In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Sept. 2005.

An Event-Driven Multithreaded Dynamic Optimization Framework

Weifeng Zhang Brad Calder Dean M. Tullsen

Department of Computer Science and Engineering
University of California, San Diego

Abstract

Dynamic optimization has the potential to adapt the program's behavior at run-time to deliver performance improvements over static optimization. Dynamic optimization systems usually perform their optimization in series with the application's execution. This incurs overhead which reduces the benefit of dynamic optimization, and prevents some aggressive optimizations from being performed.

In this paper we propose a new dynamic optimization framework called *Trident*. Concurrent with the program's execution, the framework uses hardware support to identify optimization opportunities, and uses spare threads on a multithreaded processor to perform dynamic optimizations for these optimization events. We evaluate the benefit of using Trident to guide code layout, basic compiler optimizations, and value specialization. Our results show that using Trident with these optimizations achieves an average 20% speedup, and is complementary with other memory latency tolerant techniques, such as prefetching.

1 Introduction

It is increasingly difficult for static compilation to tailor the code to all of the potential behaviors that occur during execution. Dynamic optimization [2, 9, 13, 26, 3, 10, 32, 33, 4, 27, 34] has emerged as a technique to adapt the program to its current execution behavior. Since a program is dynamically optimized while it runs, it can automatically adapt to the program's changing behavior at run-time.

This paper describes a dynamic optimization system called *Trident* which exploits two features present in many modern processor architectures: increasing hardware support for runtime monitoring of execution, and the ability to execute multiple threads of execution, either through chip multiprocessing [18], hardware multithreading [37], or a combination [22]. This system allows execution and optimization to take place concurrently, significantly reducing overheads inherent to most prior systems.

Trident builds on the continuous optimization work done in ADORE [25, 26], which uses an additional software thread to profile and optimize hot traces. Our approach reduces the overhead of profiling and optimization by using hardware support to identify optimization events, and using dedicated helper threads to guide and perform the optimization. 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 events, they are easily handled by spawning a lightweight helper thread, which does not interrupt the main thread's execution. This allows the framework to employ much more aggressive optimizations without significant fear of performance loss, even if the part of execution being optimized is short lived.

Our framework focuses on efficient dynamic optimization for a multithreaded processor – in this paper, the hardware platform we examine is Simultaneous Multithreaded (SMT) [37]. In addition to describing Trident, we examine the benefit of using Trident to guide code layout, basic compiler optimizations, and a more advanced optimization, dynamic value specialization. Trident is flexible enough to enable a variety of optimizations at once, and we demonstrate it in this paper with value specialization combined with these basic dynamic optimizations.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes the simulation methodology. Section 4 describes the Trident architecture. Section 5 gives details of the system implementation and demonstrates the framework with the base optimizations. Section 6 presents a more aggressive optimization, dynamic value specialization, using the Trident framework. We conclude the paper in Section 7.

2 Related work

Our framework is based in part on a large body of prior research in dynamic optimization. In this section we focus on summarizing research on dynamic optimization in both software and hardware based systems.

2.1 Software-based Dynamic Optimization

We will first summarize the software dynamic optimizations breaking them into two groups – those that optimize native

binaries and those that optimize while performing ISA translation.

Native binary optimization systems There have been several software dynamic optimization systems proposed, such as Dynamo [2], DynamoRIO [4], and Mojo [9]. Dynamo provides transparent dynamic optimization on binaries. Dynamo detects a program's hot traces via interpretation. As such, Dynamo exploits optimistic design strategies to detect hot traces quickly in order to reduce the costly overhead of interpretation. Mojo [9] is similar to Dynamo, but it targets multithreaded Windows applications. A common attribute of these systems is that optimization is typically performed in the same thread as the main execution within a single hardware context. Sharing the same hardware context requires pausing the current program's execution to perform optimization. This also introduces additional runtime overhead due to heavyweight user-level context switching between execution, profiling, and optimization.

The binary optimization framework proposed by Ootsu, et al. [30], 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 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 optimizations from dynamically identified hardware events.

The *ADORE* framework [26, 7] is the closest runtime optimization system to the Trident framework. ADORE uses a separate OS level thread to perform profiling and optimizations (e.g. prefetching) by taking advantage of Intel Itanium specific hardware counters [20]. In contrast, our current Trident framework focuses on adding lightweight hardware to perform all of the profiling needed to guide our dynamic optimizations. The hardware interacts with the optimization framework by generating events that our helper threads consume to make optimization decisions as well as to perform the optimizations. In comparison to ADORE, our event-driven hardware profiling support avoids context switching to the additional thread for profiling, allows continuous monitoring of more complex behavior, and allows our framework to react immediately to events and thus adapt to shorter phases.

Translation optimization systems Dynamic optimization is commonly seen in dynamic translation systems [14, 15, 3] 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 basic block level. Binary translation is discussed more thoroughly by Altman, et al. [1].

In the Java Virtual Machine (JVM) [31, 5, 11], the JIT compiler interprets/compiles abstract Java bytecode and optimizes it to run on native machines. Optimization is usually tightly coupled with the virtual machine semantics. The Jrpm system [8], with a similar motivation as [30], speculatively parallelizes a single-threaded Java program to run on a CMP with thread-level speculation [18]. Background compilation [24] reduces runtime overhead of lazy compilation within a JVM by using an extra dedicated thread to perform just-in-time compilation. Our approach builds on this by using helper threads to provide low overhead dynamic optimization of any binary running on the processor.

In parallel with our Trident framework, Shankar, et al. [34] designed a runtime specialization system under the Jikes RVM. This system profiles both load values and store addresses to find semi-invariant loads and heap locations. Multiple traces are then specialized from the same dispatch point using each of these constants. Trident differs from this work by identifying one hot value for each load instruction and creating a single specialized trace using potentially multiple hot values.

2.2 Hardware-based Dynamic Optimization

Pure hardware optimization systems are often based on the trace cache architecture. These systems perform lightweight optimizations such as constant propagation, register reassociation, and *move* instruction elimination via renaming logic [16, 21]. The inflexibility of hardware optimization systems can be partially overcome using instruction-path coprocessors [10]. The *ROAR* architecture [29] greatly improves the effectiveness of dynamic optimization via hardware support of precise speculation, which is similar in spirit to the Transmeta Crusoe processor [14].

More recent work on hardware based optimizations are represented by the *RePlay* [32] and *PARROT* [33] frameworks. In these frameworks, control dependencies are speculatively removed to form long and atomic traces so that very aggressive code reduction and optimization can be performed. Additionally, several hardware acceleration schemes, such as hot spot detector [27] and control transfer in the code cache [23], are proposed to speed up dynamic optimization.

The pure hardware optimization systems (e.g., RePlay and PARROT) usually store optimized traces in a dedicated hardware trace cache. This can restrict how long the optimized traces are or require additional solutions to allow for longer traces, since the trace is not stored in the program's virtual address space. Additionally, variable-length traces may need to 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 pure 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

Pipeline	20-stage, 256-entry ROB, 224 registers
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
L1 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 1: baseline SMTSIM configuration.

Branch profiler	256-entry 4-way associative and each entry has a 4-bit counter. Three standalone 16-bit bitmaps
Value profiler	32-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>

Table 2: Trident profiler configurations.

across interrupts and context switches. In addition, many of the above hardware systems use a dedicated, hard-wired optimization engine to perform optimizations. Trident provides a more flexible, scalable optimization scheme, which enables user-level code to perform the optimization in the program's address space.

3 Methodology

Trident is simulated on a modified version of the SMTSIM multithreading simulator [37]. The baseline processor is a 20-staged superscalar with 2 hardware contexts. The baseline configuration is shown in Table 1.

To support the Trident framework, we propose a few new, small hardware structures which 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 and the value profiler – are shown in Table 2. More details on these hardware structures are in Section 4.

SPEC2000int benchmarks with reference inputs are used for evaluation. 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 [35]. 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. 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,

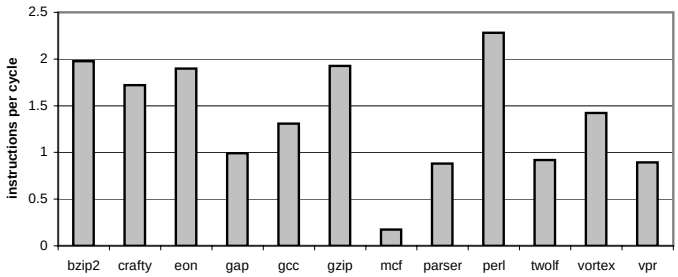

Figure 1: Performance on the baseline SMT processor.

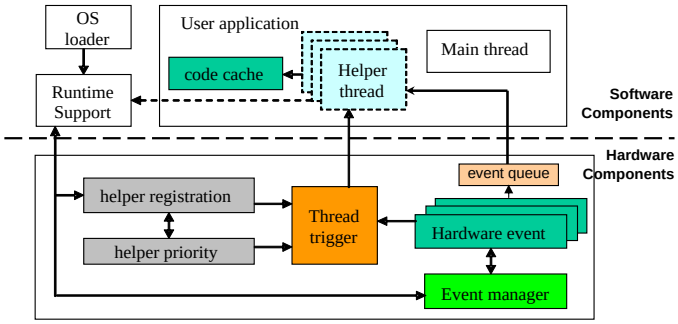

Figure 2: Trident dynamic optimization architecture

since we start simulation with no hot trace nor value specialization profiles. Figure 1 shows the base performance of each benchmark when executed alone on the baseline architecture. The base performance is used for future performance comparison.

Trident exploits helper threads to perform dynamic optimization on hot traces. The runtime optimization 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 4000 cycle latency when starting a helper thread. This consists of executing our run-time 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. This registration structure is described in detail in the next section.

4 Dynamic Optimization Architecture

Trident is an event-driven optimization system. It employs the hardware structures to monitor the program's execution behavior, and generates hot events to trigger optimizations. These structures and the mechanism to trigger helper threads on events are mostly general-purpose – we anticipate a system with a wider set of hardware-supported optimizations than are evaluated in this paper. In this section we give an overview of Trident's optimization architecture, and describe the generic hardware structures to support the Trident framework.

4.1 Trident's Optimization Architecture and Run-time Support

Figure 2 provides an overview of the Trident architecture. The major hardware components include the hardware event monitors, hardware event queue, event management and support for triggering the helper thread. To describe the architecture we will use the example of a dynamic optimization system that focuses on finding hot paths, and then performs code layout for those hot paths.

To apply Trident's optimization system, the OS loader calls run-time routines to specify that a given main thread is to be monitored and optimized for hot path code layout. In our current simulation infrastructure, the hardware monitoring is assigned to one running thread at a time.

When starting the monitoring and optimization process, the run-time support communicates to the event manager that certain hardware events are to be monitored. In our example, the hardware event is to find the starting PC and the subsequent hot path. Along with the hardware structures to find this information, the hardware provides an event queue. When an event occurs, the data found is put into the hardware event queue to be consumed by an optimizing helper thread. In our example, the helper thread will perform hot path code layout when a hot path event is inserted into the event queue.

At the same time the run-time support registers the hardware monitoring of a thread, it also creates (but does not run) a generic helper thread to process the hardware events from the event queue. During this registration the run-time 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.

When registering an optimization helper thread, the run-time 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, 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 in this paper interact with the code cache. The priority may affect the helper thread's instruction fetch throughput, to control its impact on other threads.

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, 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.

In our example, when a hardware event identifies a hot path, it dumps the event related data (starting PC and the subsequent path) into the hardware event queue – when the event queue fills up, new events overwrite older events. 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 run-time system, 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. When the helper thread starts, or when it finishes processing its current event, the helper thread reads the next event data from the hardware queue. In our example, it reads the PC and branch history, performs code layout for that hot path and stores the new code into the code cache. It then updates the code cache data structure, and finally it links the execution in the main thread's code to the code cache, and subsequent fetches to that hot path would execute from the code cache.

4.2 Hardware Monitors to Support Trident's Optimization System

For the optimizations we examine in this paper Trident supports the following hardware monitors. Some of these can trigger events, and others are just used by Trident during optimization.

Hot Path Profiler This profiler identifies the frequently executed 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.

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. For this study, we set L to be 16 branches. 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.

For our results, the best configuration we found was to keep track of 3 bitmaps. Using this design, assume for a hot branch we have the following 3 global history paths of length 8 (our results use length 16) – 10010100, 10110100, and 10110000 – to illustrate how the voting works. When picking the dominant path, the first two branch directions 10 are in agreement among the 3 sampled paths. 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 will then be used by the helper thread to create optimized hot traces.

Hot Value Profiler For the optimizations we examine in this paper, we use a hardware value profiler to identify frequently occurring values from load instructions. To accomplish this we adapted the software value profiler from [6]. The profiler is organized as a set-associative cache, where each entry is assigned to track the top values for a load. Each profiler entry keeps track of a small number (e.g. six) of load values that are tracked, and one entry to keep track of the dominant stride seen between the values for the load.

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 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 paper, we use the hot value profiler to only monitor load instructions in the scope of the most recent hot trace specified by the hot branch profiler. Therefore, the value profiler only needs a small number of entries. To specify what to value profile, load instruction PCs are put into the value profiler via a special profiling instruction, which is described later. If there is no room in the cache, then the least recently inserted entry is replaced.

Cache access counters Trident polls I-Cache access counters that estimate which cache blocks are hot. This is used to make decisions as to where to place optimized traces in the code cache in order to reduce cache misses. Similar counters are already available on modern processors like the Itanium [20].

Optimized trace watch table This hardware table monitors the performance of optimized hot traces. The goal is to identify when an optimized trace is 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 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.

Note that 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.

4.3 Trident's Optimization Flow

In this section we describe the different hardware events that can occur in more detail. An example of Trident's optimization flow, when applying all of our current optimizations, is shown in Figure 3. Optimization is driven by the hot events which are listed in Table 3.

The first step shown in Figure 3 is for the run-time system to turn on hot branch path profiling for a given thread and to allocate a helper thread registration structure as described above. When a *Hot Path* event is triggered and inserted into the hardware event queue, the event manager takes these actions: (1) copies the branch profile into the hardware event queue, and (2) checks to see if a *Thread Trigger* needs to oc-

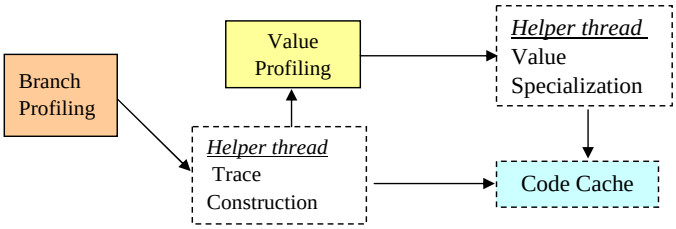

Figure 3: The dynamic optimization flow

Event	Source	Actions
Hot Path	Path profiler	helper thread spawned to construct a hot trace
Hot Value	Value profiler	helper thread spawned to perform value specialization on a hot trace
Code Cache Invalidation	Watch table	invalidate the hot trace corresponding to the virtual address in the watch table

Table 3: Trident hot events

cur. If a helper thread is not already running, then the registration structure is used to start the execution of the helper thread on a free context.

When a helper thread processes an event form the hardware queue it first determines what type of event it has, in order to invoke the correct routines. When a hot path event is consumed, the helper thread will start running the *Trace Construction* code to build an optimized sequential trace for the hot path found. This will be inserted into the code cache as shown in Figure 3, and then linked into the original code. The hot path trace generated will be terminated at the beginning of an existing hot trace, a self loop, or any indirect jumps. However, a *return* instruction doesn't terminate the trace when its matching *call* instruction is within the trace. The thread performs basic optimizations on the trace as explained in the next section.

The *Trace Construction* thread will then execute instructions to insert the PCs that it wants to value profile in the generated hot trace into the hot value profiler hardware structure. The value profiler will then raise a *Hot Value* event when hot values are detected.

When a hot value event is consumed by a helper thread, the *Value Specialization* code is run. The thread performs more aggressive optimizations (such as dynamic value specialization, described in Section 6). When the specialization is done, the helper thread stores the optimized value specialized 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, that trace can now be invalidated, and is marked to be removed later during code garbage collection.

When a code cache invalidation event is consumed by the helper thread, it will unlink the trace so that the original code jumps to its original branch destination. 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 or to replace a trace with a more optimized version.

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.

5 Trident Implementation

Trident is designed to quickly respond to hardware events and perform dynamic optimizations with very low overhead. The performance of generic dynamic optimization depends on design strategies in three major areas: hot trace selection, code cache management, and trace optimizations. These strategies have been studied by many research groups [2, 9, 4, 26, 19]. In this section, we describe our techniques to adapt or improve on some of these design strategies for our framework, and address difficult issues which have not been studied in most existing software dynamic optimization systems.

5.1 Trace Formation Optimizations

The basic optimization performed by most dynamic optimization systems is to create optimized hot traces, which by itself helps improve fetch throughput and branch prediction accuracy. When Trident forms a hot trace it gets an event with a starting PC, and a dominant branch history from the hot path profiler as described in Section 4.2. 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.

After a hot trace is constructed, 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. For example, if a call instruction and its matched return instruction are on the trace, both instructions can be removed. However, the return address is still moved into the calling register. This allows the trace to branch out before the return is reached.

On top of trace formation we perform basic optimizations such as constant propagation, copy propagation, and redundant instruction removal. Redundant loads are removed if there are no intermediate store instructions between them in the trace. A *move* 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 reallocation during basic optimizations, so we scavenge free registers to be used only within the basic block in which they are redefined.

We measure the amount of time the dynamic optimizing helper threads spend executing hardware events. Figure 4

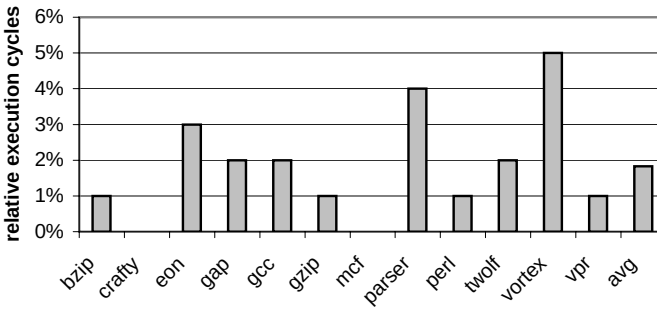

Figure 4: Percent of the main thread's execution in which the helper threads are running (processing hardware events).

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 optimizations described above along with value specialization described in Section 6. 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.

5.2 Candidate Hot Path Starting Points and Trace Linking

Trident uses the hot path hardware profiler described in Section 4.2 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 for 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 5 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 simulated various combinations of *B* and *T*, and the scheme $<3.08>$ performed the best. Since hot traces are likely executed more frequently, voting 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 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 4.2, 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.

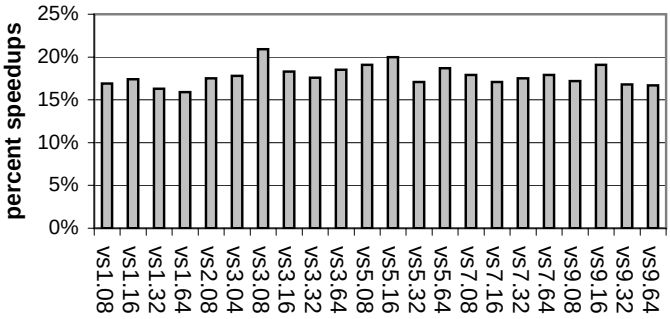

Figure 5: Comparison of hot path selection schemes.

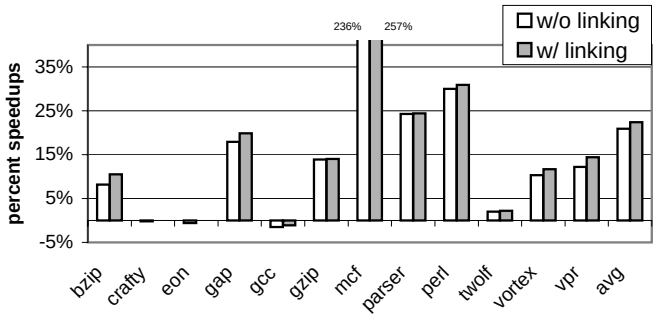

Figure 6: Performance with and without linking.

Because we don't let branches in the code cache become candidates for starting a trace, it might be worthwhile to apply *linking* among optimized traces so that one hot trace can jump directly to another optimized hot trace in the code cache without going back to the original code. Figure 6 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* [2], 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.

5.3 Hot Trace Invalidation

Trident exploits the watch table described in Section 4.2 to generate *Code Cache Invalidation* events which 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 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

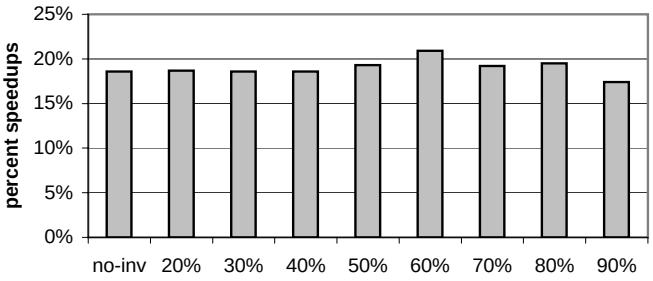

Figure 7: Code cache invalidation with different thresholds.

more dominate paths that can be represented. Invalidation involves repatching the original code with the original instruction stored in the code cache directory and flushing the corresponding I-Cache blocks.

Figure 7 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.

5.4 Color-based Code Placement

One of the most important benefits from dynamic optimization is instruction relayout. 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 the I-cache conflict between the optimized code and un-optimized code.

Here, we evaluate three different policies to layout the optimized code in the code cache. The base 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 new trace to the *coldest* color whose corresponding I-Cache blocks are least accessed.

Figure 8 shows the instruction cache misses for these different policies. Cache misses are measured in every million

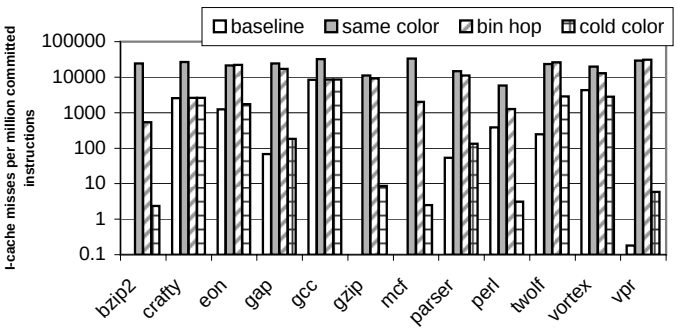

Figure 8: I-Cache misses on various placement policies.

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 conflict 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`.

5.5 Optimizations to Reduce Branch Misprediction

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 [23] 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.

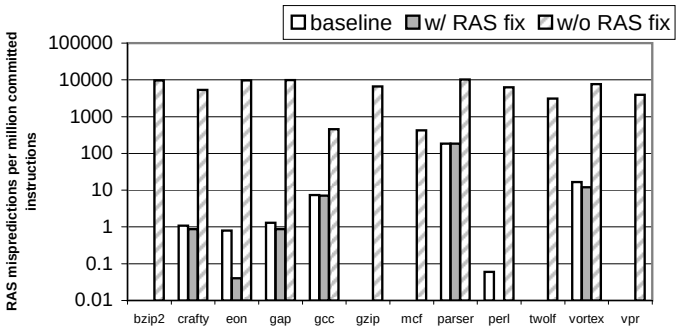

Figure 9: RAS mispredictions due to dynamic optimization.

Figure 9 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.

6 Value Specialization

Trident's fast event handling and low overhead make it suitable for more aggressive optimizations such as dynamic Value Specialization (VS). Value specialization [6, 28, 17], 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 relatively small ranges. Trident can perform value specialization on any hot trace via semi-invariant load values found via the hot load profiler. We focus on 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 [6].

As described in Section 4, Trident exploits the value profiler to watch load instructions only within the hot traces. 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.

6.1 Dynamic Value Specialization

As described in Section 4, Trident employs the hot value profiler to detect semi-invariant values of loads which are in the range of hot traces. A helper thread is triggered by *Hot Value* events to perform value specialization using these "constants". Value specialization, which includes constant propagation, copy propagation, and redundant code elimination, takes the following steps:

1. Construct a def-use chain on the trace. Hot values are then propagated along the chain. Any new constants generated during the propagation are further propagated.
2. If the load values are special integers (such as 0 and 1), consumer instructions of these values may be strength reduced. The *move* instructions generated after the strength reduction are copy propagated along the trace. Branch instructions depending on these "constants" may be eliminated.
3. 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.

6.2 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 [6]. This allows the load's dependency chain to be broken and results in reducing instructions on the hot trace critical path with the above optimizations. In addition, code below the value specialized load (and below the check and branch) can speculatively execute above it in out-of-order processors.

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 registers. 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 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 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 hasn't 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.

6.3 Exploring Stride Values

Trident's speculative value specialization is also able to explore run-time values with semi-invariant strides. To the best of our knowledge, this is the first time stride values are exploited in software-based value specialization. We found

that stride prediction is particularly useful for certain pointer-chasing codes, as also seen in [12, 36]. 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 4.2. 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 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 paper we only needed eight entries in a BVB for a given program.

6.4 The performance of value specialization

Trident performs value specialization using profiled hot load values. In this section, we evaluate the performance of Trident's value specialization (VS), and compare it with traditional (hardware based) value prediction. In VS, the value confidence scheme in Section 4.2 is used to identify hot values. All speedups quoted are relative instruction throughput

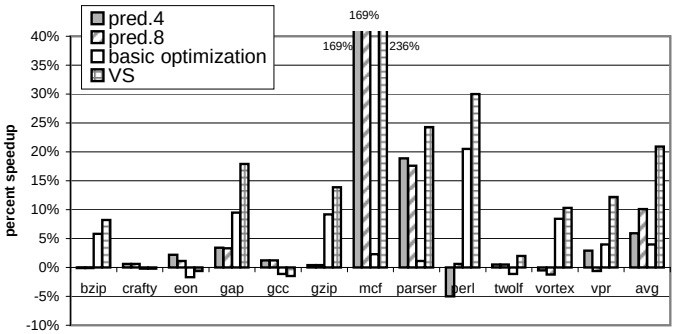

Figure 10: Comparison of value specialization with value prediction [38].

of just the main program, using instruction counts that correspond to original program execution.

One of Trident's advantages is that a value-specialized hot trace may embed many value predictions. In contrast, the complexity of conventional hardware value predictors would likely limit how many predictions can be made each cycle. We compare Trident's value specialization with an efficient hybrid predictor proposed by Wang, et al. [38]. 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 10. The first and second bars show the performance of the hardware predictor with 4 and 8 predictions per cycle, respectively. The third bar shows basic dynamic optimization, which involves basic block inlining and redundant instruction elimination. 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 11 shows the breakdown of load instructions that are covered by the hot traces with value specialization. The

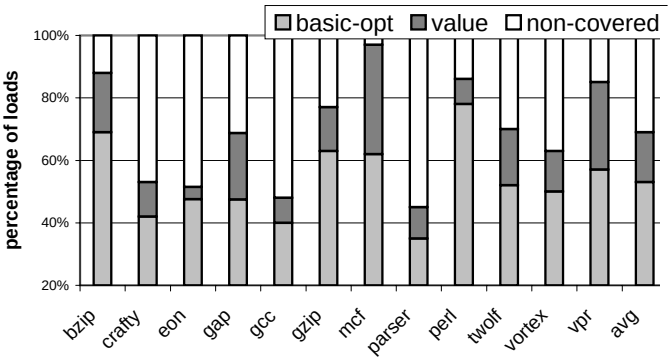

Figure 11: Breakdown of dynamic load instructions.

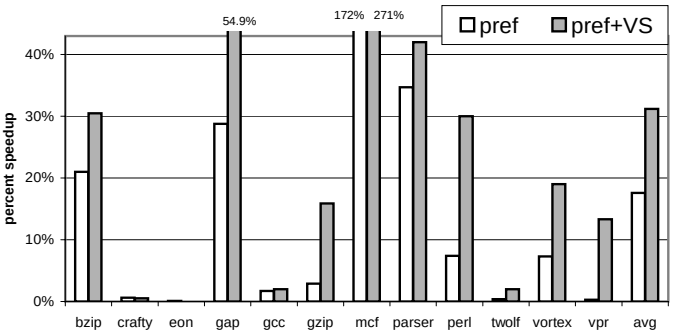

Figure 12: Performance of Value Specialization with Prefetching [36].

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 from 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.

6.5 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. [36]. 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 12. The first bar shows the IPC improvement from stream prefetching. The last bar shows the performance improvement from value specialization when combined with stream prefetching. Comparing Figure 10 with Figure 12, Trident's VS alone outperforms the hardware prefetching. Furthermore, Trident's value specialization is complementary to hardware prefetching, showing strong gains on top of that available to prefetching alone.

7 Conclusion

This paper presents an event-driven dynamic optimization framework, called *Trident*, with results applied to a simultaneous multithreading processor. Trident takes advantage of the additional hardware contexts and event counters available in modern processors. This allows execution and optimization to take place concurrently. With the support of some proposed new small hardware profiling structures, Trident spawns helper threads on hot profiling events to perform dynamic optimization on hot traces. Hot event registration, event monitoring, and helper thread triggering provide a seamless mechanism for transparent dynamic optimization.

Due to the parallel execution of helper threads with the main execution thread, Trident introduces very little negative performance impact on the application. In this paper, we improve the hot trace detection scheme over previous systems, and use 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 due to code optimization.

We demonstrate Trident's effectiveness via software-based value specialization. Trident is able to exploit semi-invariant runtime values and stride values to specialize hot traces. Our simulation shows that value specialization can achieve over 20% speedup on average. It has been shown to be a promising technique for tolerating memory latencies, even in the presence of aggressive hardware prefetching, and extends the benefit of value locality further down the dependence chain than proposed hardware prediction mechanisms.

Acknowledgments

We would like to thank the anonymous reviewers for their comments. This research was supported by NSF grants CNS-0311683 and CCF-0311710, and a grant from Intel.

References

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

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

[9] 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.

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

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

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

[24] 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.

[25] 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.

[26] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. In *The Journal of Instruction-Level Parallelism*, Jun 2004.

[27] 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.

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

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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