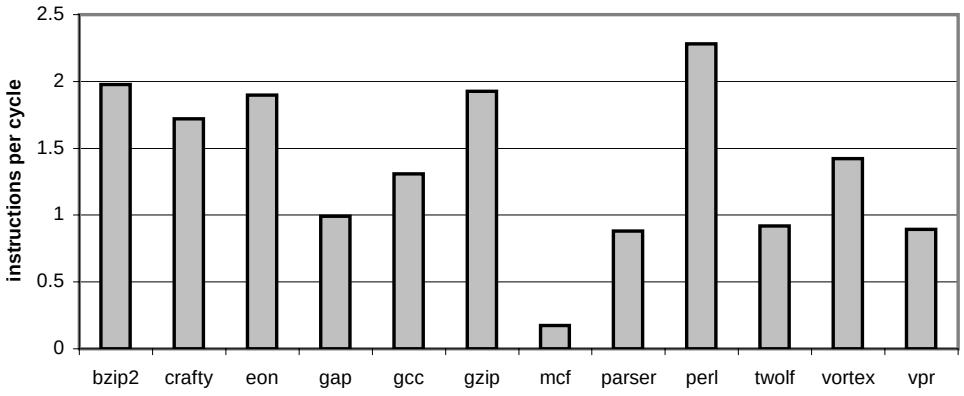

Figure IV.1: Performance of SPEC 2000int on the baseline SMT processor. Each benchmark runs alone in one hardware thread while other hardware threads are idle. The simulation is warmed up for 5 millions instructions, and then a total of 100 million instructions are simulated.

IV.B.2 Benchmarks

Trident is evaluated with SPEC 2000int benchmarks with reference inputs. All benchmarks are compiled on the Alpha platform (Digital Unix V4.0F) with the highest optimization options. Each benchmark is simulated for 100 million instructions beyond the single simulation points from SimPoint [122]. The simulator is warmed up with 5 million instructions before the true simulation starts. Dynamic optimization and related structures are not enabled until after warmup is finished. A total of 100 million instructions are simulated to demonstrate Trident's ability to quickly capture and then benefit from concurrent optimization. We expect even better performance improvement when simulating more instructions because the dynamic compilation cost and ramp-up time will be amortized, since we start simulation with no hot traces. Figure IV.1 shows the base performance (instructions per cycle or *IPC*) of each benchmark when executed alone on the baseline SMT processor. The base performance is used for performance comparison in this chapter.

IV.B.3 Simulation Assumptions

Trident exploits helper threads to perform dynamic optimization on hot traces. The runtime optimizer code executed by helper threads is written in *C* and compiled with *gcc -O5*. Special care is taken to make the runtime code thread safe. When a helper thread is triggered to run, we simulate all but the startup of the thread in detail on our SMT simulator. We therefore add a 2000 cycle latency when starting a helper thread. This consists of executing our runtime system to initialize the helper thread's registration structure for the optimization to be performed, which sets the PC, stack pointer, global data pointer, and sets the thread's priority.

IV.C Evaluation of Trace Formation

In this section, we evaluate some design options in Trident.

IV.C.1 Candidate Hot Path Starting Points and Trace Linking

Trident uses the hot path hardware profiler described in Section III.A.4 to find the potential starting points for the traces and the path to be optimized. As described there, the hot branches are first identified when a branch occurs T times while in the hot path profiler. After a hot branch is identified, the path profiler keeps track of B paths that occur after a candidate hot path starting point. It then uses a voting scheme to determine the longest dominant path. In Figure IV.2 we show the performance speedup results for different values of B and T when applying full optimization (including value specialization). The scheme is represented as a pair, $<B.T>$, where B stands for the number of path instances tracked, and T for the hot branch threshold. We simulate various combinations of B and T, and the scheme <3.08> performed the best. Voting

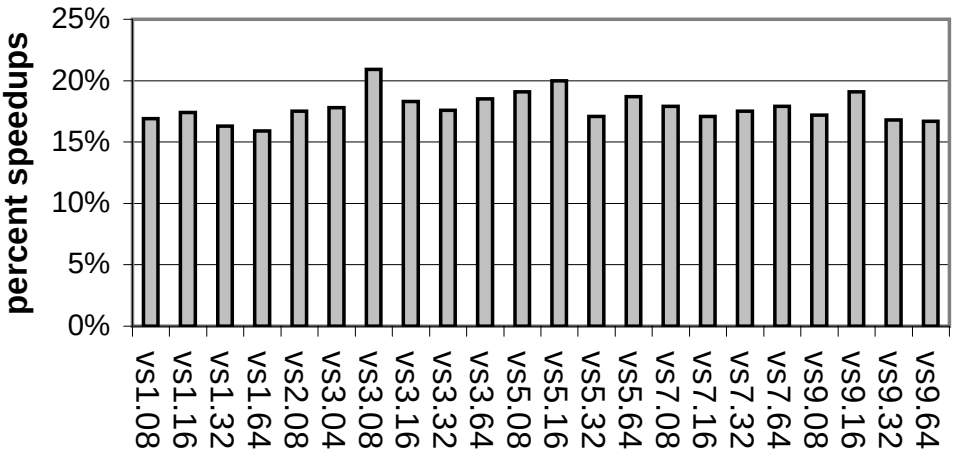

Figure IV.2: Comparison of hot trace selection schemes. The selection scheme is represented with a pair of numbers. The first number indicates the number of path history bitmaps used for voting. The second number indicates how many times a branch has to be profiled in order to be considered as *hot*. For example, <3.08> stands for 3 history bitmaps being collected for voting after the branch is encountered 8 times.

among three instances of paths boosts the possibility of detecting the dominant hot trace. Consequently, we can lower the hot branch threshold to 8. The scheme of <3.08> is used in all subsequent evaluations.

Most software dynamic optimization systems allow exit branches from a hot trace to form new hot traces. This will not happen in our system because only branches in the original code are profiled by the hot path profiler. This is enforced by filtering the branches through the Trace Watch Table before indexing them into the hot path profiler. As described in Section III.A.4, the trace watch table keeps track of all of the currently executing traces from the code cache, so the hardware knows if the branch being committed is from the code cache or not.

Because we do not let branches in the code cache become candidates for starting a trace, it might be worthwhile to apply *Trace Linking* among optimized

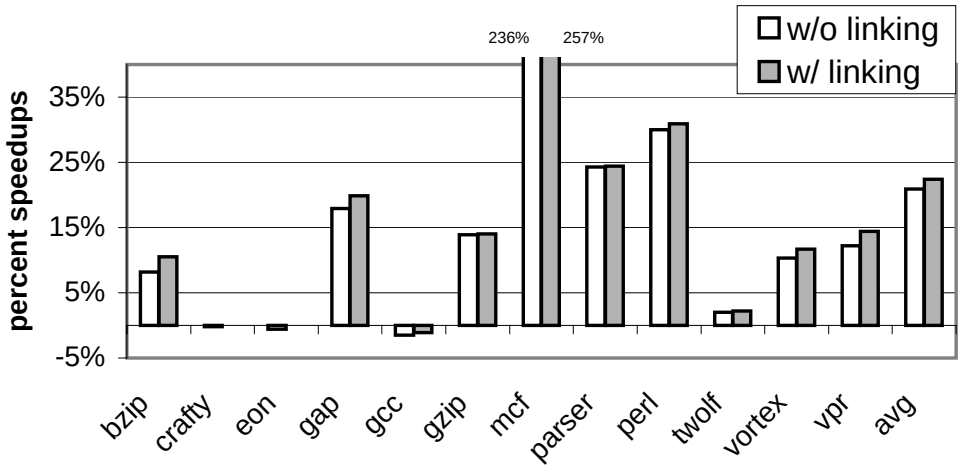

Figure IV.3: Performance with and without linking. Trace linking chains optimized traces together inside the code cache. If one trace has an exit branch whose target is the beginning address of another optimized trace, then the branch can directly jump to its target trace without jumping back to the original binary.

traces in the code cache. Trace linking is a common technique in current dynamic optimization systems to directly jump from one hot trace to another in the code cache without going back to the original code. This is done by patching the target address of the exit branch in the hot trace with the beginning address of next trace. Figure IV.3 shows the performance benefit due to hot trace linking. Unlike most existing software dynamic optimization systems where linking could impact performance by as much as *40X* [7], we only observe 1.5% performance slowdown without trace linking. This is because moving between traces without linking only incurs a couple of extra jumps. With good branch prediction accuracy, the corresponding penalty is small. In other systems, switching from hot traces to original execution typically incurs an expensive, user-level context switch.

IV.C.2 Hot Trace Invalidation

Trident exploits the watch table described in Section III.A.4 to generate *Code Cache Invalidation* events that trigger a helper thread to remove the under-performing traces from the code cache. Trident's invalidation mechanism is fine-grained, and adapts to the program's changing phase behavior quickly.

When a trace is formed, a watch table insertion instruction is put at the front of the trace, which inserts the starting PC into the table along with the trace length. The watch table then monitors the amount of the trace used, to see if it is above or below a *completion threshold*. This is used to maintain an invalidation counter to determine when the trace should be invalidated. If the amount of the trace used is below the completion threshold (e.g., 60%) enough times, then the corresponding trace is a candidate for invalidation, because not enough of it is being used and there potentially are better or more dominate paths that can be represented. Invalidation involves re-patching the original code with the original instruction (stored in the code cache directory) and flushing the corresponding I-Cache blocks.

Figure IV.4 shows the performance impact using different trace completion thresholds on the x-axis. The first bar shows the result when hot traces are never invalidated, and the rest of the bars show when a completion threshold of at least 20% to 90% of execution is needed in order for the trace to not be invalidated. Our simulation shows that the dynamic optimization performance is fairly insensitive to the invalidation threshold until it is really aggressive (e.g., requiring 90% trace completion threshold). At that point, overhead increases due to frequent trace removal and regeneration.

As opposed to the previous studies, code cache management is largely ignored. Code cache management policies have been done by Hazelwood, et al [58] with different code cache granularity. Due to trace linking, individual invalidation

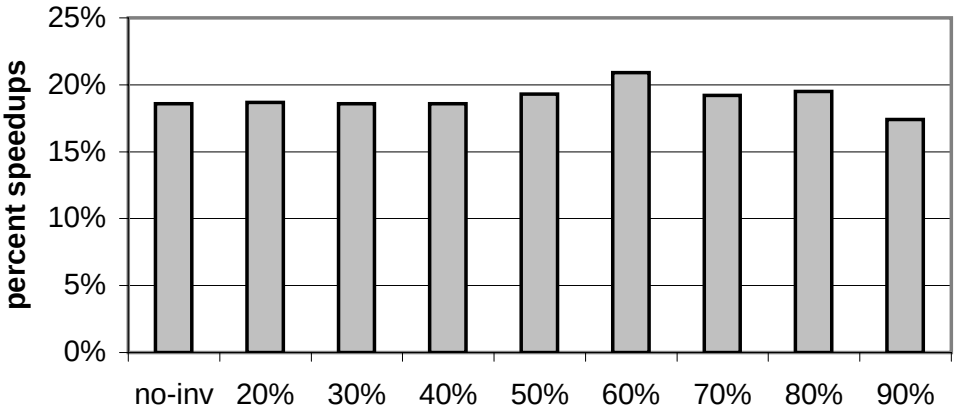

Figure IV.4: Code cache invalidation with different thresholds. A trace is invalidated if its average completion degree is below the specified threshold. The completion degree is calculated as the number of executed instructions divided by the total number of instructions in the trace. Here, **no-inv** indicates that traces are never invalidated after creation.

is complex and has high overhead. Since in our case the switching overhead is low, the invalidation is targeted at trace quality, rather than the complexity of code cache management.

IV.C.3 Trace Optimization Overhead

In our current study, an idle hardware context is assumed ready to run a dynamic optimization thread. However, we do not have to reserve an entire hardware context for dynamic optimization. Helper threads are invoked for optimization subject to the availability of hardware contexts. Hardware contexts are released after helper threads finish the optimization.

We measure the amount of time the dynamic optimizing helper threads spend executing hardware events. Figure IV.5 shows the percent of time relative to the main thread's execution, in which the helper thread is processing the hardware events. This is the amount of time needed to perform the base

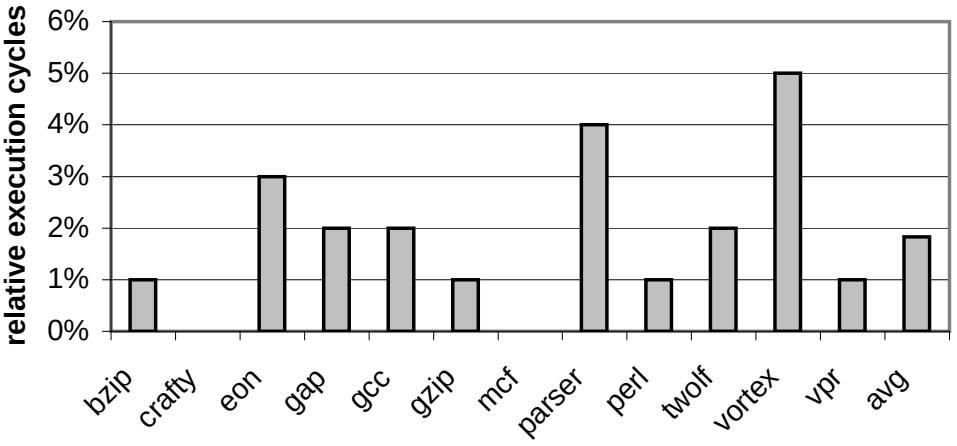

Figure IV.5: Percent of the main thread's execution in which the helper threads are running (processing hardware events). The main thread runs uninterrupted. The helper thread is triggered to run upon detection of an optimization event, and terminates after the optimization is done. The figure shows the accumulated execution time from helper threads relative to the total execution time of the main thread.

optimizations described above. The average execution ratio is less than 2%. Our simulation shows that each event on average takes 40,000 cycles to process. Since the optimization thread runs in parallel on the SMT processor, the actual negative impact on main thread execution is small, because our helper threads are spawned with lower priorities for instruction fetching. Since the optimization time is relatively small, our optimization technique should have opportunities to run even in a real multithreaded system with multiple threads running (e.g., even if contexts only become available during I/O).

To measure the cost of Trident trace formation and optimization (in regards to how it slows down the main thread), we run Trident with full optimizations without actually using the optimized traces. That is, the runtime optimizer is triggered to construct and optimize hot traces, but it does not alter

the original binary to jump to the optimized trace. The goal of this analysis is to determine the overhead the optimization code imposes on the system while it executes concurrently with the main thread. We observe the total cost to be only 0.9%. This is much lower than what would be expected in a traditional dynamic optimization system such as Dynamo [7], which would require runtime profiling and frequent switches between the main thread, the optimizer, and the profiler to enable similar optimization.

IV.D Color-based Code Placement

One of the most important benefits from dynamic optimization is instruction re-layout. Streamlined instruction blocks should improve the instruction cache behavior. But a naive implementation of code layout does not realize the full benefit if it does not control instruction cache conflict misses. Most discussions of dynamic optimization from the literature do not provide details on how to avoid I-cache conflicts between the optimized code and un-optimized code.

Here, we evaluate three different policies to lay out the optimized code in the code cache. The basis of these policies is cache block coloring. I-Cache blocks are partitioned into different colors. For example, if the I-cache has 512 cache blocks, we may partition it into 128 colors with 4 blocks in each color.

The first code placement policy, called *same color*, is to let the optimized code map to the same I-cache block as the original binary code. The original binary code may occupy non-contiguous memory blocks. In this policy, the optimized code is stored to a code cache location whose virtual address maps to the first I-cache block of the original binary code, and continues to occupy subsequent I-cache blocks as needed. The second policy is called *color bin-hopping*. In this policy, each trace created is assigned the next sequential color/bin where the prior trace created left off. The last policy, called the *coldest color*, maps a

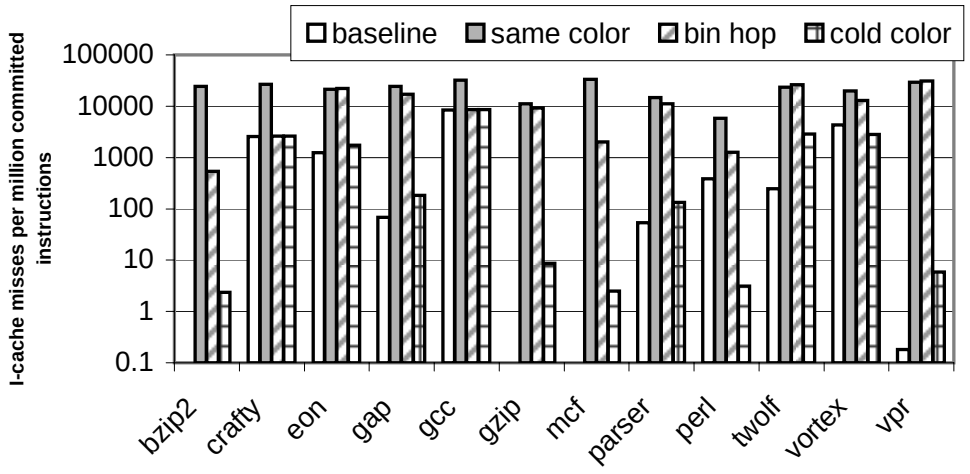

Figure IV.6: I-Cache misses on various placement policies. The first bar "baseline" indicates the I-cache misses when running the original binary on the baseline SMT processor alone.

new trace to the code cache block with the *coldest* color. That is, the I-Cache block corresponding to this code cache block is least accessed.

Figure IV.6 shows the instruction cache misses for these different policies. Cache misses are measured in every million committed instructions from the *original* binary. The *coldest color* policy achieves the equivalent, or fewer, I-Cache misses than the original binary running alone. The other two policies have higher miss counts due to conflicts between optimized and un-optimized code or between different parts of the optimized code. It can be seen that these schemes can differ by orders of magnitude in the number of conflict misses.

The amount of code generated and placed in our code caches varied from 1KByte (`mcf`) to 280KBytes (`gcc`). When using the cold color scheme, which spreads out the code to avoid conflicts, the continuous virtual address space used was up to 1.5 MBytes for `gcc`.

IV.E Architecture-Specific Optimizations

Software based dynamic optimization often results in high misprediction rates when using the traditional return address prediction stack (*RAS*) supported by most processors. The RAS uses a stack to predict return addresses, based on the prior sequence of procedure calls. During the basic optimization, both call and return instructions for inlined procedures are eliminated within hot traces. However, if the control flow exits the hot trace before the removed return is reached, the original code at the target of the exiting branch is executed. Since the return instruction in the original code may pop the RAS predictor (without a matched push), the RAS may predict wrong return addresses for many future predictions once it gets mis-aligned. Execution will be correct, but the misprediction cost will be high. This is a performance issue for any dynamic optimization scheme that eliminates calls and returns.

Kim and Smith [72] proposed a dual-address hardware prediction stack to tackle this problem, but the problem has not yet been explicitly addressed in most software dynamic optimization systems.

To solve this problem, Trident adds a compensation block in the optimized code for all exit branches which lay between a removed call instruction and the removed return instruction (if present). The compensation block contains a new instruction which does nothing but implicitly push the return address on the RAS. So, whether an inlined procedure is executed completely or not, Trident always keeps the RAS predictor in a consistent state.

Figure IV.7 shows the number of RAS mispredictions with and without our fix on the RAS predictor. We observed that the RAS misprediction rate can be as high as 97%, causing on average over 10% performance slowdown. Trident achieves RAS mispredictions equal to or less than the baseline, and it occasionally beats the baseline due to return elimination for inlined procedures.

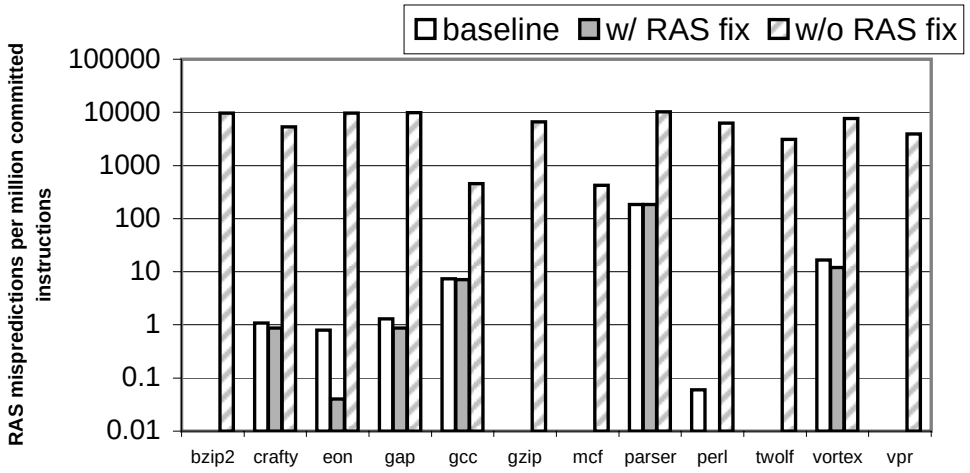

Figure IV.7: RAS mispredictions due to dynamic optimization. The first bar "baseline" represents the number of RAS mispredictions when running the original binary on the baseline SMT processor alone. The second bar indicates the RAS mispredictions with Trident's fix. The last bar is the RAS mispredictions, which would be expected when running current software dynamic optimization systems on the processor with the RAS prediction mechanism.

IV.F Summary

Trident is a trace based optimization system. Hot traces are formed and optimized when the helper threads are spawned to process hot branch events. Due to the concurrent execution of helper threads with the main execution thread, Trident introduces very little negative performance impact on the application. This is important to enable continuous profiling and optimizations. Trident's event monitoring and helper thread triggering provide a seamless mechanism for transparent dynamic optimization.

In this chapter, we show that Trident improves the hot trace detection scheme over previous systems, and uses a color-based layout policy to minimize I-cache conflicts between optimized and un-optimized code. Trident's optimization is also aware of the underlying microarchitecture, reducing mispredictions of the return address prediction stack (RAS) due to code optimization.

Acknowledgment

Portions of this chapter reproduce the material from the paper, *Weifeng Zhang, Brad Calder, and D.M. Tullsen, "An Event-Driven Multithreaded Dynamic Optimization Framework"*, in the proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005), September 2005, Saint Louis, MO. The dissertation author was the primary researcher and author and the co-authors involved in the publication [142] directed, supervised, and assisted in the research which forms the basis for that material.

V

Speculative Dynamic Value Specialization

Trident's fast event handling and low overhead make it suitable for aggressive optimizations. Due to Trident's event-driven nature, performance is highly insensitive to the latency of the optimizer. This allows us to optimize with more frequency, and to ultimately target more expensive optimizations than other systems.

In this chapter, we apply Trident to perform *Speculative Dynamic Value Specialization* (SDVS). Value specialization [19, 97, 49], sometimes done by a static compiler in a very conservative manner, is typically applied at the procedure level. A procedure may be cloned and individually specialized on typical input values which may be constants or have a relatively small number of distinct values. SDVS is unique in the sense that it exploits new opportunities for optimization by dynamically detecting semi-invariant runtime values and then applying compiler-like optimizations on these *constant* values speculatively.

Many of the specific specializations we do would provide a benefit anywhere value locality was detected. In this thesis, however, we focus on predicting

loads because a key advantage of the specialization is that it decouples a (potentially high latency) load from the dependent code. Prior research has shown significant potential from load value specialization [19]. At the same time, focusing on predicting loads also limits the size and complexity of the structures needed to collect the profiles.

V.A Related Work

In this section, we discuss prior research on code/value specialization and related value profiling techniques.

V.A.1 Code Specialization

Calder, et al [19] use offline profiling to identify top values of load instructions, and apply these values to the high level source code by hand to generate value specialized code. Their study demonstrates the potential of value specialization. Chung, et al [33] apply the same offline value profiling to clone the procedures with multiple copies, where each is specialized with different top parameter values. The goal of their research is to remove redundant calculations for power efficiency.

Fu, et al [49] study combined hardware and compiler techniques to improve value speculation scheduling. The static compiler does offline value profiling to identify candidate loads for runtime prediction. Then it inserts a new instruction to read the value from a hardware prediction table (instead of using the real load value) at runtime. Because reading from the prediction table takes only one cycle (i.e. the instruction latency is known), dependent instructions on the load can then be aggressively scheduled using static VLIW scheduling algorithms. The compiler also statically generates the recovery code to handle value misprediction. In comparison, Trident does value specialization dynamically. It does not

need any compiler-generated recovery code. The recovery for mis-specialization is done automatically using the existing hardware mechanism for mis-speculation recovery.

As discussed in Section II.B.3, Mock, et al. [93] developed a selective dynamic compilation system (DyC) to select a small portion of code for dynamic optimization according to particular runtime values. This portion of code is partially evaluated by the static compiler to reduce dynamic cost.

In research conducted in parallel with Trident, Shankar, et al. [121] explore runtime code specialization under the Jikes RVM. Runtime constants are identified by profiling heap object locations. Jikes RVM itself has built-in profiling (sampling) to help implement store profiling. During one sampling period, if there are no values stored to certain heap locations, these locations are considered as constants. Therefore, the corresponding values in the value profiler can be used for value specialization. It then picks an instruction trace from the dispatch point, similar to Dynamo's MRET trace selection [7]. For every single constant value, a specialized trace is created using that value. Multiple traces are then specialized from the same dispatch point. When the dispatch point is met in the interpreter, it jumps to the corresponding specialized trace according to the value of the load at that point. An influence metric is used to identify the dispatch point. The influence of a load instruction is determined by how many instructions down the instruction stream depend on this load. Because of the Java semantics and byte code interpretation, there are more opportunities for optimization than the statically optimized binary. The specialized traces do not necessarily start at branch boundaries.

Trident exploits the multithreaded processor architecture to collect hot execution traces and optimize them using a helper threading approach. Trident differs from the work above by identifying a single hot value for every candidate

load instruction and creating a single specialized trace (but potentially including multiple predicted loads). Instead of dispatching a specialized trace via an interpreter, Trident dynamically verifies the constants used during specialization, and recovers from mis-specialization using the existing hardware mis-speculation recovery mechanism.

V.A.2 Value Profiling

Value profiling has been used to guide static and dynamic optimization. Calder, et al. [19] use Top-N-Value tables (TNV) for fast and low overhead instruction profiling. This is a software based static profiling scheme. The goal of this research is to identify multiple top values for any individual load. Muth, et al. [97] generalize the notion of value profiles to expression profiles, which profile the runtime values of arbitrary expressions. The expression profiles allow more aggressive optimizations that may not be possible using simple value profiles.

Hardware based value profiling can be done using multihash tables [99] or a co-processor [147], as discussed in Section II.D.2. However, these profiling schemes are quite complex and hardware intensive. In this thesis, we design a hardware value profiler based on [19]. The profiler is defined as a small set-associative cache, indexed by the load PC. We introduce a value confidence scheme to select a single hot value from the top values. The confidence scheme is also used to replace values from the profiler.

V.B An Example of Dynamic Value Specialization

Dynamic value specialization benefits from two factors: (1) value prediction to break the instruction dependence chain, and (2) propagation of that knowledge (predicted values) further down the instruction stream to reduce computation. A code specialization example is shown in Table V.B. This example is

Table V.1: An example of code specialization from *parser*. The load in line 2 produces the value of zero with high frequency. Thus, the prediction on this load can further trigger instruction removal in lines 3, 4, and 5. After value specialization, the load in line 6 no longer depends on the load in line 2. These two loads can be issued and executed in parallel.

	The Original Trace		**The Value Specialized Trace**
1	LDQ R5, 104(R18)	1	LDQ R5, 104(R18)
2	LDQU R4, 0(R9)	2	*LDQU R4, 0(R9)*
3	EXTQH R4, R0, R0		(*removed*)
4	SRA R0, 56, R0		(*removed*)
5	S4ADDQ R0, R5, R0		(*removed*)
6	LDL R0, 0(R0)	6	*LDL R0, 0(R5)*
7	AND R0, R16, R0	7	*AND R0, R16, R0*
8	BNE R0, next	8	*BNE R0, next*
	...		

taken from the benchmark *parser*.

In this example, the second load produces the value of zero with very high frequency. Line 3 extracts the high byte from this value to R0, and line 4 does a right shift on the value. Since the value is predicted as zero, both of these two instructions produce the zero value again. Line 5 adds the value of zero to register R5 and stores it into R0, which has the same value as R5. Therefore, in line 6, we can directly use R5 as the source operand. Since register R0 is redefined multiple times inside the same basic block, we can safely remove instructions 3, 4, and 5. After value specialization, line 6 no longer has a dependency on line 2. Thus, these two loads can be issued and executed in parallel. The value specialization above can significantly reduce the program's critical dependence

path of execution.

Compared with traditional hardware value prediction, dynamic value specialization has two primary benefits. First, it simplifies the implementation of value prediction in several ways: (1) prediction decisions are made much less frequently and are performed in the back end of the pipeline, (2) fewer of the predictions actually require injecting values into the register file – many times the prediction is accomplished simply by transforming the code, and (3) the latency requirements for tracking and acting on load value locality are severely relaxed. The second primary benefit of applying this optimization in the trace is that it allows further optimization through propagating these values through the trace.

V.C Dynamic Value Specialization Architecture

In this section, we augment the Trident framework with a hardware value profiler to identify frequently occurring values from load instructions inside hot traces. This monitoring structure can generate a new optimization event, called the *Hot Value* event. When any semi-invariant runtime (*hot*) values are detected, Trident then applies compiler-like optimizations on these "constants" in the helper thread.

The architectural flow of dynamic value specialization is shown in Figure V.1. A hot trace is formed upon the hot branch event. When the hot trace is executed later, load instructions in the trace are monitored. Upon a hot value event, the helper thread takes the hot trace and modifies it to form a new value specialized trace. When the specialization is done, the helper thread stores the new trace into the code cache, and alters the original source binary code to jump to this newly optimized version. Note that this replaces the link to the optimized hot trace that was earlier created by the Trace Construction code. Therefore, the

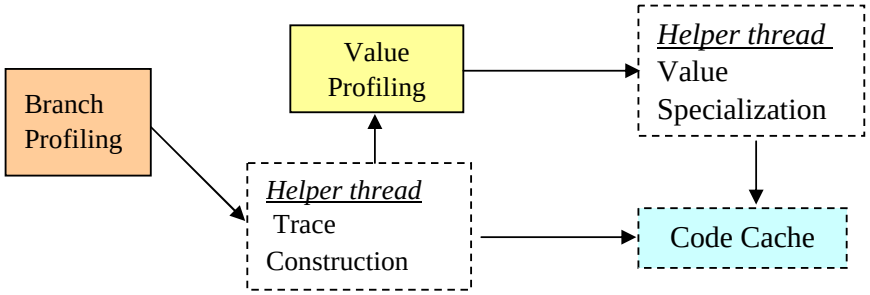

Figure V.1: Trident speculative value specialization architecture. The value profiler monitors load instructions inside hot traces. When the profiler detects a semi-invariant value, it triggers the hot value event to perform value specialization on this hot trace with this value.

old trace can now be invalidated, and is marked to be removed later during code garbage collection.

V.C.1 Hot Value Profiler

Our hardware value profiler is based on the software value profiler from [19], but we make a few improvements for more efficient hardware implementation. The profiler is organized as a set-associative cache, where each entry is assigned to track the top values for a load. As shown in Figure V.2, each profiler entry keeps track of a small number (e.g. five) of load values that are treated equally instead of being placed in the steady and clear partitions as in the software value profiler. We add a new entry to keep track of the dominant stride seen between the values for the load. Finally, we introduce a value confidence scheme so that only one value can be selected for value specialization at any time.

Each value has associated with it a confidence counter (typically 4 bits). The confidence scheme is represented by a tuple of <*max confidence, increment, decrement*>. For instance, our default scheme is <15, 1, 7>, where a value's confidence is incremented by 1 if the same value occurs again, and if a different

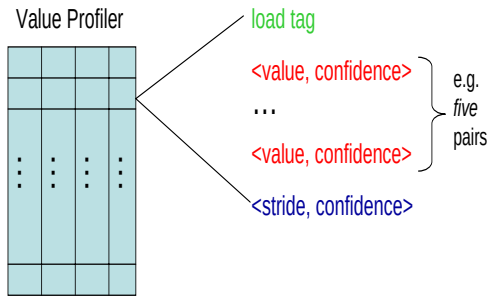

Figure V.2: The Trident value profiler is organized as a set associative cache. Each cache entry keep tracks of a load's top values as well as its dominant stride. The value confidence scheme only allows one value to be selected for value specialization. If no top values are confident, the stride value may be used if it is confident.

value occurs the confidence for that entry is decremented by 7. The confidence is saturated at 15. Whenever a value's confidence reaches 15, it is claimed *hot*. When this occurs, a hardware event is generated to indicate that the load is value predictable for that value.

Similarly for the stride entry, we use the same confidence scheme. Here we calculate the stride between the last value and the current value, and we compare the stride to the one stored. We increment if the stride is the same as the last one encountered, decrement if it is different. If the confidence counter is 0, we replace the stride, and if it is saturated at the max confidence then a hardware event is generated to indicate that a load has a stride predictable value.

When a load instruction is committed, its PC is used to index into the cache. Each time a load PC gets a tag hit, we update *all* of the confidence counters for that load PC. If the value is not present, then the least confident value is replaced.

In this thesis, we use the hot value profiler to only monitor load instructions in the scope of hot traces. When a load is committed, it is inserted into the

value profiler if it is from a hot trace. If there is no room in the cache, then the least recently inserted entry is replaced.

V.C.2 Hot Value Events

Trident can perform value specialization on any hot trace via semi-invariant load values found in the hot load profiler. Whenever one of the values inside a value profiler entry is confident, the value is *hot*, and the profiler raises a *hot value* event. When a hot value event is consumed by a helper thread, the *Speculative Value Specialization* code is run. The thread performs optimizations (described in the next section) on the hot trace.

V.D Implementation of Dynamic Value Specialization

Trident performs speculative value specialization using the predicted "constant" upon the hot value event. Value specialization, which includes constant propagation, copy propagation, and redundant code elimination, takes the following steps:

- Construct a def-use chain on the trace. Hot values are then propagated along the dependence chain. Any new constants generated during the propagation are further propagated.
- If the load values are special integers (such as 0 and 1), consumer instructions of these values may be strength reduced. For example, a register multiplying with a zero or one produces a zero or itself. Thus, we can reduce this instruction to a simple *move* instruction. The *move* instructions generated after the strength reduction are copy propagated along the trace. Branch instructions depending on these "constants" may be eliminated.

- After the copy propagation is done, *move* instructions may be eliminated if their destination register are redefined inside the same basic block as the *move* instruction. We perform this optimization, since we can prove the register is dead.

V.D.1 Verifying the Specialized Load Value

The loads that are value specialized need to be checked against the semi-invariant value. Trident directly embeds the predicted values into the newly created specialized trace, as in [19]. This allows the load's dependence chain to be broken and results in reducing instructions on the critical path. In addition, code below the value specialized load (and below the check and branch) can speculatively execute before the load value comes back from the lower memory hierarchies.

The following code sequence is generated for each load instruction which is value predicted during specialization:

- Perform the original load into a scratch register.
- *move* the predicted value into the load's original register.
- *compare-and-jump*: compare the load value with the predicted value register. If different, jump to recovery code at the end of the trace.

The recovery code at the end of the trace will be:

- *move* the original load value in the scratch register into the original load's destination register.
- *jump* back to the next instruction after the load in the original binary. This essentially ends the trace prematurely, which will be seen by the watch table, and if this occurs enough times the trace will be invalidated.

The above does put restrictions on instruction scheduling, since instructions cannot be hoisted above value specialized loads. If the predicted values are correct, the *compare-and-jump* should not be taken. In case of an incorrect value, since the load destination register is re-set with the correct value, all subsequent instructions (after we branch back to the original trace) will get the correct value.

A register can be used as a scratch register if it is redefined in the same basic block as the load instruction, and has not been used between its redefinition and the load verification. If such a scratch register is unavailable, the predicted value cannot be efficiently verified, so this load instruction is skipped for specialization. For our results, we rarely had to skip the specialization. For architectures with more register constraints in their ISA, more aggressive register scavenging might need to be performed.

V.D.2 Exploring Stride Values

Trident's speculative value specialization is also able to explore runtime values with semi-invariant strides. To the best of our knowledge, this is the first time stride values are exploited by a software optimizer during value specialization. We found that stride prediction is particularly useful for certain pointer-chasing code, as also seen in [35, 125]. This is due to the fact that some programs' allocation of data and its traversal over the data are through highly strided access patterns.

To benefit from this, Trident's value profiler keeps track of the confidence of a load instruction's value stride as described in Section V.C.1. If the stride is confident, it can be used for specialization if no other top values are confident. For a load instruction exhibiting a stride value pattern, its *true* value is the sum of its previous value (i.e., *base value*) and the stride. To calculate a load's true value, we store its base value in a separate main memory buffer, called the Base

Value Memory Buffer (*BVB*), and directly embed the stride value into the hot trace in the code cache. The *true* value is verified via the following code sequence. However, the predicted value is not propagated for further optimization.

- Perform the original load into a scratch register.
- *load* the predicted value from the *BVB* into the load's original register.
- *compare-and-jump*: compare the load value with the predicted value. If different, jump to the recovery code appended at the end of the trace.
- *add* the scratch register with the *constant* stride from the value profiler and store the sum into the BVB.

The recovery code at the end of the trace will be:

- *move* the original load value in the scratch register into the original load's destination register.
- *add* the scratch register with the *constant* stride from the value profiler and store the sum into the BVB.
- *jump* back to the next instruction after the load in the original binary. This effectively prematurely ends the trace, which will be seen by the watch table, and if this occurs enough times the trace will be invalidated.

The key idea of this scheme is the assumption that the predicted next stride value stored in BVB, which is used in the hot value specialized trace, should rarely miss in the data cache. This will provide the predicted stride value when accessing it, and if it is a hit in the L1 it should be significantly faster than traversing through pointer-chains. To aid this, our value specializer picks a data address for the BVB that maps to a cold color based upon the cache access counters. For results in this chapter we only need eight entries in a BVB for a given program.

Table V.2: Trident value profiler configuration. The profiler can monitor up to five top values and one dominant stride for a single load.

Value profiler	128-entry 2-way associative; each entry has five values and one stride value. Each value has a 4-bit confidence counter. Confidence scheme: <15,1,7>

V.E Methodology

Speculative value specialization is evaluated using SPEC 2000 int benchmarks on the simultaneous multithreading processor (SMT). The baseline SMT configuration and benchmark simulation points are described in Chapter IV.B. The performance of benchmarks on the baseline SMT processor is shown in Figure IV.1.

The configuration for the Trident value profiler is shown in Table V.2.

V.F The Performance of Value Specialization

In this section, we evaluate the performance of Trident's speculative value specialization, and compare it with traditional (hardware based) value prediction. In value specialization, the value confidence scheme in Section V.C.1 is used to identify hot values. All speedups quoted are instruction throughput of the main program relative to its instruction throughput without value specialization, using instruction counts that correspond to original program execution.

V.F.1 Comparison with Value Prediction

One of Trident's advantages is that a value-specialized hot trace may embed many value predictions. In contrast, the complexity of conventional hard-

ware value predictors would likely limit how many predictions can be made each cycle. We compare Trident's value specialization with an aggressive hybrid predictor proposed by Wang, et al. [136]. The predictor has a value history table (VHT) of 4K entries, where each entry has seven values. The VHT entry is used as an index into a pattern history table (PHT) of 32K entries. Each PHT table entry then has seven counters, which are used to keep track of which of the seven values in the VHT entry to use for the prediction. Each cycle, the hybrid predictor is assumed to make up to 4 or 8 predictions. For example, the predictor may try to make predictions only for the first four load instructions encountered per fetch. Note that this predictor likely makes an unrealistic number of predictions per cycle. Also, the nature of this predictor should allow it to identify patterns our system cannot predict.

Trident's performance is shown in Figure V.3. The first and second bars show the performance of the hardware predictor with 4 and 8 predictions per cycle, respectively. The third bar shows the basic dynamic optimization, which involves basic block inlining and redundant instruction elimination, as described in Chapter IV. The last bar shows the benefit from value specialization.

We make several observations from this data. First, we see that the performance gains from our basic dynamic optimization implementation are relatively low. However, this provides the framework for further optimizations – in this case it enables the dynamic value specialization, where we see significant performance gains, averaging over 20%. We also see that our value specialization significantly outperforms aggressive hardware value prediction. While hardware value prediction can break dependencies between the load and its dependences, beyond that the knowledge that the value is a constant is lost. However, with our dynamic value specialization the knowledge is propagated down the dependence chain, allowing gains well beyond the initial prediction.

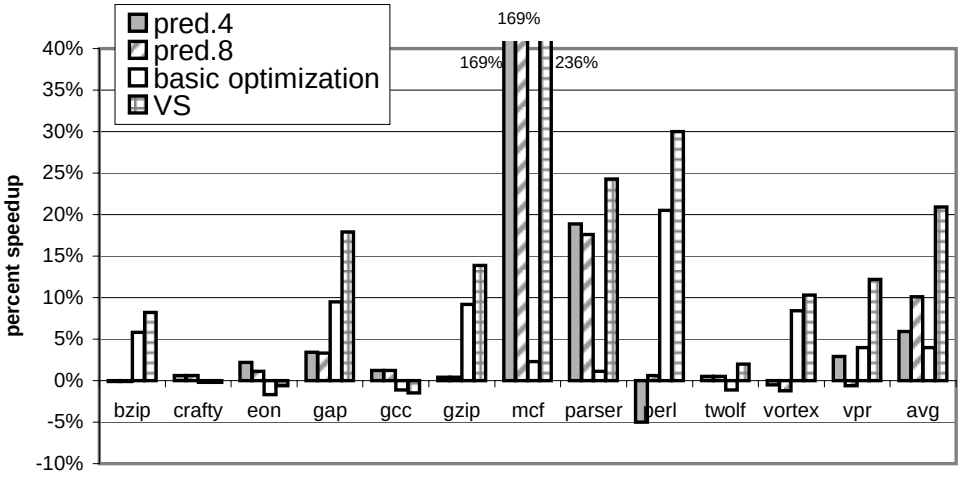

Figure V.3: Comparison of value specialization with value prediction [136]. The first two bars show traditional value prediction with 4 and 8 predictions per cycle, respectively. The last bar shows speedups from speculative value specialization.

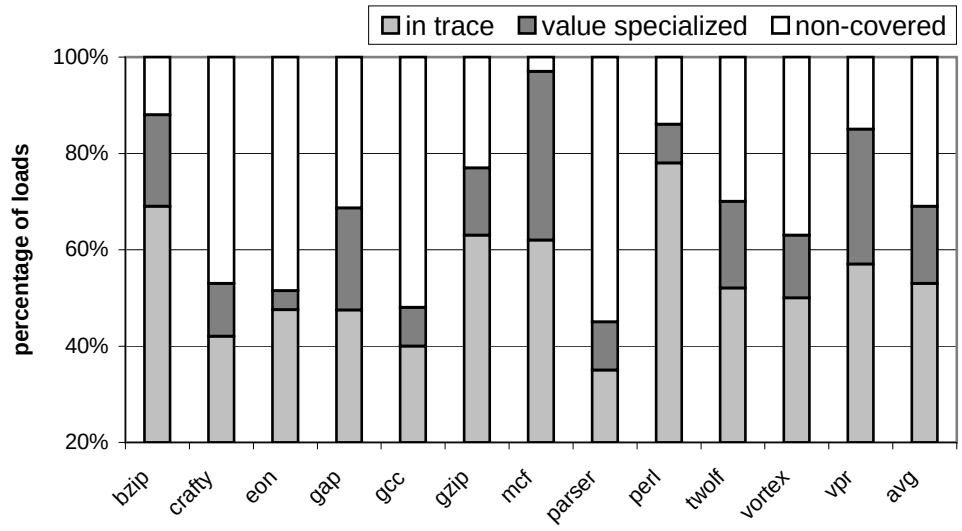

Figure V.4: Breakdown of dynamic load instructions. 100% represents total dynamic instances of all load instructions. The "in trace" represents all loads inside hot traces, which have potential to be optimized during base optimization. The "value specialized" stands for loads being value specialized. The "non-covered" shows all loads falling outside hot traces.

Figure V.4 shows the breakdown of load instructions that are covered by the hot traces with value specialization. The lower light gray shows what percentage of the loads were not value specialized in the hot trace, and the dark gray shows what percentage were value specialized. The rest of the loads (top part of each bar) were not executed in the hot traces. About 70% of the dynamic load instructions are within hot traces, and among them 16% are value specialized. `mcf` shows that it spends most of its execution time in the optimized hot traces. It benefits from value prediction, which decouples the load from the dependent instructions, and it also benefits from stride value specialization for providing pointer-chaining addresses. The stride value specialization accounts for 105% of the 236% speedup seen for `mcf`. These optimizations allow subsequent (previously dependent) loads to overlap.

V.F.2 Comparison with Load Prefetching

One significant benefit of our dynamic value specialization is that it tolerates long memory latencies by decoupling them from the dependent computation. However, other memory latency tolerant solutions may already provide the same benefit. A common mechanism is hardware prefetching. We want to see if Trident is still effective in the presence of these mechanisms. As such, we implemented a very aggressive load stream prefetcher proposed by Sherwood, et al. [125]. The prefetcher has 8 stream buffers, which each have 16 entries. The PC-stride predictor table has 256 entries, and it has a small Markov predictor with 2048 entries. Performance comparison between predictor-directed stream prefetching and Trident is shown in Figure V.5. The first bar shows the IPC improvement from hardware stream prefetching. The second bar shows the performance improvement from value specialization when combined with stream prefetching. Comparing Figure V.3 with Figure V.5, Trident's value specializa-

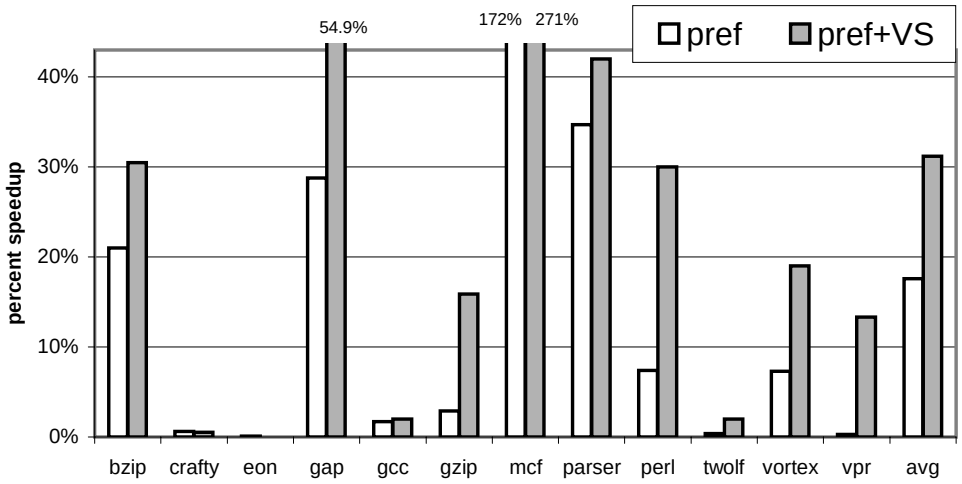

Figure V.5: Performance of Value Specialization with Prefetching [125]. The first bar is the performance from hardware stream prefetching alone. The second bar represents the performance of the combination of hardware prefetching with speculative value specialization.

tion alone outperforms the hardware prefetching. Value specialization boosts the performance of *bzip*, *gap*, *gzip*, *mcf*, and *vpr* when combined with hardware prefetching. Trident's value specialization is complementary to hardware prefetching, showing strong gains on top of prefetching alone.

V.G Summary

In this chapter, we demonstrate Trident's effectiveness via software speculative value specialization. We extend Trident with a hardware value profiler to exploit semi-invariant runtime values and stride values. The profiler raises a hot value event when a load's value becomes confident. The hot value event then triggers Trident to perform speculative value specialization on the hot trace. Our simulation shows that value specialization can achieve over 20% speedup on average. It is shown to be a promising technique for tolerating memory latencies, even

in the presence of aggressive hardware prefetching. Value specialization extends the benefit of value locality further down the dependence chain than previously proposed hardware prediction mechanisms.

Acknowledgment

Portions of this chapter reproduce the material from the paper, *Weifeng Zhang, Brad Calder, and D.M. Tullsen, "An Event-Driven Multithreaded Dynamic Optimization Framework"*, in the proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005), September 2005, Saint Louis, MO. The dissertation author was the primary researcher and author and the co-authors involved in the publication [142] directed, supervised, and assisted in the research which forms the basis for that material.

VI

Adaptive Dynamic Software Prefetching

Memory latency has become a dominant factor in the performance of modern processors. One way to attack this problem is to prefetch the memory values before they are actually consumed. The rationale behind prefetching is to overlap the long latency load operation with other useful work. Thus, prefetching decreases the observed latency, increases memory level parallelism, and allows cache-hit dominated performance even when the working set is larger than the cache.

Load prefetching can be classified into prediction-based prefetching and execution-based prefetching. Prediction-based prefetching can be done by the compiler to explicitly insert the prefetching instructions (called *inlined prefetching*) to bring the data into the cache [95, 20, 138, 85, 31, 62], or by a hardware prefetching mechanism [67, 28]. Hardware prefetching dynamically predicts memory load addresses that are hard to find statically. The execution-based prefetching [87, 146, 37] takes advantage of hardware features in a modern processor architecture, like Simultaneous Multithreading (SMT) [133]. It uses a

spare hardware thread to pre-execute the load instructions speculatively [37]. The speculative pre-execution thread runs ahead of the main thread to execute the precomputation slice (p-slice). The speculative thread helps bring values into the data cache, thus absorbing the memory latency on behalf of the main thread. This mechanism is useful if the precomputation thread is able to run far ahead of the main thread. We will study this in more detail in next chapter.

In this chapter, we focus on improving inlined software prefetching using the Trident optimization framework. We extend Trident's hardware support and dynamic optimizer to permute the object code by inserting software prefetch instructions. In this approach, basic *hot traces* formed by Trident are monitored by hardware to detect *delinquent loads*, which frequently miss in the data cache. Upon detection of such loads, hardware-generated *hot events* trigger the execution of a software thread to perform optimization. The thread inserts prefetch instructions into the original hot trace to create a new trace, which may prefetch multiple delinquent loads.

We also find that Trident's low overhead makes it possible to pursue more aggressive optimizations by applying given optimizations repeatedly. This allows continuous incremental improvement or even allows the system to use trial and error to apply an optimization most effectively. This is the motivation behind our approach to support adaptive software prefetching.

VI.A Related Work

In this section, we discuss prior research related to inlined prefetching and hardware prediction based prefetching.

VI.A.1 Hardware Based Prefetching

Hardware based prefetching relies on hardware address predictors to generate and guide prefetching stream. The prefetching efficiency depends on how accurate the addresses can be predicted.

Chen and Baer [28] use a Look-Ahead PC (*LA-PC*) to initiate load prefetching ahead of the normal fetch engine. The LA-PC is guided by the branch predictor to predict load addresses. Joseph and Grunwald [65] propose Markov prefetching using current cache missing addresses to index into the Markov prediction table to predict addresses for prefetching.

Smith and Hsu [128] propose to enable next-line prefetching by tagging the cache structure with *prefetch* bits. When a cache block is fetched and its prefetching bit is not set, then its next sequential block is prefetched. Jouppi [67] introduces stream buffers as a more efficient next-line prefetching architecture. The stream buffers allow multiple prefetching streams to run in parallel, and prefetch data multiple iterations ahead of the current fetch stream. Palacharla and Kesseler [102] propose a non-unit stride detection scheme to enhance the effectiveness of stream buffers. This model was further extended by Farkas, et al. [45] to use a PC-based stride predictor to predict strides on a per load basis. This is different from the minimum-delta stride scheme, which uses the global miss addresses to calculate the stride for a given load. Thus, a stream buffer should get higher prediction accuracy since the prediction is based on the history of the load which the stream buffer is allocated to. Sherwood, et al. [125] extend the above architecture to use a stride-filtered Markov predictor to guide the prediction stream. The predictor-directed stream buffer (*PSB*) can generate the next prefetch address without a fixed stride if a Markov transition is found. Timely prefetches may be achieved by allowing the stream buffers to run independently ahead of the execution stream.

Hardware based mechanisms directly detect memory access patterns from load address streams. While our approach maintains much of the runtime adaptability of these hardware schemes, it can target more complex memory access behaviors, because it is based on analysis of the actual code.

VI.A.2 Software Inlined Prefetching

Software prefetching explicitly inserts prefetch instructions into the code. We call this *inlined* prefetching. A large amount of research has been done on compiler-enabled software inlined prefetching [20, 95, 88].

Luk and Mowry [88] examine pointer chain prefetching for *Recursive Data Structures* (RDS). They also add *jump pointers* to prefetch heap objects farther in advance than one pointer traversal. Roth and Sohi [115] extend the jump pointer prefetching technique via a software/hardware scheme to provide various trade-offs between accuracy and prefetching overhead. Cahoon and McKinley [16] propose a greedy prefetching technique to target RDS traversals in Java using data-flow analysis. Zhang and Torrellas [144] employ user-added annotation to mark up objects. These instructions mark the data objects to be grouped together. The grouping information is stored in a hardware buffer, and any miss in the group triggers hardware to prefetch all data objects in the group. Yamada, et al. [140] propose a compiler-assisted hardware technique to combine data relocation and block prefetching to improve memory performance. Saavedra and Park [117] propose an adaptive execution scheme in which the compiler inserts software prefetches and generates a *software agent* to control these prefetches at runtime. This scheme uses a single prefetching distance to control all prefetches within the whole loop body. In contrast, our technique targets true cache misses by dynamically generating software prefetches which are tuned to each individual load.

Statically inserted prefetching may not work well across different data inputs or different architectures, and does not allow the changing of the prefetch instructions for legacy code. Our self-adapting software prefetching extends prior research by applying some static prefetching techniques dynamically. Dynamic prefetching allows prefetch instructions to be dynamically inserted or changed, and to target true delinquent loads. Our technique also enables effective prefetching by automatically adapting prefetches to the program's runtime behavior. In addition, our technique works transparently on existing binary code.

VI.A.3 Prefetching via Dynamic Optimization Systems

Inagaki, et al. [62] extend an efficient software profiling algorithm [138] to target both intra- and inter- loop stride loads in a dynamic optimization system. Their technique mainly focuses on Java compilers. They proposed a lightweight profiling technique by interpreting the Java object a few times to identify load access patterns. Compared with our approach, profiling via interpretation still imposes a high overhead. In addition, our prefetching works for general purpose and even legacy programs.

Chilimbi and Hirzel [31] propose an automated approach to inject prefetching code into hot data streams based on the correlation of hot data reference sequences. This scheme gathers a temporal data reference profile via bursty sampling, and extracts data reference patterns frequently occurring in the same order. Prefetching is inserted dynamically at proper program points to prefetch these references. Compared with our approach, this scheme has higher runtime overhead due to software profiling, and requires static binary instrumentation.

Lu, et al. [85, 25] developed a dynamic optimization system, called *ADORE*, to perform software prefetching on delinquent loads. ADORE analyzes the code in the hot trace to identify load access patterns. Loads with more compli-

cated patterns are predicted using the profiling algorithm described in [138]. The prefetching distance is calculated according to the load's average miss latency. Our technique builds on that research, but it has clear distinctions from their work. First, our prefetching technique adapts the prefetch distance and repairs the hot trace instead of having to re-generate the entire hot traces. This gives us the ability to efficiently and adaptively search for the optimal prefetch distance. Second, we perform the same-object based prefetching to avoid redundant prefetches.

VI.B Dynamic Software Prefetching Architecture

The goal of this research is to use dynamic trace optimization to improve the performance of the memory subsystem for a thread. The following provides a high level overview of how Trident works and how our prefetching approach works inside of Trident:

- **Trace Formation and Linking Trace.** The branch profiler triggers the helper thread to form a hot trace and link it into the program's execution, as described in Section III.B and Section IV.A.

- **Monitor Trace Loads.** We add a hardware structure called the Delinquent Load Table (DLT), as shown in the Table VI.B, to Trident to monitor the performance of loads that are executed on these hot traces. In Trident, when an instruction is committed, the hardware knows if it resides within a hot trace formed by Trident based upon Trident's hardware *watch table*. We therefore update the DLT with only loads that are in hot traces. Note that the watch table also monitors a trace's minimal execution time, and we will describe its use in Section VI.E.3. Table VI.B shows all of the fields in the watch table and the DLT. The fields in the DLT will be described in more

Table VI.1: Trident delinquent load table. The watch table is augmented with an optimization flag. This flag works together with the delinquent load's maturing flag to avoid over-optimization.

Watch table	Trace starting PC Trace length Trace minimal execution time Trace optimization flag
Delinquent load table (DLT)	Load tag Access counter L1 miss counter Total miss latency Stride Stride confidence bits Last effective address Mature flag

detail in the rest of this section.

- **Delinquent Load Event.** When a hot trace load misses in the memory hierarchy, we then look up the DLT and determine if it meets the criteria to be classified as a delinquent load. The criteria are that the load's miss rate is above a threshold, and the load's average memory latency for the last M misses is larger than the half of the L2 miss latency. If both of these conditions are true, then the DLT will trigger a Delinquent Load event. When a delinquent load event is triggered for a hot trace, we set a bit in the Trident watch table for that hot trace to indicate that the hot trace is currently being re-optimized. This is to prevent other re-optimization events from being triggered for that hot trace while we are doing our optimization.

- **Insertion of Prefetches into Hot Traces.** A *delinquent load* event will trigger the execution of the helper thread described earlier (if a context is available). The helper thread runs our software optimizer to perform the prefetch insertion algorithms described below.

- **Linking in the Re-Optimized Hot Trace.** Once the trace is re-optimized Trident links it into the execution by re-patching the original binary to jump to the newly formed trace, and Trident removes the old hot trace from the hardware watch table. A thread's execution will then automatically start using the new hot trace.

VI.C Delinquent Load Table

The *Delinquent Load Table* (DLT) generates delinquent load events to trigger a helper thread to perform dynamic optimization and insert software prefetching instructions. The DLT is organized as an associative cache, indexed by the load PC. It uses the least recently used replacement policy.

To determine if a load is delinquent or not, we examine the miss rate and average miss latency after a load has been executed N times. This is called the *load monitoring window*. After N accesses, these statistics are calculated to determine if the load is delinquent, and then the counters are cleared, and the load is re-examined at the end of the next load monitoring window (after the next N accesses). The DLT keeps track of the following information for the load:

- **Access counter.** This counter keeps track of how many times this load has been accessed during a given monitoring window. At the end of the window (i.e. N accesses), it resets itself (along with other counters as described below) to start a new round of counting.

- **Miss counter and miss latency.** The miss counter keeps track of how

many cache misses this load encounters during the current window. Together with the access counter, it provides an approximate miss rate within a monitoring window.

When a load misses in the cache, the miss counter increments and its miss latency is added to the sum. At the end of the window, a load is claimed as delinquent if (1) its miss counter reaches a threshold (i.e. miss rate is above the threshold), and (2) its average miss latency is higher than half of the L2 miss latency. Here, the average miss latency is calculated as the total miss latency divided by the total miss count. The delinquent load then triggers a delinquent load event, which in turn invokes the helper thread to run. These counters and total miss latency stay unchanged and will be cleared later by the helper thread during optimization.

If the access counter reaches its threshold before the miss counter, the load is not delinquent. At the end of the window, the access and miss counters are reset, and the monitoring of the load continues.

- **Stride address prediction.** Our software prefetching optimization focuses on taking advantage of stride predictable loads. Therefore, each DLT entry keeps track of (a) the load's last address, and (b) the load's last address stride, and (c) a 4-bit address stride confidence counter. These values are updated every time the load is committed (not just on misses). The confidence counter starts with the value of 0 and is incremented by 1 if the current stride equals the last stride, and decremented by 7 if they are different. A load is said to be stride predictable if the stride confidence counter is 15. Although in many cases the stride can be identified by analyzing the code in the hot trace, the hardware support allows us to identify a large number of pointer loads that turn out to have stride access patterns, due to the way memory structures are allocated and used. This allows effective

prefetching of loads that a static software prefetcher will have great difficulty with.

- **Prefetch mature flag.** This flag is used to indicate if a load has been tuned enough times. We want to avoid generating too many delinquent load events for a load that our prefetch algorithm can not cover or hide all of the latency for. If the mature flag is set, then the load will not generate a delinquent event on a miss.

VI.D Dynamic Prefetch Optimizer

In Trident, the runtime optimizer is triggered to run as a helper thread on an idle hardware context when a delinquent load event is detected. If a prefetch instruction has not been inserted into the hot trace to prefetch this delinquent load, the prefetch optimizer will generate a new trace and insert a prefetch instruction to target this load. Otherwise, the optimizer will try to repair the prefetch instruction as described in Section VI.E. Before the optimizer finishes, it resets the hot trace's optimization flag so that it can be re-optimized in the future.

VI.D.1 Delinquent load identification

During prefetch insertion, the dynamic optimizer first identifies all delinquent loads within the trace, and then partitions these loads into different types so that prefetch instructions can be inserted accordingly.

Because there are at least a few thousand cycles between when the delinquent load event is triggered and when the dynamic prefetch optimizer is ready to start its execution (if there was no contention for the spare thread), the optimizer first checks if there are other loads that need to be prefetched in the

same hot trace. To identify all delinquent loads in the hot trace, we look up each of the loads in the DLT. If they satisfy the delinquent load classification described in Section VI.C, then they are added to the delinquent load list. Note that if a load has not yet completed execution of a full monitoring window, its miss rate and latency are calculated using current counter values in a partial monitoring window.

After all of the delinquent loads are identified in the trace, the optimizer then classifies all of these delinquent loads as Stride, Pointer, or Same Object based upon the following criteria:

- **Stride.** For a load instruction within a loop, if the recurrence between instances of the load is a single simple arithmetic instruction (e.g. LDA, ADD, or SUB) whose arguments are a constant and a register, then this load is classified as a stride load. This simple definition picks up most strided loads. We also mark any load the DLT found stride-predictable, which picks up more complex recurrences.

- **Pointer.** If the load is not classified as Stride, then we check to see if it is a pointer load. If this load's destination register is used, before any modification, as the base register of any other load instruction, then the destination register contains a pointer value. So this load is classified as a Pointer load.

- **Same Object.** To enhance the effectiveness of the prefetcher, we perform whole object prefetching. Prior results [119], confirmed by our own data, show that a significant portion of misses are to other fields in a recently-accessed object. Our optimizer recognizes accesses to alternate fields of the same object, and prefetches all needed offsets as soon as the base address is available. This work is different than previous whole-object prefetchers [62] in its adaptability, in its elimination of redundant prefetches, and in that it

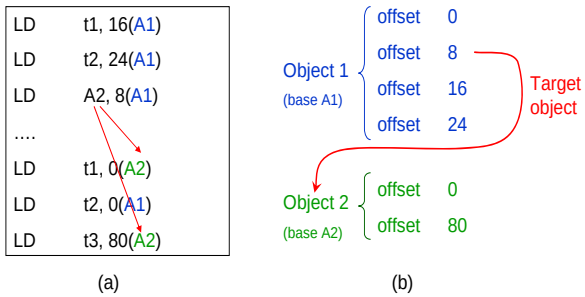

Figure VI.1: An example of object groups. In this example, there are four loads with the same base register (*A1*), and other two loads with the base register *A2*. Two objects are formed in (b). The second object is the target object of the pointer load in the first object. Object 1 can be prefetched using a single prefetch instruction. Object 2 needs two prefetch instructions.

is guided by the actual runtime cache miss profile. Furthermore, the same object based prefetching helps enable our proposed adaptive, self-repairing prefetching mechanism.

For each delinquent load classified as Stride, the optimizer searches both forward and backward on the hot trace for other loads with the same live base register. If these loads exist, then the optimizer puts them into a group, called *Same Object* group. The end result is a set of same object groups, where each set contains at least one delinquent load that is stride predictable. As shown in Figure VI.1 (a), there are four loads that have the same base register. Thus, Trident forms an object to group these loads. Note, a delinquent load can only belong to at most one group, and a group can contain more than one delinquent load. The degenerate case is that a group can consist of only one single load, which is the stride address delinquent load.

If we have multiple loads using the same base register, which has been identified as a pointer, we also classify those loads as same object and prefetch

them together. The same object classification also allows us to prefetch multiple loads with a single prefetch instruction, and eliminate redundant prefetches to the same cache line. When applying our self-repairing optimization, it allows us to also repair all the object prefetch distances as a group, rather than one at a time with separate optimization events.

Any load instruction that is not classified as one of the above types will not be prefetched in our current framework.

VI.D.2 Stride-Based Prefetching of Same-Object Loads

We first focus on stride address predictable groups, because these are the loads for which we can perform timely prefetching. As long as a same object group has at least one delinquent load that is Stride predictable, then the whole group is classified as stride address predictable.

Each stride address predictable same object group is processed using the following algorithm:

- Find the minimum load offset from the base register in the group.
- Insert a stride prefetch instruction using the group's base register and the minimal offset as this format:

$$prefetch \qquad (offset + stride)(base) \tag{VI.1}$$

- Find the delinquent load with the next smallest offset. If its offset from the prior prefetch is less than the cache line block size, simply mark this load as prefetched and skip it; otherwise, insert another stride-based prefetch as above with the non-covered delinquent load's offset. When a load is skipped, the offset plus the base register may put that load into the next cache block, which should have been prefetched. To address this, we prefetch

one additional cache block after a skipped load. This still allows us to skip several loads, and only prefetch each block once.

This process repeats until all delinquent loads in the group are processed.

VI.D.3 Prefetching for Pointer Loads

After the stride-based same object prefetching is done, the only delinquent loads we target for additional prefetching are pointer loads if they have not yet been processed in the algorithm above. For example, a pointer chasing load in a loop looks something like:

$$ld \quad r1, offset(r1) \tag{VI.2}$$

and we dereference this pointer twice by inserting the following instructions after the above instruction in the hot trace:

```
ld        scratch, offset(r1)
prefetch  offset(scratch)
```
(VI.3)

These two instructions potentially prefetch the object in the next two iterations of the loop. Notice that the first instruction should be a non-faulting load. We found a significant portion of pointer loads prefetchable using this stride-based same-object algorithm.

Note that if a pointer load belongs to a Same Object group, the pointer is also dereferenced right after its stride-based prefetch instruction.

VI.E Adaptive, Self-Repairing Prefetching

VI.E.1 Prefetch Distance for Stride Address Predictable Loads

The stride prefetching insertion algorithm described above only prefetches one iteration ahead in a loop for the object. What we really want to do is determine how far ahead to prefetch an object, which is called the prefetch distance.

Prefetching can target loads which miss at different memory levels. Existing dynamic prefetching systems such as [85, 25] estimate the prefetch distance (in number of iterations) as:

$$distance = \frac{average\ load\ miss\ latency}{average\ cycles\ per\ iteration} \tag{VI.4}$$

where, in our case, the average load miss latency for a particular load and the average number of cycles spent in the trace are calculated by sampling hardware counters. With this the stride based prefetch instruction described in statement (VI.1) becomes:

$$prefetch \qquad (offset + (stride * distance))(base) \tag{VI.5}$$

We provide results for this approach, where we calculate a fixed prefetch distance for a load by using average load miss latency and the average cycles per loop iteration for a trace. Most prior prefetching systems keep the prefetch distance fixed like this after it is determined either statically or dynamically, and do not provide a mechanism to later tune this distance. A primary contribution of our paper is the ability to adapt this distance (as well as the stride) – not only allowing us to get it right more often, but also allowing us to further adapt if the nature of the load changes, which we describe next.

VI.E.2 Adaptive discovery of prefetching distance

The above prefetching distance estimation gives us a good starting point to initiate prefetching. The problem is that as you insert prefetches, even for the prefetched load, the recurrence time between instances of that load will change; that is, the iteration time used to calculate the prefetch distance is no longer correct. This problem is exacerbated by neighboring loads that are subsequently prefetched. Each successful optimization may expose other loads that were previously being prefetched on time.

Due to the heavy interaction between neighboring loads and the correct prefetch distance for each, we found that careful estimation of the correct distance was of little use, and a much simpler scheme provided equivalent performance.

Our Adaptive prefetching algorithm works as follows.

- All stride based prefetch instructions for delinquent loads are inserted in the hot trace as in statement (VI.5) with the initial distance of 1.
- We continue monitoring the behavior of these loads in the DLT. If the prefetch is not hiding enough latency, the load will eventually be marked again as a delinquent load and cause another delinquent load event.
- If the delinquent load is stride predictable and there exists a prefetch instruction for it, the optimizer increases or decrements the distance stored in the instruction as outlined in the next section, and we patch the prefetch instruction in the trace. The prior distance can be back calculated by using the predicted stride and the known offset, or using book-keeping information stored along with the trace.

The above optimization is done by the helper thread. Note that the repairing is easy and fast, since we do not generate a new trace or change the layout of an existing trace. We just update the prefetch instruction bits with the

new distance. This process is repeated until the prefetch distance causes the load to stop triggering delinquent load events, or the load becomes *mature*, which we describe later.

This approach works very well, especially when there are potentially multiple delinquent loads in a hot trace. Each load will have its prefetch distance adjusted until the loads in the trace are no longer delinquent. VI.2. As each load is prefetched more effectively, neighboring loads that then become exposed because the code runs faster will generate another delinquent load event, and be repaired. Stabilization is achieved quickly because the repair operation is much quicker than generating a new prefetch-optimized hot trace.

We also modeled a scheme where the initial distance is set to the estimated distance from the previous section, but saw no gain because the low overhead of the optimization system allows it to converge quickly.

VI.E.3 Prefetch Maturing

When a load triggers the delinquent load event for the first time, the optimizer inserts a prefetch instruction to target this delinquent load. Any subsequent delinquent load events for this load will cause its prefetch instruction to be repaired by the optimizer. Note if a delinquent load cannot be prefetched, as described in Section VI.D.1, or it cannot be repaired due to lack of stride patterns, the optimizer sets its mature flag in the DLT, so it will not cause a delinquent event, until the mature flag is cleared. For our experiments, the only way the mature flag is cleared is when a load is replaced due to capacity constraints in the table. Future work may want to examine clearing the mature flag when there is a working set or phase change in the program's execution to potentially capture new behavior [123].

For each of the repairable delinquent loads in the trace, the optimizer

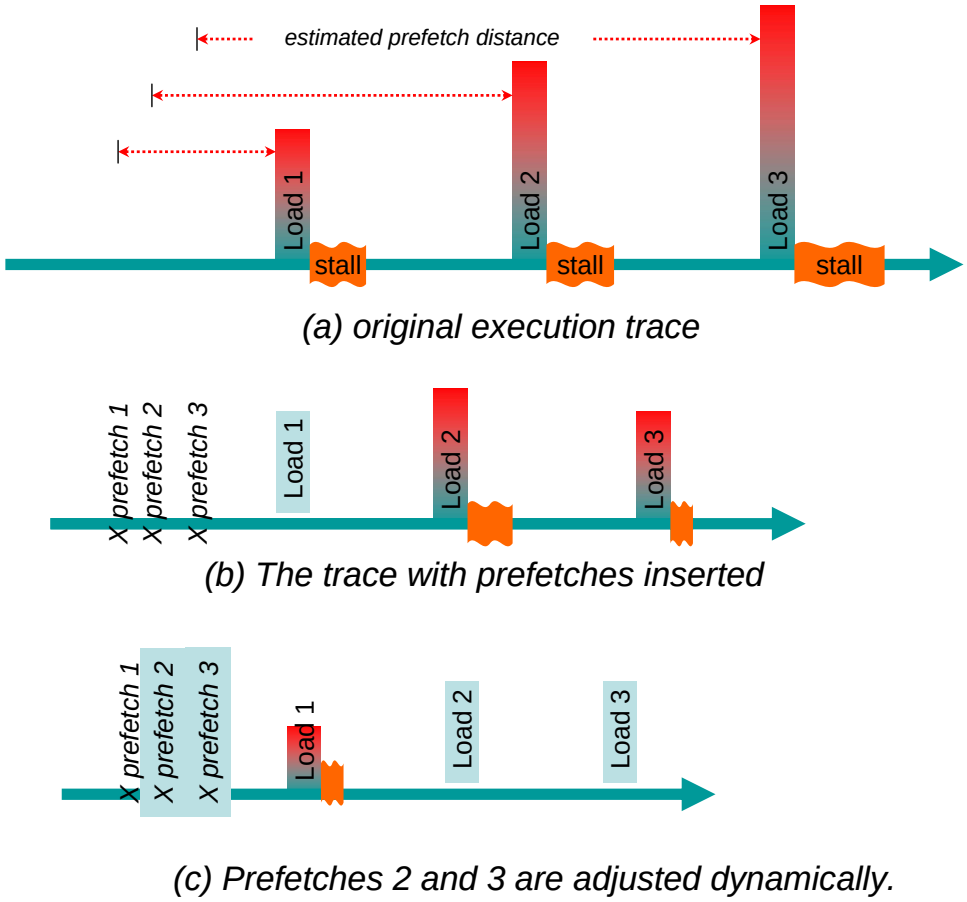

Figure VI.2: Trident's adaptive discovery of optimal prefetch distances. In this example, we assume there are three delinquent loads, which stall the program's execution, in the original trace, as shown in (a). Software prefetches are inserted with estimated prefetch distances. These prefetches make the total execution time of the trace shorter than before. However, since each prefetch distance is estimated independently, these distances are not far enough to hide all load latencies. In this example, load 2 and load 3 are still delinquent, as shown in (b). Trident then adjusts these prefetches (2 and 3) to hide latencies from loads 2 and 3. The adjustment further reduces the total execution time. Because of this, load 1 becomes delinquent again. This triggers Trident to adjust the prefetch instruction for load 1. However, other loads may become delinquent after this adjustment. This adjustment process continues until all loads are stabilized or matured.

re-calculates its maximal prefetch distance. The maximum is the memory access latency divided by the trace's minimal execution time from the watch table. Here, the minimal execution time of the trace should represent the best possible scenario where all loads in the trace potentially hit in the cache. When executing an optimized trace, the number of cycles from when the trace is first fetched until it finishes execution represents the time to execute the trace, and the watch table keeps the minimum number of cycles seen for each trace currently being used.

When a prefetch instruction is repaired, the optimizer increases the load's prefetch distance by 1 up to its maximal distance. Increasing the prefetch distance allows the prefetch to happen further ahead of the potential use of the data, which will hopefully reduce the load miss latency. Thus we expect the average access latency for the load to decrease when we increase the prefetch distance.

However, as the prefetch distance increases, the possibility of prefetched data being replaced by data from other loads/prefetches also increases. If this occurs, the load's average access latency may instead increase. We therefore calculate the average access latency when repairing a prefetch, and when it is observed to start to increase the optimizer decrements the distance by one. To do this calculation, the average access latency is computed using the load's access counter, miss counter, and total miss latency from the DLT table. In addition, we store in an optimization buffer in the program's memory the load's previous average access latency.

Therefore, our repairing mechanism varies a load's prefetch distance from one to its maximal distance, trying to find an optimal distance. To avoid a load being repaired too many times, the optimizer sets the load's mature flag in the DLT when the number of repairs attempted is twice as many as its maximal (distance) value. When a load is first optimized, we set a repair counter for the

load to this number. Each time a load is repaired the counter is decremented. When the counter is zero, then we no longer try to repair the load, and the mature flag is set in the DLT.

Note, in order to make the above decisions, the optimizer always maintains relevant information from all delinquent loads, such as the number of repairs left, the maximal distance, and the average access latency history. This is stored in a memory buffer used by the optimizer. This information could alternatively be stored in the DLT.

VI.F Methodology

To evaluate the performance of our self-repairing software prefetching technique, we run the runtime optimizer code of *adaptive dynamic inlined prefetching*, concurrently with the applications on a simulated multithreaded processor.

VI.F.1 Baseline Processor Architecture

Our baseline architecture is simulated as a 20-staged simultaneous multithreading (SMT) processor [133, 132] with 2 hardware contexts. The baseline configuration is shown in Table VI.2.

Performance is evaluated using the SMT processor simulator [133], modified to model the Trident hardware and runtime infrastructure. The simulator also models memory timing and bus occupancy among different memory hierarchies. It simulates the actual execution of the main thread, running concurrently with the optimizing helper threads as they modify the executable, place traces in the code cache, and patch the main thread to begin using the new traces. Significant care is taken to insure that instruction throughput (IPC) results correspond to only the number of instructions the *original* code would have executed.

Table VI.2: The configuration of the baseline SMT processor with hardware stream prefetching. There are total 8 stream buffers, and each buffer can hold up to 8 fetch blocks. The prefetching is guided by a two-delta stride predictor.

Pipeline	20-stage, 256-entry ROB, 224 registers
	Two hardware contexts
Queue Sizes	64 entries each IQ, FQ, and MQ
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. History table 1024 entries. Prefetching is guided by a stride predictor.

Since modern processors often include a hardware prefetching mechanism, we implement a reasonably aggressive hardware stream buffer prefetcher [125] in our baseline architecture. The stream buffers are guided by a stride predictor, and buffers are allocated using a confidence scheme. We simulate two stream buffer configurations: (1) 4 stream buffers and each buffer has 4 entries, (2) 8 stream buffers and each has 8 entries. As shown in Figure VI.3, the 4X4 configuration achieves an average 35% speedup relative to no prefetching, and the 8X8 configuration, 40%. We therefore choose the hardware stream buffers of the 8X8 configuration as our baseline, which is used to evaluate the relative performance of software prefetching in the next section.

Note that in our current study, software prefetching works independently of the underlying hardware prefetching mechanism. Because it focuses on loads that actually miss, it will naturally adapt to the loads that the hardware prefetcher cannot handle. With this system, the compiler need not know what, if any, hardware prefetcher is active.

VI.F.2 Benchmarks

Performance is evaluated using SPEC 2000 (integer and FP) benchmarks and a few pointer intensive applications from prior research. We selected the top 14 benchmarks with the longest average miss latencies for our study. These include *applu, art, dot, equake, facerec, fma3d, galgel, gap, mcf, mgrid, parser, swim, vis,* and *wupwise*. All benchmarks are compiled on the Alpha platform (Digital Unix V4.0F) with the highest optimization options. Each benchmark is simulated for 100 million instructions beyond the single simulation points from SimPoint [122] except *dot* and *vis*, which both are fast forwarded 5 billion instructions. The simulator is warmed up with 5 million instructions.

Figure VI.3 shows the base performance of each benchmark when exe-

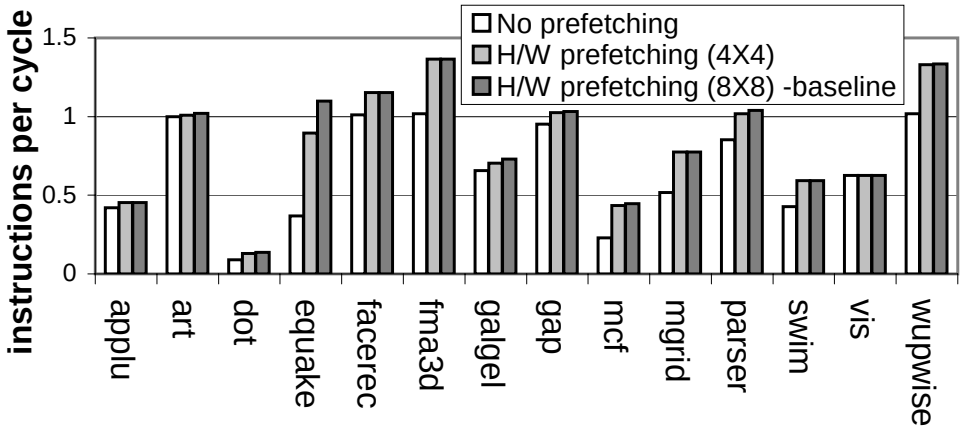

Figure VI.3: Performance on the baseline SMT processor with hardware stream prefetching enabled. The default stream prefetching mechanism has 8 stream buffers. Each buffer can hold up to 8 fetch blocks.

cuted alone on the baseline architecture. The performance from our baseline (the 8X8 hardware prefetching) is used for future performance comparison.

VI.F.3 Prefetching via Trident Architecture

The goal of this research is to use dynamic code optimization to improve the performance of the memory subsystem. We use the event-driven multithreaded dynamic optimization framework (Trident) to generate base optimized hot traces. Upon delinquent load events, base-optimized hot traces are re-optimized to insert software prefetching instructions to target frequent cache-missing loads.

The runtime optimization code performs optimizations on the streamlined instruction traces. Optimizations include forming hot traces with base optimizations as outlined in Section VI.B, inserting software prefetching instructions into hot traces, and repairing prefetching as needed. The runtime code executed in Trident is written in *C* and compiled with *gcc -O5* on the Alpha platform. The

Table VI.3: Trident hardware monitoring structures. The DLT table detects address stride values for the delinquent loads. A load is delinquent if its miss rate is higher than 3% and its average miss latency is higher than 1.5 times the L1 miss penalty.

Branch profiler	256-entry, 4-way associative. Each entry has a 4-bit counter. Three standalone 16-bit bitmaps
Watch table	256-entry. Each entry monitors current trace's minimal execution time.
Delinquent Load Table	2-way associative; total 1024 entries. Access counter threshold: 256 (8 bits) Miss counter threshold: 8 Each entry keeps track of the load's accesses, misses, miss latency, last address, and its stride.

helper thread startup latency is assumed 2000 cycles.

Monitoring Hardware Trident uses a few small hardware structures to monitor the program's execution. These hardware structures can generate hot events upon detection of certain program behaviors, and trigger Trident to perform dynamic optimization. The configurations of the major hardware structures – the branch profiler, the hot trace watch table, and the delinquent load table – are shown in Table VI.3. The default DLT table has the access counter threshold of 256 and the miss counter threshold of 8, which approximates a cache miss threshold of 3% for delinquent loads.

VI.G Performance

In this section, we evaluate the costs, effectiveness, and performance of our dynamic prefetching technique. Performance improvement is relative to the baseline architecture, whose performance is shown in Figure VI.3.

VI.G.1 Overhead of the Dynamic Prefetch Optimizer

Our adaptive, event-driven prefetching approach has costs that traditional hardware techniques do not incur, which is the cost of generating the prefetch code at runtime. If this cost is high, it can negate the performance gains of prefetching. However, our approach keeps this cost low, because we never interrupt the main thread to run the optimizer, we run the optimizer at a lower execution priority on a spare hardware context, and the optimization thread tends to have low execution resource demands.

To measure the cost of our dynamic prefetch optimizer to see how much it affects the performance of the main execution thread, we run Trident with our prefetch optimization without actually using the optimized traces. That is, the runtime optimizer is triggered to construct and optimize hot traces, but it does not alter the original binary to jump to the optimized trace. The goal of this analysis is to determine the overhead the optimization code imposes on the system while it executes concurrently with the main thread.

We observe the total cost to be only 0.6%. This is much lower than what would be expected in a traditional dynamic optimization system such as Dynamo [7], which would require runtime profiling and frequent switches between the main thread and the optimizer and profiler to enable a similar optimization.

In our current study, an idle hardware context is assumed ready for dynamic optimization. However, we do not have to reserve an entire hardware context for dynamic optimization. Helper threads are invoked for optimization

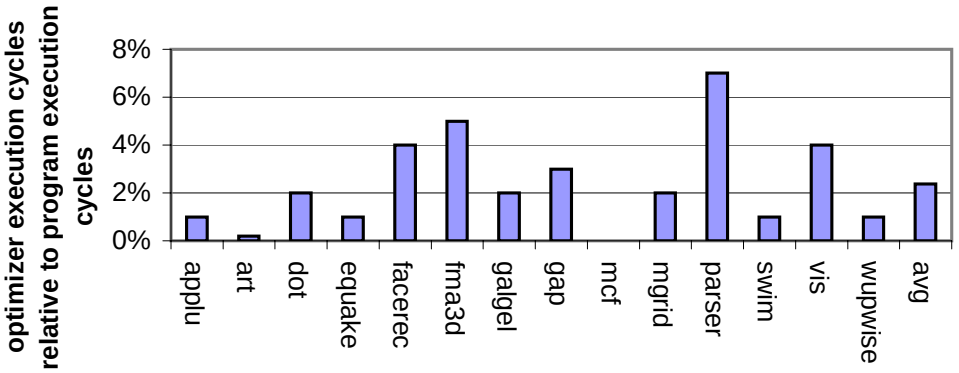

Figure VI.4: The execution time of optimization threads relative to the main program's execution time. The execution time does not include the time of repetitious discovery of prefetch distances.

subject to the availability of hardware contexts. Hardware contexts are released after helper threads finish the optimization. Figure VI.4 shows the percentage of each benchmark's total execution cycles when the optimization thread runs concurrently with the benchmark. The results show that the helper threads are active for a small fraction of the main thread's total execution time, on average 2.2%. Since the optimization time is relatively small, our optimization technique should have opportunities to run even if contexts only become available during I/O.

Note that the cost of our optimizations will increase with our adaptive techniques; however, the optimization threads with self repairing prefetching are typically active at most 25% more than the base case. Therefore, the total cost is still under 1%. The cost of the interference between the main thread and the helper threads are fully reflected in subsequent results.

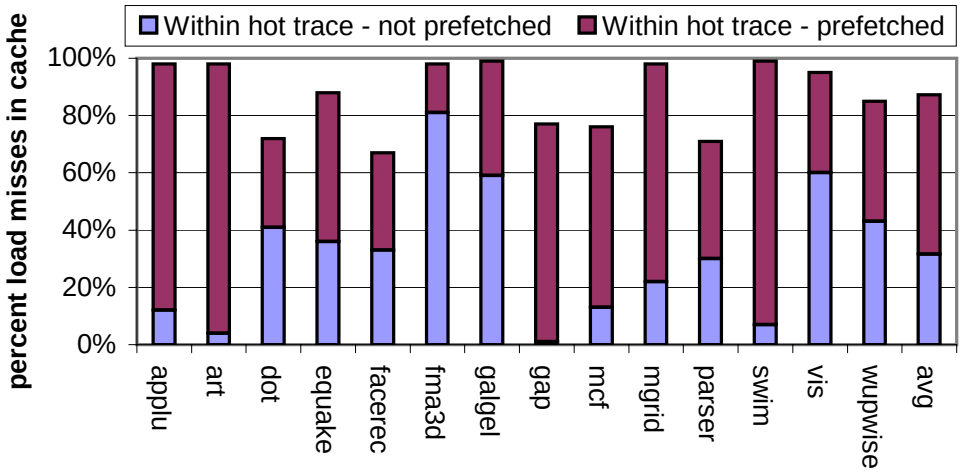

Figure VI.5: Percentage of load missed covered by hot traces and the prefetcher. The difference between the height of the bar and 100% indicates the percentage of cache misses that occur when executing the non-traced code.

VI.G.2 Load Coverage by Software Prefetching

To gauge the potential of our software prefetching, we first measure the dynamic load miss coverage. Only load instructions within hot traces can be potentially prefetched by our current optimization technique. Figure VI.5 shows the percent of cache misses which occur within hot traces, and those that can be potentially software prefetched. The difference between the height of the bar and 100% represents cache misses that do not occur while executing hot traces.

The results show that our hot trace scheme covers over 85% of load misses, and nearly 55% of all misses are potentially covered by our prefetcher. Note that *dot*, *facerec*, and *parser* have relatively low miss coverage. This is in large part because of the low dynamic coverage of the hot traces. In contrast, *gap* has low hot trace coverage, but nearly all its hot trace load misses are prefetched.

VI.G.3 Performance of Software Prefetching

In this section, we want to show the performance improvement from basic software prefetching, whole object prefetching, and our adaptive self-repairing prefetcher. The performance improvement over the baseline hardware prefetching is shown in Figure VI.6.

The first bar (*basic*) shows the performance improvement with a configuration similar to prior dynamic prefetching schemes [85, 25]. This divides the average cache miss latency by the average trace execution time to estimate the prefetching distance. We refer to this scheme as the baseline software prefetching approach (even though it includes some features unique to our system, such as strided prefetching of pointer loads). As shown in Figure VI.6, the average speedup for the baseline software prefetching technique is about 11% (over the baseline – hardware prefetching alone). This means the prefetch distance estimation works reasonably well.

The second bar (*whole object*) represents the performance improvement of the stride-based same object prefetching over the baseline hardware prefetching. It achieves higher performance improvement than the baseline scheme on the pointer intensive applications such as *dot* and *mcf*. Performance improvement is mainly due to the jump-pointer type prefetching.

The third bar in the figure shows the performance of software prefetching with our adaptive self-repair technique. In this approach, the baseline software prefetcher is initiated first, with a default prefetch distance of 1. Its prefetch distance is gradually repaired by the runtime optimizer. As observed in Figure VI.6, software prefetching with self-repairing significantly boosts the prefetching performance. This is because our adaptive prefetching technique can dynamically correct the prefetch distance as the latency of the hot trace is dynamically tuned. Note that *applu*, *facerec*, and *fma3d* do not show any further performance im-

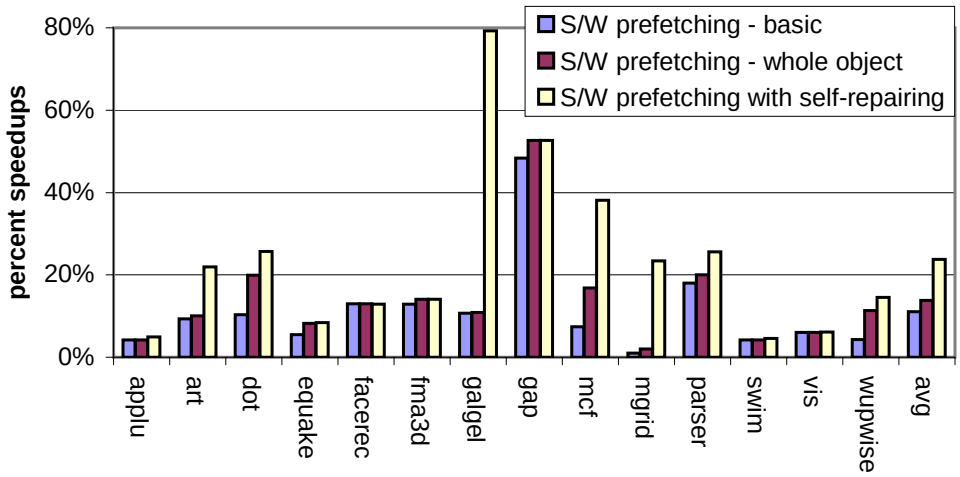

Figure VI.6: Performance improvement of software prefetching with and without self-repairing relative to hardware prefetching (8X8). The basic bar represents the performance from current dynamic optimization systems (e.g. ADORE). The whole-object bar represents the improvement if we add the same object based prefetching. The last bar is the performance from our adaptive dynamic prefetching.

provement with self-repairing, because the naive estimates were sufficient – for example, *applu* has such a large inner loop (over 1000 instructions) that a prefetch distance of 1 is optimal.

As noted previously, we also examine an alternate strategy where the initial prefetch distance is estimated more carefully and repaired/incremented from there. We found it achieves performance almost identical to the results shown here. This demonstrates the efficiency with which the system adapts, as the initial value becomes irrelevant. The simpler scheme eliminates certain hardware overheads (also necessary for the non-adaptive results shown) needed to estimate the initial prefetch distance.

Overall, the self-repairing prefetcher outperforms the basic software prefetcher by increasing the speedup from 11% to 23% on average. This comes from a combination of (1) doing a better job of getting the prefetch distance right, and (2) adapting to changes in the hot trace and cache behavior. This demonstrates that our low-overhead, adaptive prefetch approach enables us to overcome the difficulty in calculating statically, or even at runtime, the appropriate prefetch distance.

More insight into our software prefetching approach is provided by Figure VI.7. Each bar represents the percentage breakdown of all dynamic loads which hit, partially hit, or miss in the cache. When a cache block is fetched due to a prefetch instruction, the first load access to this block is counted as a *Hit-prefetched*, but any subsequent accesses are counted as *Hits-none* rather than prefetching hits. When a cache line is displaced by a prefetch, we record the line tag so that we can identify the *Miss due to prefetching* if a subsequent cache miss matches that tag. Figure VI.7 shows two key results that indicate the power of our adaptively repairing prefetcher. Misses due to prefetching rarely occur, and we have a very low incidence of partial prefetch hits.

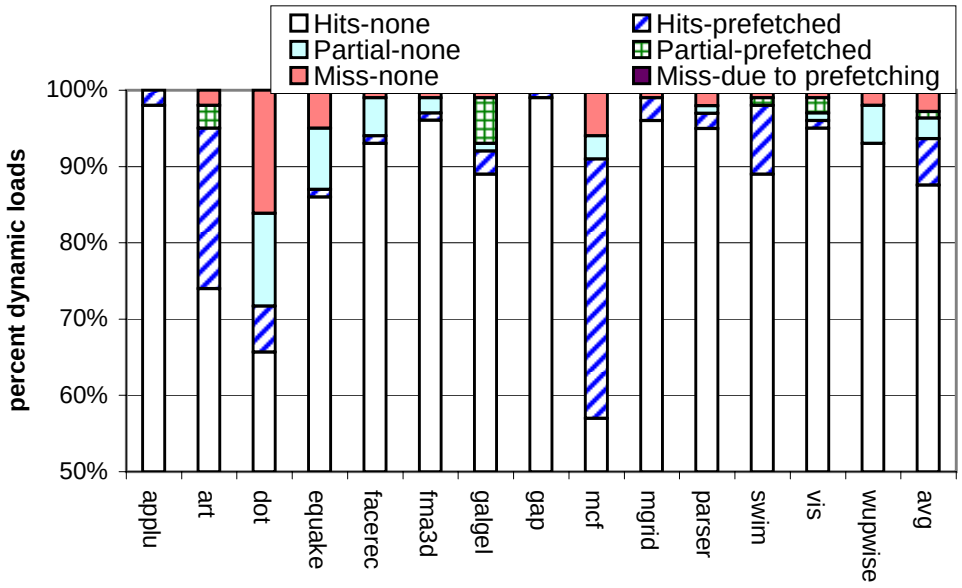

Figure VI.7: Percentage breakdown of all dynamic loads. The "hits-none" represents the normal cache hits. The "hits-prefetched" represents the cache hit due to software prefetching. A prefetch hit is only counted once. Any subsequent access to the prefetched cache block is classified as "hits-none". The "miss-none" indicates the normal cache misses. The "miss-due to prefetching" represents the side effect of software prefetching, which are cache misses caused by a prefetched block replacing a block that would have gotten a hit in the future.

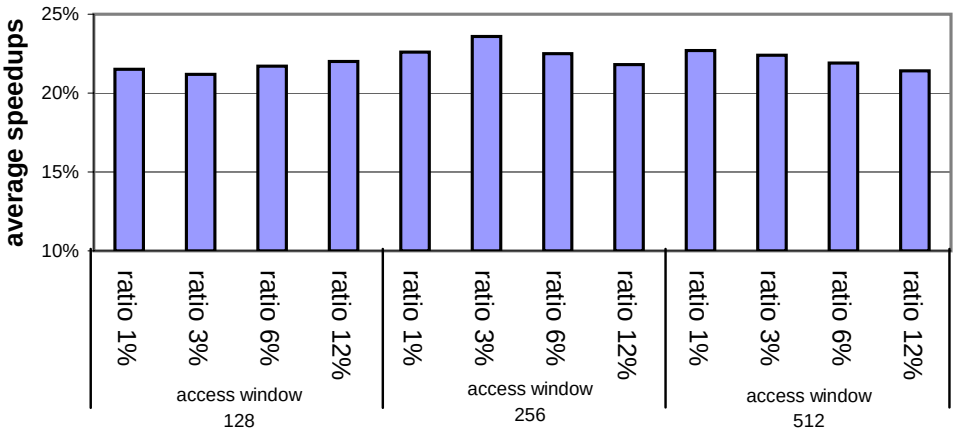

Figure VI.8: Average performance improvement of software prefetching with different load monitoring window sizes and cache miss rate thresholds. The miss rate is calculated as the miss count divided by the access count during a given monitoring window.

VI.G.4 Software Prefetching Sensitivity

This section shows the sensitivity of our self-repairing prefetcher to the DLT sizes and our delinquent-load identifying thresholds. We show the results for three load monitoring window sizes (128, 256, and 512). We also show results for miss rate thresholds of 1%, 3%, 6%, and 12%. This is the miss rate that needs to occur within the load monitoring window to classify the load as a delinquent load. Figure VI.8 shows the average performance improvement of software prefetching for these different configurations. We found that at least 8 misses during the load's monitoring window provides an adequate indication to classify the load as delinquent. If this number is too small, then it may be overly aggressive with its prefetching. On the other hand, if this number is too big, it may miss delinquent loads. Overall, a cache miss rate threshold of 3% (at least 8 misses out of 256 accesses) works best for the program's we examined.

Figure VI.9 shows the performance improvement of software prefetching

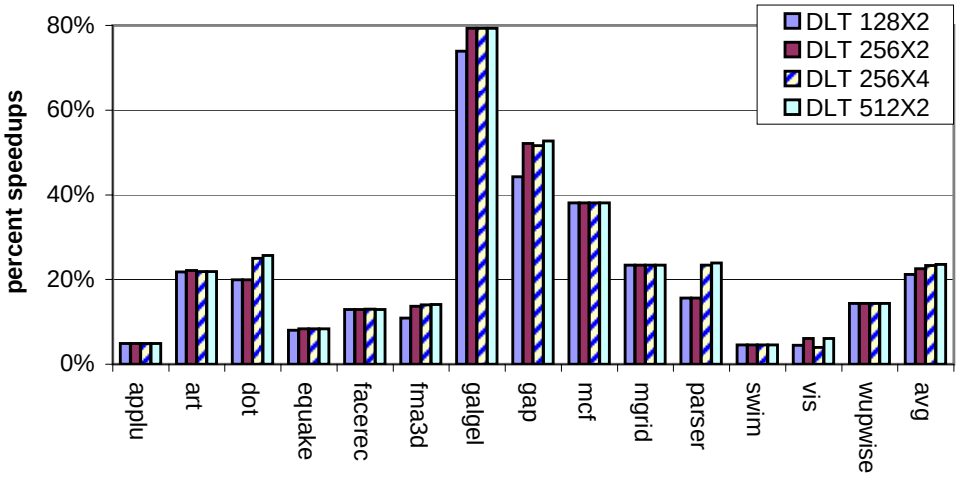

Figure VI.9: Average performance improvement of software prefetching with different DLT sizes. The DLT is organized as a set-associative cache. Each configuration represents the number of sets and the cache associativity.

with different delinquent load table (DLT) sizes. We found for most programs that the performance only slightly increases when the table size doubles. However, for benchmarks with large working sets, such as *dot* and *parser*, performance is boosted with a large DLT size. We anticipate the DLT with 1024 entries should work well for most programs.

Since our prefetching technique relies on some new hardware structures, we also want to evaluate how much these hardware resources would boost performance if we simply used them to increase the size of the data cache. We estimate all hardware resources used in the DLT table and the watch table are about 16KB. If these hardware resources are used to increase the size of the L1 data cache, we can essentially increase the data cache from 2 ways to 3 ways. However, by increasing the data cache size, we observe merely a 0.8% performance boost over the baseline.

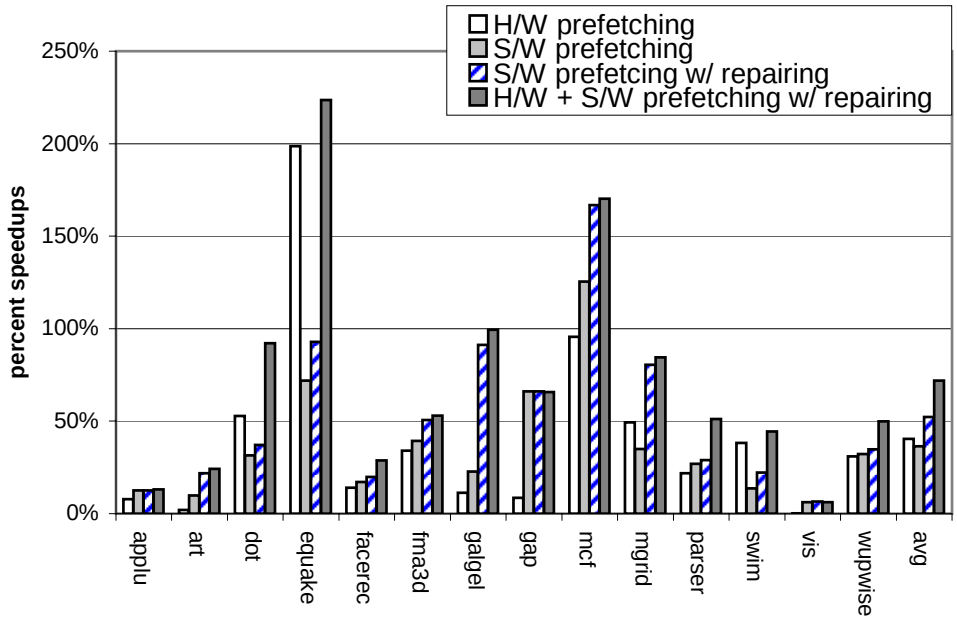

Figure VI.10: Performance comparison with hardware prefetching. The second bar represents the performance when applying the existing dynamic prefetching system alone. The third bar stands for the speedup when using our adaptive dynamic prefetching alone. The last bar is for hardware stream prefetching combined with our adaptive dynamic software prefetching.

VI.G.5 Comparison with Hardware Prefetching

Finally, we compare the performance of software prefetching and hardware prefetching alone in Figure VI.10. Recall that all previous results include hardware prefetching in the default baseline. Relative to the baseline *without* any prefetching, our software prefetching with self-repairing outperforms hardware prefetching (*the 8X8 configuration*) in most benchmarks, by an average 11% increase in performance. We notice that software prefetching achieves relatively moderate speedups for *dot*, *equake*, and *swim*. This is due to two factors: (1) Our dynamic software prefetching only targets delinquent loads within hot traces. Thus, low coverage of dynamic loads, such as in *dot*, limits software prefetching performance. (2) Software prefetching has cost. Prefetching instructions are fetched, issued, and executed along with other *regular* instructions from the application. They reduce the *effective* instruction fetch/issue bandwidths because fewer regular instructions are fetched and issued. At the same time, prefetching instructions take away execution resources (e.g., ALUs or load/store queue slots) from regular instructions. Finally, the prefetched data may replace the existing data in the cache due to cache capacity conflict. Thus, when the programs such as *equake* and *swim* exhibit simple stride patterns with short prefetching distances, hardware prefetching may be more advantageous. However, when software prefetching is combined with hardware prefetching, the cost is offset since software prefetching now targets delinquent loads which cannot be handled efficiently by hardware prefetching.

VI.H Summary

Software prefetching is a promising technique to tolerate long memory latencies and achieve full performance from modern processors. Software

prefetching has to be accurate and timely in order to be effective.

In this chapter, we extend the event-driven, multithreaded dynamic optimization framework, *Trident*, to perform software prefetching by dynamically inserting prefetch instructions into hot traces. The low overhead of the Trident framework allows the runtime optimizer to repeatedly optimize the same trace to adjust prefetching either because existing prefetching is not effective or because the program's behavior changes. This is done both through runtime estimation of loop timing, and through progressive updating and evaluation of the prefetch distance. Thus, inserted software prefetches are re-evaluated, adjusted, or removed according to the runtime behavior.

Our prefetching technique also performs stride-based object prefetching to not only hide the latency of the first access of the object, but all of the fields touched in that object. Whole object prefetching identifies all of the accesses to the different fields of an object in a hot trace. We then insert the minimum amount of software prefetches to prefetch all of the parts of the object that will be used. This technique combines the effectiveness of software prefetching, which can analyze the code to recognize access patterns, with many of the advantages of hardware prefetching, which can exploit some patterns static software systems cannot, and which can adapt to the actual runtime behavior of individual loads.

With our dynamic self-repairing prefetcher, which finds the proper prefetch distance by trying multiple distances until the correct one is found, we achieve an average 23% speedup relative to the baseline which includes a hardware stride based prefetcher. In addition, our self-repairing prefetching mechanism achieves 12% better performance than prior dynamic prefetching techniques without repairing.

Acknowledgment

The text in this chapter is in part a reprint of the material from the paper, *Weifeng Zhang, Brad Calder, and D.M. Tullsen, "A Self-Repairing Prefetcher in an Event-Driven Dynamic Optimization Framework"*, in the proceedings of the 4th International Symposium on Code Generation and Optimization (CGO 2006), March 2006, New York, NY. The dissertation author was the primary researcher and author and the co-authors involved in the publication [143] directed, supervised, and assisted in the research which forms the basis for that material.

VII

Accelerating Precomputation Based Prefetching

In this chapter, we exploit the Trident system to dynamically construct precomputation threads and optimize them for efficient prefetching.

Precomputation based prefetching [36, 116, 87, 71, 86] is an alternative approach to inlined prefetching to attack the memory wall problem. It takes advantage of the modern processor's on-chip parallelism by running a short version of the main thread code on an otherwise idle hardware context. The precomputation code may issue and execute the cache-missing loads earlier than the main thread due to code simplification. By the time the main thread executes the same loads, the data may already exist in the cache. Thus, the precomputation thread absorbs the cache miss penalty on behalf of the main thread.

As discussed in Chapter I.B.2, effective prefetching should meet these three criteria: prefetching accuracy, timeliness, and low overhead for the prefetching address computation. Precomputation based prefetching has very high accuracy because prefetching addresses are actually calculated using the same instructions as the main thread. Because of this, however, precomputation can suffer

relatively higher runtime overhead (in the number of instructions) than prediction based prefetching, which typically takes one cycle to predict addresses. But this overhead is minimized because the instructions are executed in a separate thread. Compared with inlined prefetching, precomputation based prefetching has the potential to handle more complex address patterns (e.g., pointer chasing), which are hard to predict. But complex load behavior can prevent the precomputation thread from running sufficiently ahead of the main thread. In addition, because address precomputation is decoupled from the main thread, the prefetching address stream may diverge from the main thread, if the prefetcher is based on control flow or address speculation. Runaway prefetching reduces prefetching efficiency, or worse, results in more data cache misses in the main thread.

In this chapter, we exploit the following techniques to enable efficient precomputation based prefetching. These techniques reduce precomputation overhead and improve precomputation execution for timeliness.

- The precomputation thread always runs at low priority. Instructions from the precomputation thread are fetched and executed only when the main thread cannot utilize the full fetching and execution bandwidths. Low priority prevents the helper thread from competing for execution resources from the main thread. This technique helps reduce the precomputation overhead.
- Hardware prediction is used along with induction variable analysis on the precomputation code to jump start precomputation threads several loop iterations ahead of the main thread of execution. We leverage the hardware monitoring mechanism to predict load patterns. We use the predicted values to simplify/specialize the precomputation thread.
- A lightweight software mechanism for prefetching address coherency is exploited to detect and recover runaway prefetching. In this mechanism, co-

herency checking is completely done by the precomputation thread, with no overhead to the main thread.

We propose to embed precomputation based prefetching into the Trident dynamic optimization framework. We dynamically construct precomputation code (called *p-slices*) from the main thread's hot execution traces. Special instructions are inserted into the hot trace to automatically trigger/spawn the p-slice to run in an idle hardware thread (called *p-thread*), and to terminate the p-thread when the main thread exits the hot trace. The p-thread is synchronized with the main thread using the lightweight address coherence mechanism.

VII.A Related Work

This section summarizes prior research on precomputation based prefetching. Research exploiting helper threads for other functions is briefly mentioned in Section II.E.

VII.A.1 Prefetching via Precomputation

Precomputation threads typically run in a separate thread [36, 116, 146, 35, 50], or in a dedicated hardware engine [94, 5, 114], concurrently with the main thread.

Zilles and Sohi [146] manually construct speculative slices for pre-execution to target problem instructions (cache misses or branch mispredictions). They also propose a technique to bind branch prediction generated by pre-execution to correct branch instances. Roth and Sohi [116] propose the speculative data-driven multithreading (DDMT) architecture to speculatively execute future cache miss instructions. Helper threads are statically constructed via offline analysis.

Collins, et al. [36] dynamically identify delinquent loads and store backward-slice instructions in hardware. They use a chaining prefetching scheme to allow

a single thread to loop through multiple prefetches of the same load so that the p-thread may get sufficiently ahead of the main thread. Those systems use the extra thread contexts to run prefetch code without modifying the original code.

More recent work by Lu et al. [86] dynamically constructs p-slices via a runtime optimizer running on an idle core. Prefetching is performed by piggy backing a single user-level p-slice construction thread. That is, this single thread is multi-tasked to do profiling, phase detection, p-slice construction, and execution of p-slice for prefetching. Our research is similar to this work in terms of dynamic p-slice construction. The main differences are: (1) Our prefetching p-threads and the p-slice construction thread run concurrently by taking advantage of the processor's on-chip parallelism. So prefetching can be performed while a new p-slice is constructed. (2) More importantly, our research focuses on how to accelerate p-threads for more efficient prefetching. We use loop induction variable analysis to allow the p-threads to be started several loop iterations ahead of the main thread.

Precomputation threads can also run in the specialized hardware engines. Annavaram, at al [5] propose the dependence graph precomputation (*DGP*) scheme to prefetch irregular loads. The DGP scheme dynamically uncovers the prefetching slice for cache miss instructions. Whenever the pre-decode stage detects the load/store instruction that is marked for prefetching, it automatically derives the dependence graph to precompute the prefetch address. This is done by chasing through all instructions currently in the instruction fetch queue to build up register dependencies. The precomputation graph is then executed on a specialized engine to do prefetching.

Slice processors [94] uses a hardware structure, called the Slicer, to construct a p-thread in the commit stage instead of the fetch stage. The processor stores the p-threads in the slice cache, and the p-thread is initiated upon detecting

a trigger instruction in the main program flow.

Hardware based precomputation for prefetching often imposes significant hardware complexity, and it is challenging to identify proper trigger points to start precomputation threads early.

It is also interesting to note that precomputation based prefetching is a completely different mechanism from *Run-ahead Execution Prefetching* (REF) [24]. On a processor with REF, when the thread stalls due to a long latency operation (i.e. a cache miss), the processor check-points its current state, and then continues execution speculatively. The speculative execution may trigger more cache misses down the execution stream, thus overlapping these load misses. When the initial cache miss is served, the thread resumes its execution after its state is restored from the checkpoint. Therefore, the thread running on the REF processor should encounter fewer cache misses than on the processor without REF, due to prefetching from run-ahead execution. However, the REF approach cannot prefetch dependent loads. A detailed comparison between our approach and REF is left for future research.

VII.A.2 Static Precomputation Construction

Prior research [37, 87, 71] has demonstrated the ability to construct p-slices statically.

Collins, et al. [37] statically construct precomputation slices from instruction streams within a small window to prefetch delinquent loads. The constructed p-slices are statically linked into the program's binary. Luk [87] 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 [71] to automatically generate p-slices at the high level language. Kim, et al. [70] also study how to reduce p-thread impact on the main thread's

performance on Pentium 4 processors. The activation of p-threads is dynamically throttled with lightweight hardware thread synchronization support. Their study focuses on reducing contention for the processor's execution resources from p-threads. Quinones, et al. [108] develop a compiler framework, called *Mitosis*, to generate speculative parallel threads. Mitosis adds a precomputation slice at the beginning of each speculative thread to compute thread input values to start speculative multithreaded execution. Liao, et al. [82] propose post-pass binary analyses to construct p-slices at the binary level, but these analyses are difficult to perform dynamically.

Rabbah, et al. [109] enable prefetching without resource competition by embedding the precomputation code into VLIW traces. This mechanism takes advantage of unused issue slots in the VLIW architecture. Prefetching is dropped when stalled so that it does not stall the main thread.

Ro and Gaudiot [111] statically annotate the instructions to be used in p-slices. The marked instructions are extracted by hardware at the front end of the pipeline to form the actual slices. This model does not duplicate the static code for p-slices, and therefore does not require additional fetch bandwidth. However, it needs a new design in the front end of the pipeline, which may impact the microarchitectural timing. It also uses a fixed distance to trigger precomputation threads.

Static construction of the precomputation code finds it difficult to determine prefetching latency accurately at the high level source code, and it often leads to inefficient prefetching due to its inability to adapt to the program's runtime load and control flow behavior. In addition, it does not support legacy code. Our work focuses on enabling 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 from p-threads that get off track.

VII.B Dynamic Precomputation Prefetching Architecture

In this section we describe how we form precomputation slices with support from the Trident dynamic optimization system, and then describe our techniques to improve speculative precomputation for efficient prefetching.

VII.B.1 Overview

In this research, we use Trident's dynamic optimizer to construct precomputation slices from the program's hot execution traces, and store these slices in the code cache, along with hot traces.

As described in previous chapters, hot traces contain instruction blocks that are frequently executed together. After being formed dynamically, these hot traces are monitored by hardware to detect *delinquent loads*, which frequently miss in the data cache. Upon detection of such loads, hardware triggers *delinquent load events* to spawn a software thread to perform optimization. Here, the optimization thread constructs a precomputation slice from the event-generating hot trace, to prefetch all delinquent loads that reside in the hot trace. Note, the current hot trace is also re-generated to insert triggers, which automatically spawn/kill the precomputation thread when the program's execution enters/exits the hot trace. Section VII.B.3 explains this in more details.

During the basic p-slice construction, we also optimize p-slices with a few techniques to jump start and accelerate their execution.

VII.B.2 Precomputation Code Construction

After a hot trace has been executed a number of times, a delinquent load event may be generated if the DLT detects a delinquent load within that hot trace, as described in Section VI.C. Then, the runtime optimizer is spawned to perform optimizations on this trace. In this research, the optimization is to construct a p-slice to prefetch delinquent loads within the trace if this trace has a self-loop. We do not include non-looped traces for p-slice construction because these traces have short execution times. Thus, the p-thread based on these traces will be spawned/killed too frequently. In addition, the p-thread may not be spawned early enough to be effective.

The goal of the 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 read from the DLT. Because in Trident there is a delay between the event and when the optimizer thread reads the DLT, it is common to have multiple loads tagged as delinquent by the time the trace is being optimized to insert prefetches. For each recorded load, the optimizer analyzes the hot trace, in the reverse order of execution from the load's current position, to build up a slice of instructions the load depends on, either directly or indirectly. This is called *back-slicing* [36]. 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.

The p-slice construction steps are similar to prior research [36, 86]. We also perform the following optimizations to improve the p-slice's quality for prefetching:

1. During back-slicing, we check if a local load matches any preceding local store, meaning that they access the same stack address. If a match is found, we convert them into a single MOVE instruction. Unmatched local loads, if included in the final p-slice, are hoisted outside the p-slice loop.

2. Any other loop-invariant load instructions are also hoisted out of the loop.

3. A hot trace may contain multiple copies of the same code due to loop unrolling done by the static compiler. We found that even unrolled loops frequently have induction variables whose strides are less than the cache line size. Thus, multiple copies of the same code on the slice may essentially fetch the same cache block. We perform loop *re-rolling* (i.e. removing the redundant loop copies) to reduce duplicated computation inside a p-slice.

4. During back-slicing, all delinquent loads belonging to the same object are grouped together. Then we perform same-object based prefetching as in Section VI.D.2 by clustering all prefetches falling into the same cache line into a single prefetch. This helps reduce any redundant prefetches.

5. We remove the control flow from p-slices to streamline the code. Note, the original hot trace may contain a single path, but multiple branches, which we do not include in the p-slice. Skipping control flow helps reduce the instruction count inside the p-slice. This optimization works for three reasons. First, the hot trace should represent the most common path. Second, the prefetching thread often continues to prefetch effectively even when control flow does not match the main thread exactly. Third, we include code in the trace that allows us to recover when the p-thread diverges too much from main thread, as discussed in Section VII.D. This provides insurance against divergent address streams, with a much lower overhead than including all the control flow.

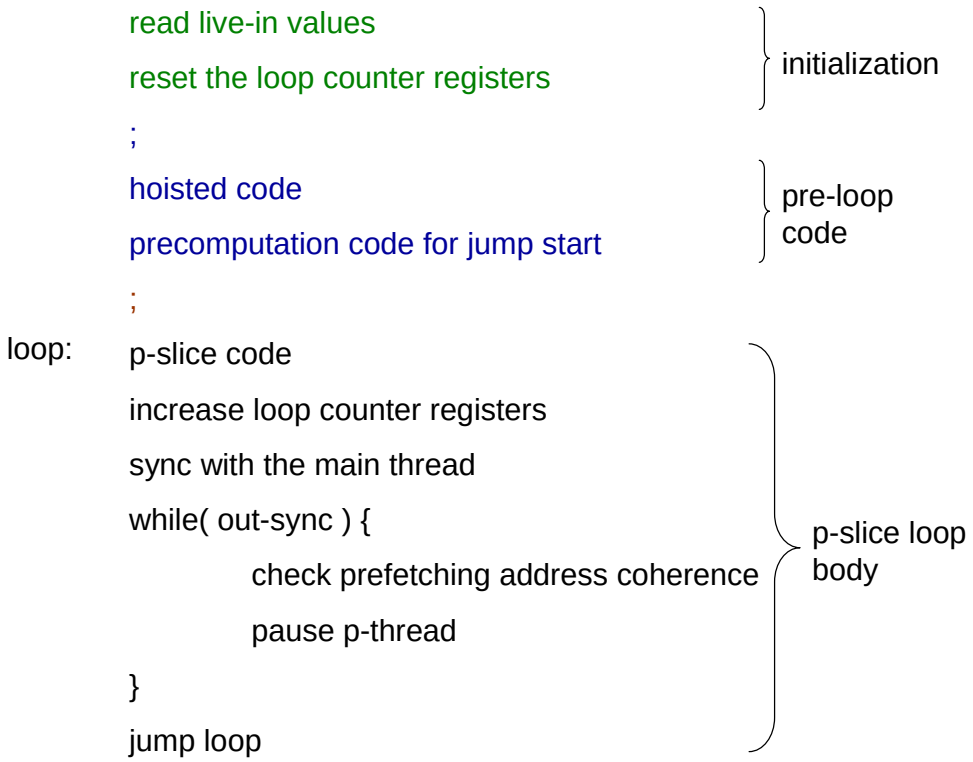

Figure VII.1: The layout of precomputation slices. The p-slice code can be divided into three portions. The first portion contains the p-slice initialization code. It reads live-in values from the main thread to start up the p-thread, and initializes the p-slice loop counter to keep in sync with the main thread. The second portion includes any loop-invariant computation code. We also insert code here to jump start the p-thread as needed. The last portion is the p-slice loop body. We augment the code here to enforce prefetching address coherency with the main thread.

During p-slice construction, the live-in values used to start the precomputation thread are also identified. Then, we lay out the p-slice code as shown in Figure VII.1. Details of the code are explained next.

VII.B.3 Precomputation Thread Triggering and Termination

To start up a p-thread, we need to communicate all live-in values from the main thread to the p-thread. One approach [36] is to use a hardware mechanism to perform a fast copy of register states. Another approach is to use a memory based mailbox mechanism [86]. In this paper, we devise a scalable live-in initialization mechanism. We assume a hardware value pipe (or queue) that the main thread writes to and the p-thread reads from. We need only one pipe per hardware thread.

To support this mechanism, we implement two primitives to read from and write to the value pipe. At the same time, we implement two additional primitives to spawn and kill p-threads. Some of these primitives may already exist in modern processors.

- **Read**: read a value from the pipe to a register. The reader stalls if the pipe is empty.
- **Write**: write a register value to the pipe. The writer stalls if the pipe if full.
- **Trigger**: spawn a p-thread.
- **Kill**: terminate a p-thread.

During the p-slice construction, the optimizer also identifies the precomputation spawning point and the termination point, and inserts software-based synchronization primitives into the hot trace. The hot trace in the main thread is re-generated with the layout shown in Figure VII.2. The original hot trace is wrapped with two pieces of code. The header piece writes live-in values to the

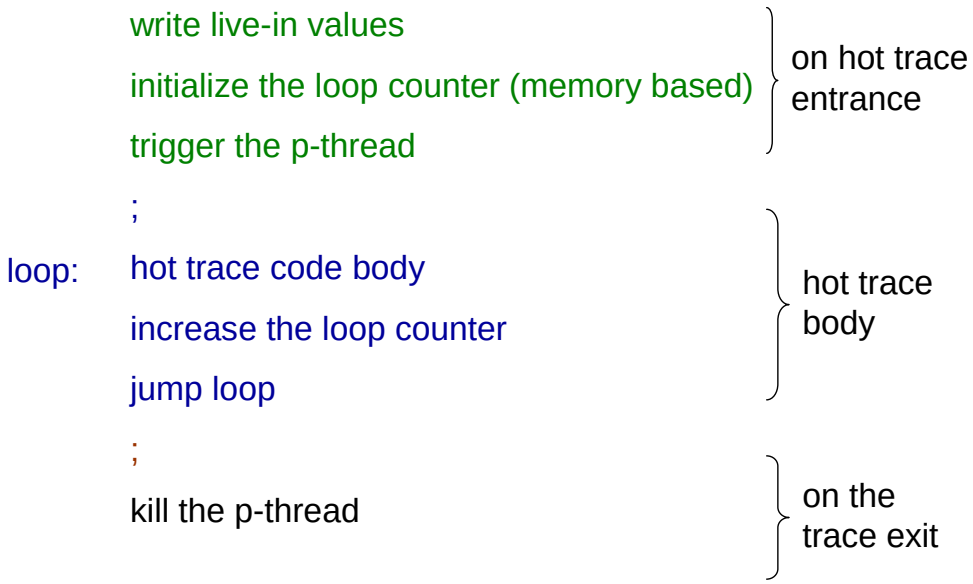

Figure VII.2: The hot trace layout to start up and terminate the p-thread. The original hot trace is wrapped with the p-thread startup and termination code. We introduce a memory based loop counter in the hot trace. The p-thread reads this counter from memory to synchronize itself with the main thread. Since the main thread does not have any data dependence on this counter, incrementing the counter every loop iteration should have minimal impact on the main thread.

value pipe for the p-thread to read. It also initializes a loop counter, which is stored in the memory, and triggers the p-thread. The other piece of code simply terminates the p-thread when the main thread leaves the hot execution region. Because these instructions (p-thread initialization and triggering) do not introduce any data dependency to the main thread, they should have little impact on the main thread.

We found that most p-threads only need one or two live-in values. In fact, we assumed the pipe is 5 values deep. At this size, we experienced no stalls because the pipe was full.

VII.B.4 Precomputation Thread Priority

To minimize any negative performance impact on the main thread, a p-thread is always triggered to run at low priority during instruction fetching, as opposed to [36]. 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. We assume the ICOUNT2.4 fetch policy [133]. It means that at most 4 instructions can be fetched from up to two threads at any given cycle. If the first thread is able to supply 4 instructions, then instructions from the second thread cannot be fetched. Thus, due to the p-thread's low priority, instructions from the p-thread are fetched only when the main thread cannot consume the whole fetch bandwidth. We adjust priorities by imposing a constant bias against the p-threads.

VII.B.5 Precomputation Thread Synchronization

If a p-thread is too far ahead of the main thread, prefetching may not be effective, since prefetched data may be overwritten by other loads before being

consumed by the main thread, or prefetches may begin to replace useful data in the cache.

To prevent this from happening, a p-thread is synchronized to set a limit on how far ahead of 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 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 will block (or pause) 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. However, to minimize the impact of spin-waiting on the main thread, we make the p-thread's fetch priority even lower. This could also be done with some hardware assistance, possibly reusing the live-in pipe, but we chose here to use a low-overhead software scheme. For a multi-core implementation, however, we can use a mailbox or the live-in pipe (queue) as described above to communicate the main thread's loop counter.

VII.C Accelerating Precomputation Threads

Most research on precomputation based prefetching focuses on not letting the p-thread run too far ahead of the main thread, but there are cases where the p-thread cannot get far enough ahead to make prefetching effective.

In this section, we employ a few techniques to accelerate the p-thread ahead of the main thread. First, we use a co-location policy to reduce I-cache misses from the p-thread, and to minimize the I-cache conflict between the p-thread and the main thread. Second, we exploit dynamic hardware load stride

prediction to speculatively specialize p-slices, allowing for simpler p-slices with lower overhead. Finally, we dynamically examine a p-slice to identify its loop induction variable(s) inside the p-slice, and peel off that computation, allowing us to jump start the p-slice execution a few iterations ahead of the main thread. This technique works even if a p-slice does not have any predictable live-in values. For example, a two-dimensional array is dynamically allocated as shown in Figure VII.3. It first allocates a pointer array (row). Each pointer in the array then points to a memory buffer (column) allocated separately. Depending on the runtime memory allocation policy, the starting address of each column may or may not be strided. If the p-thread does a column traversal, the column's starting address is its live-in value. For every loop iteration, memory addresses accessed by the p-thread are strided, even if its live-in value (the column starting address) is not predictable.

VII.C.1 Precomputation With Speculative Strides

The Trident framework not only has the ability to detect access patterns revealed in the code, but also to detect strided loads via hardware monitoring. Thus, some loads that have a very complex recurrence (resulting in high-cost p-slices) can actually be prefetched with a simple strided recurrence instead. Since p-threads are executed speculatively, this is fine.

VII.C.2 Precomputation Jump Starting

Sometimes, the only way to get the prefetch thread ahead of the main thread is to give it a head start. Existing software precomputation schemes (e.g., [86]) typically start p-threads from the same starting point (same iteration) as the main thread.

Our goal is to jump start p-threads multiple iterations ahead of the main

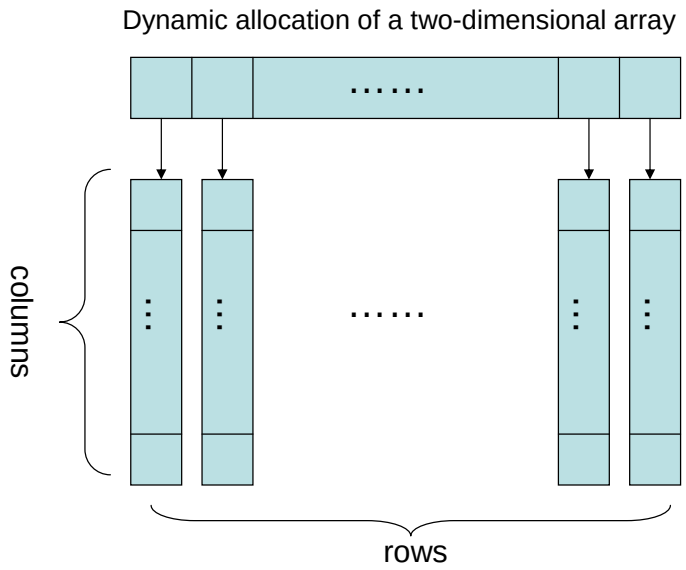

Figure VII.3: An example of dynamic memory allocation for a two-dimensional array. The pointer array (row) is allocated first, and each column is allocated separately. Then the pointers in the row array are assigned to point to individual column. Depending on the runtime memory allocation policy, the starting addresses of columns may or may not be strided.

Table VII.1: An example of the p-slice with the jump start instruction inserted. The p-slice has a loop with induction variable t1. The variable is incremented by 128 every iteration. We extract this induction variable from the loop and add 256 to it before the p-slice loop starts. This essentially lets the p-slice start off two iterations ahead of the main thread.

The original p-slice	The p-slice with the jump start instruction inserted
	ADDQ t1, 256, t1
##— loop starts —	##— loop starts —
LDQ v0, 0(t2)	LDQ v0, 0(t2)
ADDQ v0, t1, t3	ADDQ v0, t1, t3
PREF zero, 8(t3)	PREF zero, 8(t3)
PREF zero, 72(t3)	PREF zero, 72(t3)
ADDQ t1, 128, t1	ADDQ t1, 128, t1

thread. To do this, we scan the hot trace to identify its loop induction variables which are also included in the p-slice. We then peel them off and hoist them outside the p-slice loop (see Figure VII.1). Then we either duplicate the peeled code several times, or in many cases simplify it to a single instruction (e.g., if the induction variable adds a constant every iteration). Table VII.C.2 shows an example of a p-slice with a jump start. This example is extracted from one of the hot traces from *art*.

In this example, *t1* is the induction variable. We found that *t1* does not show any predictable pattern on each invocation of this p-thread. However, after the p-thread starts to execute, t1 is added by 128 in every loop iteration. So, we can still peel off the induction and hoist it outside the loop. The hoisted code looks like this:

```
ADDQ    t1, 256, t1
```

Here, the *ADDQ* instruction adds the constant 256 to register *t1*. Even if the initial value of *t1* is unpredictable, we can still jump start the p-thread two iterations ahead.

Note that we do not need to make any prediction in order to jump start the p-thread. In the example in Table VII.C.2, we do not predict the live-in value of t1. In contrast, *Future Execution* [50] relies on prediction of live-in values to get p-threads started ahead. Our technique works correctly even if the live-in values do not have any predictable patterns, because our jump starting is based on the actual code.

Determining the correct jump start distance can be difficult. It depends on the iteration count, the timing of both the main thread and the p-thread code (and how effectively loads are being hidden, etc.). However, we can apply a similar repairing technique as described in Section VI.E.2 to dynamically adjust the jump start distance until the prefetched loads are covered.

Note that induction unrolling was introduced in [116, 36] by simply duplicating the induction calculation instructions multiple times. We extend this idea by combining the induction analysis with prediction. For example, if the induction variable is a simple pointer chasing chain, we may break this pointer chain by predicting its stride value. We found this is not uncommon during our experiments. Then we can pre-evaluate the induction calculation into typically a single *add* instruction. This is also beneficial if we want to adaptively adjust the jump start distances using the repairing technique as described in Chapter VI.

VII.C.3 Precomputation Code and Hot Trace Co-Location

Another optimization we perform for p-slices is to co-locate the p-thread code with its hot trace. When a p-slice is constructed, the runtime optimizer needs to re-generate a new trace with the p-thread trigger and termination instructions inserted, as shown in Figure VII.2. Due to the coldest color layout policy in Trident, a new trace is located at the code cache blocks, which map to the least frequently used cache blocks. We can take even greater advantage of this allocation policy by appending the p-thread code to the end of its corresponding hot trace. This reduces I-cache misses for the p-thread, allows at least part of the p-thread code to be prefetched when the hot trace runs, and also eliminates cache conflicts between the p-thread and its corresponding hot trace.

Additionally, if a hot trace is invalidated, its p-thread code is invalidated accordingly. No separate code cache management is required. 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.

address delta = ABS (the p-thread prefetching address
- the last load address in DLT)
if (address delta / stride) > (runahead distance + threshold)
runaway prefetching is true

Figure VII.4: Runaway prefetching detection for a strided load.

VII.D Precomputation Prefetching Address Coherence

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 p-thread prefetching stream to diverge from the main thread's address stream. This may be due to unexpected control flow or due to store instructions that are left out during the p-slice formation, but we also see this in our system because of our use of speculated strides. 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 [35] uses the global history register to check control flow consistency between the main thread and the p-thread. But control flow consistency itself does not guarantee the prefetching address coherence due to, for example, the skipped store instructions during the p-slice construction.

In this research, we propose a low-overhead software solution where the p-thread checks if its address stream is coherent with the main thread. First, it looks up 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. For example, if the load has a speculated stride in the DLT, we can simply check if it is runaway prefetching as shown in Figure VII.4. If the load is not strided, we compare the address delta with a predefined range threshold to see if the p-thread gets off track. 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, but because the main thread will experience stalls when the prefetcher diverges, the p-thread will always get ahead and check for the divergence soon after it happens. This 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, but may not be possible for more complex address computation in the p-slice. In that case, the p-thread terminates itself immediately.

VII.E Methodology

We evaluate the performance of our p-thread based prefetching on a simulated simultaneous multithreading (SMT) processor [133]. The processor is assumed to have four hardware contexts. The processor fetch policy is ICOUNT2.4, as explained in Section VII.B.4. The processor baseline configuration, simulation benchmarks, and simulation points are the same as described in Section VI.F. The base performance when these benchmarks are executed alone on the baseline architecture is shown in Figure VI.3.

VII.E.1 Trident's Monitoring Hardware

Precomputation based prefetching uses hot traces to construct the p-thread code. Trident relies on hardware monitoring structures - the branch pro-

filer and the delinquent load table - to trigger optimization events to form hot traces and construct p-slices. These monitoring structures are the same as shown in Table VI.3 in Chapter VI.

VII.E.2 The Dynamic Optimizer

In this research, the main tasks performed by the runtime optimizer are generating and optimizing hot traces, generating the p-thread code, or inserting inlined software prefetches into hot traces if the p-thread code cannot be generated. We developed a lightweight optimizer to perform our proposed optimizations, which runs as a concurrent thread on our simulator, alongside the main thread and prefetching helper threads. The optimizer performs optimizations on the streamlined instruction traces. The optimizer is written in *C* and compiled with *gcc -O5* on the Alpha platform.

VII.F Performance Evaluation

In this section, we evaluate the cost and performance of our event-driven, dynamically generated, precomputation based prefetching technique. While the cost of the optimization system is fully reflected in the performance results, we isolate that cost for better understanding in the following section. The performance improvements shown in subsequent sections are relative to the aggressive baseline hardware prefetching, whose performance is shown in Figure VI.3.

VII.F.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. We want to minimize this overhead so that it will not overwhelm the gains of prefetching. We expect overheads to be low,

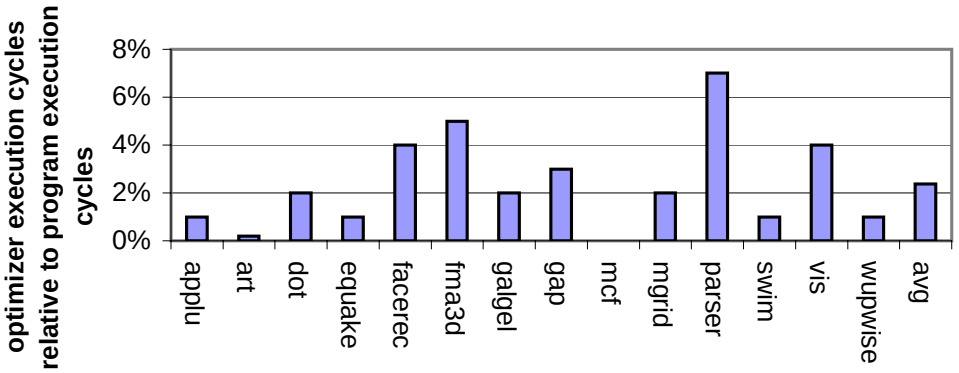

Figure VII.5: Concurrent execution cycles between the main and optimization threads. Prefetching threads do not run since hot traces are not activated during this measurement.

because monitoring is done in hardware, and optimization occurs in a separate thread. Thus the cost is indirect, depending on how often the optimizer runs, and how much it interferes with the main thread. To measure these two factors, we let Trident respond to optimization events as usual, but we do not actually use the optimized code. That is, the runtime optimizer is spawned to construct and optimize hot traces, but not alter the original binary to jump to the optimized traces. Pre-computation slices are constructed, but p-threads are not triggered to prefetch.

Figure VII.5 shows the percent of execution in which optimization threads are running concurrently with the main thread. The graphs shows that the Trident optimization threads are running, on average, 2.2% of the time. The actual interference is even less, as we observe the total performance degradation to the main tread to be only 0.6%. This is because our optimization thread does not interrupt the main thread's execution, and its low priority also leads to low execution resource demands. Note that we also discount extra instructions inserted in the main thread when calculating its instruction throughput.

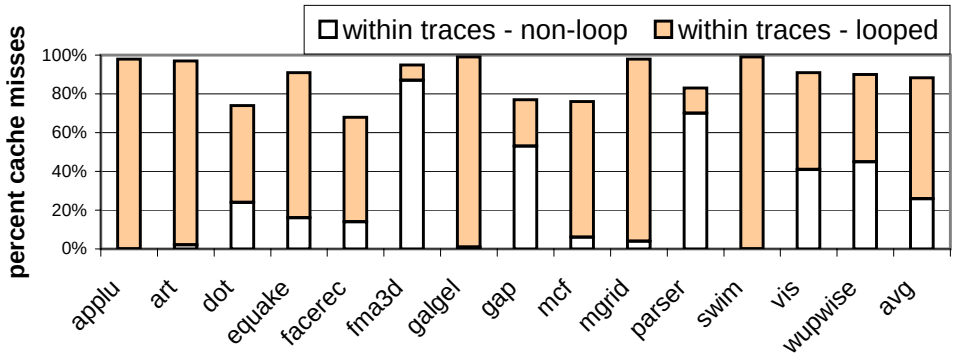

Figure VII.6: Dynamic load misses within hot traces. Since p-slices are constructed on hot traces with self-loops, our precomputation based prefetching only targets delinquent loads within looped hot traces.

VII.F.2 Load Coverage by Software Prefetching

The nature of the system we use limits us to prefetching only loads in hot traces, and particularly, only those loads appearing within hot trace loops. This section examines the impact of that constraint. Figure VII.6 shows the percent of cache misses which occur within 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 hot traces are broken up into those that are found in loops (looped) by the dynamic optimizer, and those not in loops. We observe that over 85% of load misses are within dynamic generated hot traces. Among them, nearly 55% of misses occur inside looped traces. These loads have the potential of being prefetched by using precomputation.

VII.F.3 Performance of Precomputation Based Prefetching

Figure VII.7 presents the performance of our dynamic prefetch framework, and a comparison to prior techniques. The baseline is the architecture described in the prior section with hardware stride predicted stream buffers for

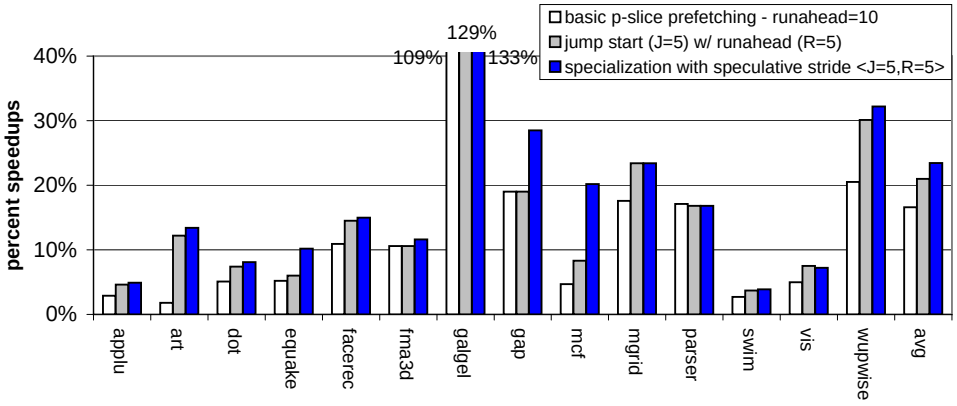

Figure VII.7: Performance of precomputation based prefetching. The performance gains are relative to the baseline architecture with a hardware stream buffer prefetcher. The first bar ("basic p-slice prefetching") is intended to represent the performance from prior research. The second bar is the Trident based scheme with the jump start distance 5 and the runahead distance 5, which makes the total prefetch distance 10. The last bar shows the performance when combining p-slice code specialization with jump start. P-threads are constructed only for the looped traces.

prefetching. All results shown represent performance gains over that architecture.

The first bar, labeled *basic p-slice prefetching*, represents the previous approach to p-slice generation. This result does actually include some of our enhancements described in this chapter, but we will let this result stand in for a generic thread-based prefetcher. For example, this basic p-slice result takes full advantage of the event-driven dynamic optimization framework, adapting to the specific runtime characteristics of the application. This result assumes a default prefetch distance for loads of 10 loop iterations for all loops, which was found to be a good overall distance. The p-thread starts execution from the same loop iteration as the main thread. This result provides 16% gains on average over the hardware prefetcher.

The second bar in Figure VII.7 represents performance improvement if we jump start the p-threads 5 iterations ahead of the main thread. Note that p-threads are constructed only for the looped traces. For fair comparison with the basic p-thread approach, we set the jump start distance to 5 and a prefetch distance 5 iterations from the jump starting point. So *effective* prefetch distance is still 10. This appears to be an effective tradeoff, sacrificing some initial misses to get out ahead of the main thread more quickly. Combined with this, for the second bar we also perform co-location of p-slices with hot traces. With this change, we observe as much as 20% performance improvement from *galgel* and 10% from *wupwise*. Overall we achieve an average 4% gain over the basic scheme.

The third bar in the figure shows the performance of using the speculative strides from the DLT to create low-overhead p-slices for loads that have more complex recurrences. Overall, the total contribution is 7% over the basic p-slice approach. This comes from a combination of (1) improving the p-thread code quality, (2) specializing the p-thread code for fast execution, and (3) jump starting p-threads to let them perform more efficient prefetching by running ahead of

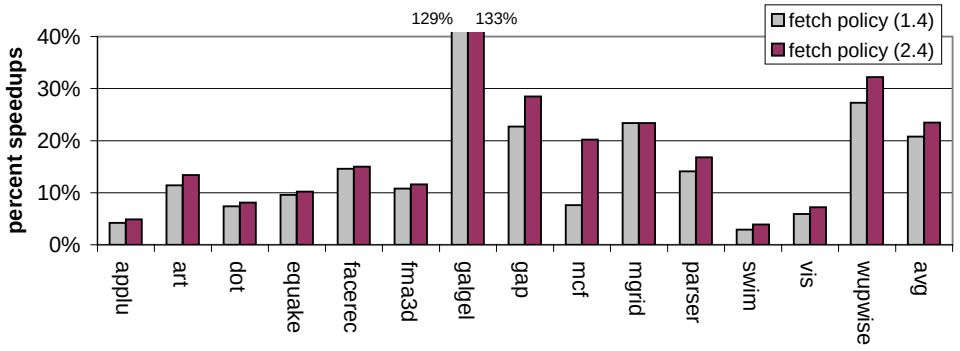

Figure VII.8: Comparison of instruction fetch policies. The policy ICOUNT 1.4 indicates that only one thread is fetched, with up to 4 instructions, at any given cycle. The policy of ICOUNT2.4 allows two threads, with total 4 instructions, to be fetched in one cycle. This is the policy used in this study.

the main thread.

As a side comparison, we want to see how much the low priority of p-threads impacts the performance using a different instruction fetch policy. In this research, we use the instruction fetch policy ICOUNT2.4, as described in Section VII.B.4. Figure VII.8 compares this policy with ICOUNT1.4. With the policy ICOUNT1.4, only one thread is fetched, with up to 4 instructions, at any given cycle. With ICOUNT1.4, both p-threads and the optimization threads can get completely locked out if the main thread does not stall. As we can see in this figure, the performance difference between these two policies is small, about 2.4%.

VII.F.4 Prefetching Address Coherence

Figure VII.9 shows the performance of the prefetching address coherence scheme. P-threads are code specialized with the jump start distance of 5. The second bar reproduces the results from the last bar in Figure VII.7.

The first bar represents the performance of precomputation prefetching

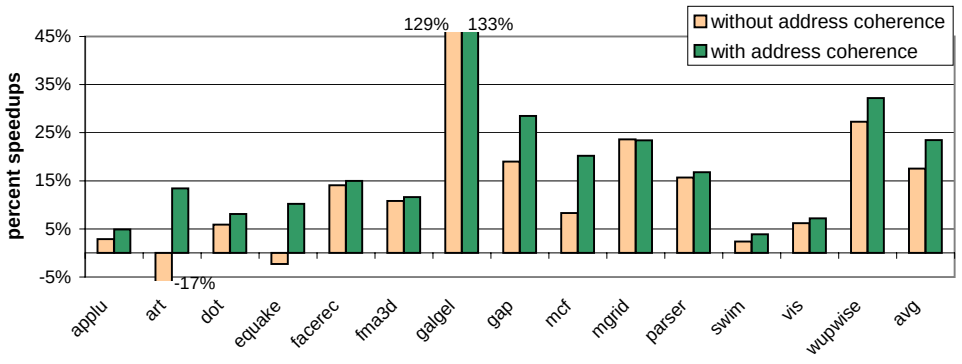

Figure VII.9: Performance of the prefetching address coherence scheme. P-threads have the jump start distance of 5 and the runahead distance of 5.

without the address coherence detection. It means, the p-thread runs independently with the main thread until it is killed. Thus, the p-thread may diverge from the main thread's address stream. Without the address coherence detection, we observe 17% and 2% negative speedups from *art* and *equake*, respectively. In this research, the divergence is mainly due to our use of speculated strides.

The address coherence scheme improves the efficiency of precomputation based prefetching by detecting and re-synchronizing/terminating runaway prefetching.

VII.F.5 Jump Start and Runahead Distances

Figure VII.10 shows the performance gains from different combinations of jump start and runahead distances.

At the runahead distance 1, the average performance increases considerably when the jump start distance increases. This is because the effectiveness of p-thread based prefetching improves when the p-thread is enough ahead of the main thread.

For the benchmark set used in this research, a large runahead distance (e.g., 10) with a large jump start distance (e.g., 9) does not further improve the

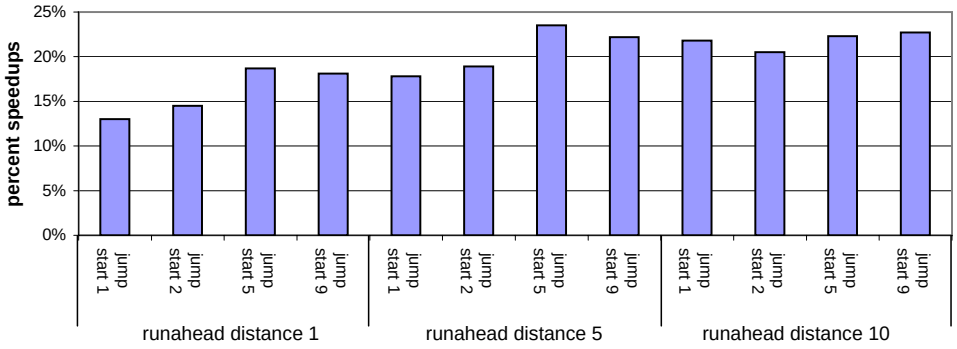

Figure VII.10: Performance of jump start distances with different runahead distances.

prefetching efficiency. This is because prefetching too far ahead increases the possibility of the prefetched data being replaced before used by the main thread or the useful data being replaced by the prefetched data.

Overall, the jump start distance 5 with the runahead distance 5 is a good combination in this research. Note that these two distances may be adjusted for individual traces via the adaptive searching technique described in Chapter VI. We will leave this for future work.

VII.F.6 Comparison with Inlined Prefetching

Figure VII.11 shows the performance from the combination of inlined prefetching with precomputation based prefetching.

The first bar in the graph shows the results for the inlined software prefetching as described in Chapter VI. This is an aggressive dynamic inlined 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.

The second bar combines the precomputation based approach, which is applied to all *looped* hot traces, with inlined prefetching (the first bar) for all *non-*

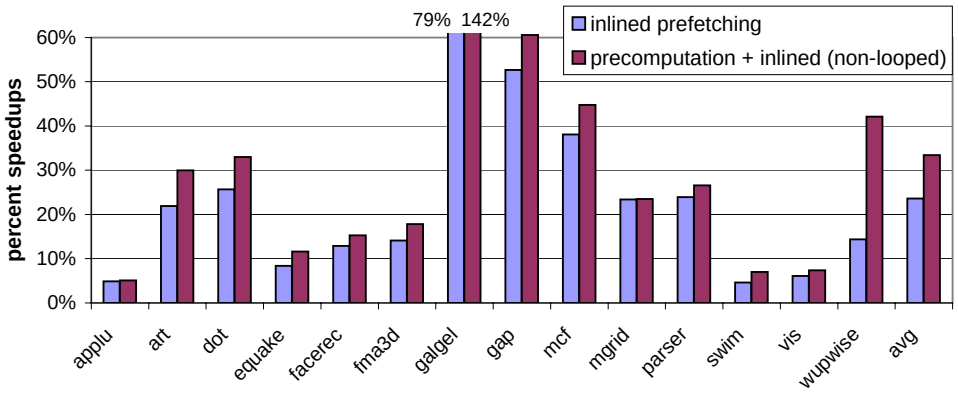

Figure VII.11: Performance of combining precomputation based prefetching (for looped traces) with inlined prefetching (for non-looped traces). P-threads have the jump start distance of 5 and the runahead distance of 5.

looped hot traces. This combination outperforms the basic p-thread scheme (in Figure VII.7) by increasing the speedup from 16% to 33% on average. Relative to aggressively inlined prefetching, our accelerated precomputation scheme achieves 10% performance improvement.

VII.F.7 Comparison with Larger Data Cache Sizes

In the Trident system, we use extra hardware resources to enhance the program monitoring on hot branches and delinquent loads. We want to study how much performance improvement we would expect if all these hardware resources were allocated to increase the data cache size, instead. We estimate that the total extra resources used in this study is about 14KB. If we dedicate them to increase the L1 data cache from two way to three way instead of enabling our dynamic optimization, we only observe 0.8% performance improvement. This compares with the 33% overall improvement we get by applying those resources to enable our event-driven generation of prefetch threads. This demonstrates that Trident uses the extra hardware resources more efficiently than simply increasing the data

cache sizes.

Summary

Pre-computation 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 ahead enough of the main thread to hide the memory latency as much as possible.

In this chapter, 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 also examine a mechanism to keep the p-threads in sync with the main thread of execution based on checking prefetched address streams with the main thread's address stream.

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 33% speedup relative to the baseline, which includes a hardware stride based prefetcher. In addition, using precomputation for loops with dynamic prefetching for non-loops achieves 10% speedups over the adaptive inlined prefetching technique in Chapter VI.

VIII

Summary and Future Work

This thesis proposes an event-driven dynamic optimization model. Based upon this model, we implement the Trident dynamic optimization system. The Trident system strives to reduce software runtime overhead as well as hardware complexity/inflexibility in dynamic optimization systems, with much improved runtime adaptability.

In light of this study, we conclude that effective dynamic optimization should have:

- Low overhead profiling of the program's behavior,
- Low overhead optimization,
- Continuous profiling and recurrent optimization to adapt to the program's changing behavior.

Static optimization achieves sub-optimal performance due to inaccurate assumptions about the underlying machine architecture and uniform assumptions across all phases of the program's behavior. Existing dynamic optimization systems overcome some of these limitations by specifically customizing optimizations at runtime. However, this is done at the cost of high runtime overhead (in software systems) or great hardware complexity (in hardware systems). As a result,

these systems often exploit less aggressive optimizations, and only achieve limited runtime adaptability.

Our solution to these issues is event-driven multithreaded dynamic optimization. It exploits the modern processor's on-chip parallelism to perform low-overhead optimization without interrupting the main thread's execution. Hardware support identifies the performance-critical events to trigger dynamic optimization with no software overhead. Upon these events, optimization is performed in an otherwise idle hardware thread, which runs concurrently with the program's execution. By allowing profiling, optimization, and execution to occur in parallel, our solution enables extremely low-overhead dynamic optimization. This makes it possible to enable continuous and recurrent optimization, which is difficult to do in prior dynamic systems.

VIII.A Trident Optimizations

Trident is a hot trace based optimization system. Hot traces are formed and optimized by helper threads, which are triggered by hot branch events. Trident's event monitoring and helper thread triggering provides a seamless mechanism for transparent dynamic optimization.

In Chapter IV, we first examine the benefit of using Trident to perform basic compiler optimizations. Trident's low overhead profiling and concurrent optimization allows more freedom to re-consider some design tradeoffs in traditional dynamic optimization systems. For example, trace linking is essential to the performance of traditional dynamic optimization systems. Trace linking is a technique that lets one hot trace directly jump to another hot trace by patching its exit branch target address to the beginning of the next hot trace. Because traditional systems do a context switch when starting to execute the optimized code from the un-optimized code (or vice versa), this technique can reduce the overhead

of context switches. In the Trident system, optimization is triggered by hardware events, so Trident does not need to gain control over the program's execution flow to optimize a trace. In addition, Trident does not have context switching overhead when switching between the optimized code and the un-optimized code. We found the trace linking has little performance improvement. In contrast, Dynamo [7] suffers up to *4000%* performance slowdown without trace linking.

Another advantage is that Trident enables invalidation of individual traces. In traditional dynamic optimization systems, trace invalidation involves chasing through traces via links and un-patching the target addresses of the linked branches with their original target addresses. This operation is very expensive, especially with multithreading support in the code cache [13]. With the removal of trace linking, Trident simplifies trace invalidation in the code cache. Trident uses the Trace Invalidation events to remove individual traces. At the same time, this also provides Trident with more freedom to place traces into the code cache. Trident uses the cold color based placement policy to guide code layout for the instruction cache conflict reduction between the optimized code and un-optimized code.

Trident exploits a voting scheme during hot trace selection to improve hot trace quality. The voting scheme selects the longest common subsequence among multiple branch history paths after a particular hot branch. This scheme potentially avoids selecting a *warm* path, which may occur by simply picking the first path after the hot branch is detected.

Trident takes advantage of its knowledge of the underlying machine architecture by performing architectural specific optimization. In this thesis, we investigate branch mispredictions of the Return Address Prediction Stack (RAS) due to code optimization. Trident adds instructions to the compensation block in the hot trace to keep the RAS in a consistent state. The RAS is not mis-aligned

even if the hot trace branches out early, before reaching the end of the trace.

Speculative Dynamic Value Specialization

In Chapter V, we demonstrate Trident's effectiveness and flexibility via software-based speculative dynamic value specialization. We extend Trident with a lightweight hardware value profiler to exploit semi-invariant runtime values and stride values of load instructions within hot traces. The profiler raises a hot value event when a load's value becomes confident (i.e., this load is *hot*). The hot value event then triggers Trident to perform speculative value specialization on the hot trace.

Trident inserts software checks along the hot trace so that values used for specialization can be dynamically verified to ensure the correctness of the program. Recovery is automatically performed using the existing hardware misspeculation handling mechanism. Since instructions following the specialized load and the software check do not have data dependencies on them, they can be executed speculatively. Thus, dynamic value specialization can boost instruction level parallelism (ILP) significantly. Our simulation shows that value specialization can achieve over 20% speedup on average. It is a promising technique for tolerating memory latencies, even in the presence of aggressive hardware prefetching.

Comparing with existing dynamic optimization systems, Trident explores new optimization opportunities via value prediction. Trident is the first to exploit load stride values in dynamic code specialization systems.

Speculative value specialization extends the benefit of value locality beyond traditional value prediction. The advantage of this technique over compiler-based value specialization is its ability to specialize on values identified dynamically during execution and to adapt value specialization as the application changes

behavior. Additionally, dynamic value specialization significantly enhances the effectiveness of value prediction by propagating the prediction knowledge further down the dependence chain than previously proposed hardware mechanisms.

Adaptive Dynamic Software Prefetching via Self-Repairing

We extend the Trident optimization framework to perform software prefetching to target true cache misses by dynamically inserting prefetching instructions into hot traces. This technique performs stride-based object prefetching to hide the latency of the first access of the object as well as all of the fields touched in that object. During insertion, Trident identifies all of the accesses to the different fields of an object in a hot trace. Then it inserts the minimum amount of software prefetches to prefetch all of the parts of the object that will be used. Thus, Trident combines the effectiveness of software prefetching, which can analyze the code to recognize access patterns, with the advantages of hardware prefetching, which can exploit some patterns static software systems cannot and can adapt to the actual runtime behavior of individual loads.

Software prefetching has to be accurate and timely in order to be effective. We also show that the nature of dynamic prefetching has an intrinsic characteristic, which requires adaptive discovery of prefetching distances. This is mainly because, (1) heavy interaction between prefetches and neighboring load instructions makes it difficult to get prefetching distances correct through profile estimation. Incorrect prefetching distances only partially hide the load latency. (2) Prefetching should adapt as the program execution changes phases. However, in existing dynamic optimization systems (e.g. ADORE [84]), the inserted prefetch instructions stay unchanged until being flushed and re-generated in the code cache due to the high runtime overhead.

In contrast, the Trident framework allows the runtime optimizer to re-

peatedly optimize the same trace to adjust prefetching either because existing prefetching is not effective or because the program's behavior changes. This is done both through runtime estimation of loop timing, and through progressive updating and evaluation of the prefetch distance. Thus, the adaptive prefetching proposed in this thesis overcomes limitations of both static prefetching and dynamic prefetching approaches. Trident re-evaluates the effectiveness of the inserted prefetches through continuous hardware monitoring. Prefetches may be re-adjusted, or removed altogether, according to the program's runtime behavior.

With the dynamic self-repairing mechanism, Trident finds the proper prefetch distances by trying multiple distances until the correct one is found. We achieve an average 23% speedup relative to hardware stride based prefetching. In addition, Trident's self-repairing prefetching mechanism achieves 12% better performance than prior dynamic prefetching techniques without repairing.

Precomputation Based Software Prefetching

In Chapter VII, we attack the memory wall problem by exploiting Trident to build the precomputation code to target delinquent loads. Precomputation based prefetching is a powerful technique to hide long memory latencies, especially for complex load behavior. However, this approach has difficulty allowing precomputation threads to run far ahead enough of the main thread to hide all of the memory latency. At the same time, it allows the prefetching thread's address stream to possibly diverge from the main thread due to decoupling of the prefetching thread from the main thread. Runaway prefetching unnecessarily displaces useful data, resulting in more data cache misses. These problems can dramatically reduce the effectiveness of precomputation based prefetching. The goal of this study is to improve the precomputation efficiency by overcoming these problems.

We extend the Trident optimization framework to enable precomputation based prefetching by dynamically constructing the precomputation thread code from the main thread's hot execution traces. By embedding our precomputation thread generation in an event-driven dynamic optimization framework, we enable a few sophisticated techniques to accelerate the p-thread ahead of the main thread.

Trident exploits dynamic hardware load stride prediction to speculatively specialize/simplify precomputation code with lower overhead. It dynamically performs code analysis on a hot trace to extract the loop induction variables and duplicate the induction precomputation code ahead of the p-slice loop. This allows Trident to jump start the p-thread multiple loop iterations ahead of the main thread. Trident inserts code to enable a low overhead mechanism for tracking prefetching addresses to determine when the precomputation thread address stream becomes out of sync with the one in the main thread.

Thus, Trident combines software code analysis with hardware performance monitoring to improve the efficiency of precomputation threads. It can exploit some patterns static software systems cannot, and can adapt to the actual runtime behavior of individual loads. Trident-enabled precomputation based prefetching achieves an average 33% speedup relative to hardware stride prefetching. In addition, Trident complements the precomputation mechanism with dynamic inlined prefetching for non-looped hot traces, which achieves 10% speedup over the adaptive inlined prefetching technique in Chapter VI.

VIII.B Future Work

VIII.B.1 In This Thesis

There remain a few important research topics which have not been explored in this thesis. (1) The results in this thesis assume that there is a hardware context always available to spawn a helper thread. It is worth evaluating how much it impacts overall performance when Trident shares the hardware context with other applications or optimizations. (2) In speculative dynamic value specialization, values are dynamically verified. Specialized traces are not invalidated on mis-specialization. We rely on Trident's existing trace invalidation mechanism to invalidate mis-specialized hot traces. Since a value specialized trace branches out on the software check when it detects a mis-predicted value, a frequently mis-specialized trace has relatively low instruction completion degree. Thus, this trace is eventually invalidated when its average completion degree drops below the trace invalidation threshold. Future research should investigate the impact of more active invalidation on mis-specialization. At the same time, improving the value specialization efficiency by dynamically repairing the load values used for specialization is needed. (3) In precomputation based prefetching, the precomputation thread is triggered at the beginning of the hot trace, and is terminated at the exit of the trace. Future research should explore how to trigger the prefetching thread even earlier by inserting trigger instructions further up the execution stream. That is, trigger instructions reside in other hot traces, which are executed ahead of the current hot trace which corresponds to the to-be-spawned precomputation thread. In addition, it is also interesting to study the impact of our prefetching coherency mechanism relative to other mechanisms such as the control flow consistency based on the global branch history [36].

Hot Trace Profiling

Trident employs the hot trace as an optimization vehicle to enable more advanced optimizations. Thus, it is important for the trace selection scheme to select hot traces with high dynamic coverage of instructions and high completion degree.

In this thesis, we use the extended MRET mechanism from Dynamo [7] to detect hot branches. Multiple history paths are then collected after the hot branch, and the longest common subsequence among these paths is chosen as the final hot trace. Future work should focus on investigating other profiling mechanisms during trace selection to achieve high trace quality with efficient hardware support (e.g., the programmable path profiler [135]).

Alternatively, it may also be advantageous to investigate using the existing performance counter sampling mechanism without any hardware extension. Our low overhead helper threading mechanism in Trident, together with profiling samples, helps detect hot traces in a similar approach to ADORE [84], but with lower overhead. This is because Trident can still exploit its multithreading mechanism to perform optimization and to handle optimization events concurrently.

This idea can be extended even further to do software-based profiling and branch/value prediction. In software based profiling, profiles are stored in the memory buffer to be consumed by the optimization thread. Thus this mechanism may do profiling on very complex program behavior with high accuracy and high flexibility.

Code Cache Management

In this thesis, we largely ignore all issues related to the code cache management. Trident invalidates individual traces, not for the purpose of the code cache management, but for improving trace quality to avoid less effective

hot traces. However, the idea of offloading the invalidation to helper threads in Trident can be extended to manage the code cache without imposing much software overhead.

Thus, future work should investigate how to take advantage of Trident's multithreading mechanism to enable fast yet flexible code cache management. Current research on code cache management performs medium grained partitioning on the code cache. Partitions are invalidated and flushed via a FIFO approach. Future study needs to model the code cache capacity and the code cache replacement policies. By exploiting Trident's ability to invalidate individual traces, one can do code compaction in the code cache, and perform code relocation if traces previously placed in the code cache start to cause I-cache conflicts.

The helper thread managed code cache should have low runtime overhead and low complexity.

VIII.B.2 Advanced Optimizations with Trident

We recognize there is considerable opportunity to build upon and extend the Trident framework for other advanced optimizations, which are not addressed in this thesis.

- **Runtime Code Parallelization**: We can exploit the Trident optimization system to dynamically parallelize the program using traditional parallelization mechanisms. This is similar to the scheme in [101] but it does not need any static instrumentation or profiling. Optimization can be performed on loops or at the procedure level. The code is analyzed dynamically to resolve data dependencies. Unresolved data dependency can be synchronized via lightweight synchronization mechanism [134].

  Alternatively, transactional memory in recent research [55, 21, 110] can serve

as a dynamic code parallelization mechanism. The transactional memory model was initially proposed as a replacement for inefficient locking and synchronization mechanisms. It was also studied to parallelize applications in a straight-forward manner [21]. However, parallelizing the program with transactions needs significant performance tuning. This is because transactions, which are executed atomically, run in parallel. Any data dependency among transactions causes transactional execution to abort. At the same time, overflowing of transaction memory buffers during transactional execution also introduces high overhead. So, transactions should be selected with maximal parallelism but minimal inter-transaction data dependency. On the other hand, they are selected to avoid frequent buffer overflows. To ease complicated performance tuning, one can use the Trident system to dynamically detect loops in the program and naively parallelize them as transactions without further analysis. Then, use the adaptive approach such as Trident's self-repairing to dynamically tune the parameters of these transactions to avoid frequent abortion and buffer overflow, or simply recover from bad transactions by falling back to the original code.

- **Power Aware Dynamic Optimizations**: Power consumption is an increasingly limiting factor in the microprocessor design. Simultaneous multithreading (SMT), chip multiprocessing (CMP), or a combination, are most likely going to be the dominant processor architecture in the near future. However, compared with conventional superscalar processors, SMT worsens power density because an SMT processor increases resource utilization. CMP worsens power density because more processor cores are placed on the same die area. Increasing power consumption and heat generation can easily cause processors to fail. Dynamic optimization, on the other hand, may have a profound impact on meeting the processor's power challenges.

For example, Hazelwood et al [57] showed that dynamic optimization can be helpful to eliminate the architectural voltage emergencies by rearranging problematic instruction sequences. Here, we propose to exploit the low overhead and fast dynamic optimization, enabled by the Trident system, to re-optimize the code according to the instantaneous power and performance needs and to redistribute the code among the cores [90], or to re-optimize the code to target the specific core architecture in the heterogeneous multi-core systems [78].

- **Software Security and Reliability**: Dynamic optimization has the potential to add value to the domains of software security and reliability. Security shepherding [73] exploits the dynamic optimization system, *DynamioRIO*, to monitor control flow transfers during program execution to enforce a security policy. In the hot trace based optimization systems, like Trident, only the code within hot traces can be monitored for secure execution. Thus, one can take a different approach from the security shepherding. Software security checks can be inserted by the static compiler after sophisticated offline analysis. However, these check may result in significant performance slowdown at runtime. By leveraging dynamic optimization systems like Trident, one can dynamically re-optimize the hot traces and remove software checks from the trace according to runtime security requirements.

In the big picture, how dynamic optimization impacts future microarchitecture design is an interesting question. Dynamic optimization systems usually leave the underlying microarchitecture intact. Modern out-of-order (OOO) processors themselves provide limited dynamic optimization. For example, instructions can be issued out of order from the instruction window if operands of latter instructions are ready. The OOO performance largely depends on the size of the instruction window. But a large instruction window not only consumes

extra power, but also complicates both control logic and the scheduling policy, which negatively impact the architectural cycle time.

With the support of dynamic optimization, it is possible to greatly simplify the underlying microarchitecture. The out-of-order processors have limited OOO execution, but rely heavily on dynamic optimization systems to reschedule instructions. To fully achieve the potential from co-design of microarchitecture and dynamic optimization, more thorough studies are necessary.

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, 1986.

[2] F.E. Allen and J. Cocke. *A Catalogue of Optimization Transformations*. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[3] Y. Almog, R. Rosner, N. Schwartz, and A. Schmorak. Specialized dynamic optimizations for high performance energy-efficient microarchitecture. In *International Symposium on Code Generation and Optimization*, March 2004.

[4] E. Altman, D. Kaeli, and Y. Sheffer. Welcome to the opportunities of binary translation. In *IEEE computer*, March 2000.

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

[6] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 2000.

[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[8] T. Ball and J. Larus. Efficient path profiling. In *29th International Symposium on Microarchitecture*, December 1996.

[9] T. Ball and J.R. Larus. Optimally profiling and tracing programs. In *ACM Symposium on Principles of Programming Languages*, pages 59–70, 1992.

[10] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support $IA-32$ applications on Itanium(R)-based systems. In *36th International Symposium on Microarchitecture*, December 2003.

[11] M. Bond and K.S. McKinley. Practical path profiling for dynamic optimizers. In *International Symposium on Code Generation and Optimization*, March 2005.

[12] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *FDD0-4*, December 2001.

[13] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, March 2003.

[14] D. Bruening, V. Kiriansky, T. Garnett, and S. Banerji. Thread-shared software code caches. In *International Symposium on Code Generation and Optimization*, March 2006.

[15] M.G. Burke, J.D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, M. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, June 1999.

[16] B. Cahoon and K.S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[17] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[18] B. Calder, G. Reinman, and D.M. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[19] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, March 1999. (http://www.jilp.org/vol1).

[20] 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.

[21] B. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional execution of java programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

[22] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. In *Computer Languages*, January 1981.

[23] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y.N. Patt. Simultaneous subordinate microthreading (ssmt). In *International Symposium on Code Generation and Optimization*, March 2004.

[24] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. In *IEEE Micro(5), pp 32-45*, May-June 2005.

[25] H. Chen, J. Lu, W. Hsu, and P. Yew. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture*, 2004.

[26] Howard Chen, Jiwei Lu, Wei-Chung Hsu, and Pen-Chung Yew. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture*, 2004.

[27] M. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java program. *30th Annual International Symposium on Computer Architecture*, June 2003.

[28] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[29] W.K. Chen, S. Lerner, and R. Chaiken D.M. Gilles. Mojo: a dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[30] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S.B. Yadavall, and J. Yates. Fx!32: A profile-directed binary translator. In *IEEE Micro(18)*, March/April 1998.

[31] 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.

[32] Y. Chou and J.P. Shen. Instruction path coprocessors. In *27th Annual International Symposium on Computer Architecture*, 2000.

[33] E. Chung, L. Benini, and G.D. Micheli. Energy efficient source code transformation based on value profiling. In *International Symposium on Low Power Electronics and Design*, 2001.

[34] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

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

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

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

[38] K. Cooper and A. Dasgupta. Tailoring graph-coloring register allocation for runtime compilation. In *International Symposium on Code Generation and Optimization*, March 2006.

[39] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java Just in Time. In *IEEE Micro*, 1997.

[40] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Ghrysos. Profileme: hardware support for instruction level profiling on out-of-order processors. In *MICRO-30*, 1997.

[41] J.C. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta code morphing system: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *International Symposium on Code Generation and Optimization*, 2003.

[42] K. Ebcioglu and E. Altman. DAISY: dynamic compilation for 100% architectural compatibility. In *Technical Report RC 20538, IBM T.J. Watson Research Center, New York*, 1996.

[43] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S.J. Patel, and S.S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In *MICRO-34*, 2001.

[44] B. Fahs, T. Rafacz, S. Patel, and S.S. Lumetta. Continuous optimization. In *32th Annual International Symposium on Computer Architecture*, 2005.

[45] K.I. Farkas, P. Chow, N.P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[46] B. Fields, R. Bodik, M.D. Hill, and C.J. Newburn. Interaction cost and shotgun profiling. *ACM Transactions on Architecture and Code Optimization*, Vol 1(3), September 2004.

[47] C. Fischer and R. LeBlanc Jr. *Crafting a Compiler with C.* Benjamin-Cummings, Redwood City, CA, 1991.

[48] D.H. Friendly, S.J. Patel, and Y.N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessor. In *31st International Symposium on Microarchitecture*, 1998.

[49] C.Y. Fu, M. Jennings, S. Larin, and T. Conte. Value speculation scheduling for high performance processors. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[50] I. Ganusov and M. Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2005.

[51] P. Gibbons and S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, 1986.

[52] N. Gloy, T. Blackwell, M.D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *MICRO-30 International Symposium on Microarchitecture*, December 1997.

[53] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. Dyc: an expressive annotation-directed dynamic compiler for c. In *Theoretical Computer Science, 248(1-2)*, October 2000.

[54] M. Gschwind and E. Altman. Precise exception semantics in dynamic compilation. In *Symposium on Compiler Construction, France*, April 2002.

[55] L. Hammond, B. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[56] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE MICRO*, August 1999.

[57] K. Hazelwood and D.Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *International symposium on low-power electronics and design*, August 2004.

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

[59] K. Hazelwood and M.D. Smith. Generational cache management of code traces in dynamic optimization systems. In *36th International Symposium on Microarchitecture*, December 2003.

[60] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A Kyker, and P. Roussel. The microarchitecture of the pentium(r) 4 processor. In *Intel Technology Journal*, 2001.

[61] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[62] T. Inagaki, T. Onodera, K. Komatsu, and T. Nakatani. Stride prefetching by dynamically inspecting objects. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[63] Intel Corp. *Intel IA-64 Architecture Software Developer's Manual, Rev2.1*, October 2002.

[64] Q. Jacobson and J.E. Smith. Instruction pre-processing in trace processors. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, 1999.

[65] D. Joseph and Dirk Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[66] R. Joshi, M. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *International Symposium on Code Generation and Optimization*, March 2004.

[67] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture*, May 1990.

[68] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 Chip: a dual-core multithreaded processor. *IEEE Micro, Vol 24(2)*, Mar-Apr 2004.

[69] D. Kerns and S. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, 1993.

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

[71] 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.

[72] H. Kim and J.E. Smith. Hardware support for control transfers in code cache. In *36th International Symposium on Microarchitecture*, June 2003.

[73] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *the Proceedings of the 11th USENIX Security Symposium*, August 2002.

[74] A. Klaiber. The technology behind crusoe processors. In *Technical report, Transmeta Corporation*, Jan. 2000.

[75] D. E. Knuth. A history of writing compilers. *Computers And Automation*, December 1962.

[76] D. E. Knuth and F. Stevenson. Optimal measurement points for program frequency counts. *BIT 13, pp313-322*, 1973.

[77] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, March 2001.

[78] Rakesh Kumar, Keith Farkas, Norman Jouppi, Partha Ranganathan, and Dean Tullsen. Processor power reduction via single-isa heterogeneous multi-core architectures. In *Computer Architecture Letters, Volume 2*, April 2003.

[79] M.S. Lam. Instruction scheduling for superscalar architecture. *Annual Review of Computer Science, Vol 4, pp173-201*, 1990.

[80] M.S. Lam, E. Rothberg, and M.E. Wolf. The cache performance and optimization of blocked algorithms. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[81] P. Lee and M. Leone. Optimizing ml with run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, May 1996.

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

[83] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[84] J. Lu, H. Chen, W.C. Hsu, B. Othmer, P.C. Yew, and D.Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *36th International Symposium on Microarchitecture*, December 2003.

[85] J. Lu, H. Chen, P.C. Yew, and W.C. Hsu. Design and implementation of a lightweight dynamic optimization system. In *The Journal of Instruction-Level Parallelism*, June 2004.

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

[87] 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.

[88] C.-K. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[89] S. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computer Frontiers*, 2004.

[90] M.D.Powell, M. Gomaa, and T.N.Vijaykumar. Heat-and-run: leveraging smt and cmp to manage power density through the operating system. In *ASPLOS-04*, October 2004.

[91] M.C. Merten and et al. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *26th Annual International Symposium on Computer Architecture*, June 1999.

[92] M.C. Merten, A. Trick, E. Nystrom, R.D. Barnes, and W.M. Hwu. A hardware mechanism for dynamic extraction and relayout of program hotspots. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[93] M. Mock, C. Chambers, and S.J. Eggers. Dyc: an expressive annotation-directed dynamic compiler for c. In *33rd International Symposium on Microarchitecture*, 2000.

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

[95] T.C. Mowry, M.S. 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, October 1992.

[96] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[97] Robert Muth, Scott A. Watterson, and Saumya K. Debray. Code specialization based on value profiles. In *7th International Static Analysis Symposium*, June 2000.

[98] R. Nair and M.E. Hopkins. Exploring instruction level parallelism in processors by caching scheduled groups. In *24th Annual International Symposium on Computer Architecture*, 1997.

[99] S. Narayanasamy, T. Sherwood, S. Sair, B. Calder, and G. Varghese. Catching accurate profiles in hardware. In *Proceedings of the Nineth International Symposium on High-Performance Computer Architecture*, 2003.

[100] E. Nystrom, R.D. Barnes, M.C. Merten, and W.M. Hwu. Code reordering and speculation support for dynamic optimization systems. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[101] K. Ootsu, T. Yokota, T. Ono, and T. Baba. A binary translation system for multithreading processors and its preliminary evaluation. In *5th Workshop on Multithreaded Execution, Architecture, and Compilation*, 2001.

[102] S. Palacharla and R. Kessler. Evaluating stream buffers as secondary cache repacement. In *21st Annual International Symposium on Computer Architecture*, April 1994.

[103] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Java VM'02*, 2001.

[104] S. Patel and S.S. Lumetta. rePlay: A Hardware Framework for Dynamic Optimization. In *IEEE transactions on computers, Vol 50, No. 6*, June 2001.

[105] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. In *U.S. Patent 5,381,533*, January 1995.

[106] Erez Perelman, Trishul Chilimbi, and Brad Calder. Variational path profiling. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2005.

[107] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *33rd International Symposium on Microarchitecture*, 2000.

[108] C.G. Quinones, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzales, and D.M. 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.

[109] 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.

[110] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Annual International Symposium on Computer Architecture*, 2005.

[111] Won W. Ro and Jean-Luc Gaudiot. Spear: A hybird model for speculative pre-execution. In *International Parallel and Distributed Processing Symposium*, April 2004.

[112] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson. Power awareness through selective dynamically optimized traces. In *31th Annual International Symposium on Computer Architecture*, June 2004.

[113] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *MICRO-29*, December 1996.

[114] 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.

[115] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Annual International Symposium on Computer Architecture*, May 1999.

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

[117] R. H. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *International Conference on Parallel Architectures and Compilation Techniques*, 1996.

[118] S. Sair and M. Charney. Memory behavior of the spec2000 benchmark suite. In *IBM Thomas J. Watson Research Center Technical Report RC-21852*, October 2000.

[119] S. Sair, T. Sherwood, and B. Calder. Quantifying load stream behavior. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, 2002.

[120] S. Sastry, R. Bodik, and J.E. Smith. Rapid profiling via stratified sampling. In *Annual International Symposium on Computer Architecture*, June 2001.

[121] A. Shankar, S. Sastry, R. Bodik, and James E. Smith. Runtime specialization with optimistic heap analysis. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2005.

[122] 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*, September 2001.

[123] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, December 2003.

[124] T. Sherwood, E. Perelman, G. Hammerley, and B. Calder. Automatically characterizing large-scale program behavior. In *ASPLOS-X*, October 2002.

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

[126] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM Journal of Research and Development*, Vol 50(2/3), March 2006.

[127] B. Slechta, D. Crowe, B. Fahs, M. Fertig, G. Muthler, J. Quek, F. Spadini, S.J. Patel, and S.S. Lumetta. Dynamic optimization of micro-operations. In *Proceedings of the Nineth International Symposium on High-Performance Computer Architecture*, February 2003.

[128] J.E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing*, November 1992.

[129] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *29th Annual International Symposium on Computer Architecture*, 2002.

[130] J.G. Steffan and T.C. Mowry. The potential of using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[131] Sun Microsystems. *The Java Hotspot performance engine architecture*, 1999.

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

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

[134] D.M. Tullsen, J.L. Lo, S.J. Eggers, and H.L. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, January 1999.

[135] K. Vaswani, M. Thazhuthaveetil, and Y.N. Srikant. A programmable hardware path profiler. In *International Symposium on Code Generation and Optimization*, March 2005.

[136] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th International Symposium on Microarchitecture*, December 1997.

[137] R.L. Wexelblat. *History of Programming Languages*. Academic Press, New York, NY, 1981.

[138] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[139] W. Wulf and S. McKee. Hitting the wall: Implications of the obvious. In *ACM SIGArch Computer Architecture News*, 23(1), March 1995.

[140] Y. Yadama, J. Gyllenhaal, G. Haab, and W.M. Hwu. Data relocation and prefetching for programs with large data sets. In *27th International Symposium on Microarchitecture*, 1994.

[141] C. Zhang and C. Thompson. Pa-risc to ia-64: transparent execution, no recompilation. In *IEEE computer 33(3)*, March 2000.

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

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

[144] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching. In *Annual International Symposium on Computer Architecture*, June 1995.

[145] H. Zhou, J. Flanagan, and T. Conte. Detecting global stride locality in value stream. In *30th Annual International Symposium on Computer Architecture*, 2003.

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

[147] C. Zilles and G. Sohi. A programmable co-processor for profiling. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, 2001.

[148] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *35th International Symposium on Microarchitecture*, November 2002.