Performance Characterization of a Hardware Mechanism for Dynamic Optimization

Brian Fahs Satarupa Bose Matthew Crum Brian Slechta
Francesco Spadini Tony Tung Sanjay J. Patel Steven S. Lumetta
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{bfahs,sbose,mcrum,slechta,spadini,tonytung,sjp,steve}@crhc.uiuc.edu

Abstract

We evaluate the rePLay microarchitecture as a means for reducing application execution time by facilitating dynamic optimization. The framework contains a programmable optimization engine coupled with a hardware-based recovery mechanism. The optimization engine enables the dynamic optimizer to run concurrently with program execution. The recovery mechanism enables the optimizer to make speculative optimizations without requiring recovery code.

We demonstrate that a rePLay configuration performing a small suite of simple optimizations on Alpha code attains an average of 13% reduction in execution cycles on the SPEC2000 integer benchmarks over a rePLay configuration not performing optimizations, and a 21% reduction over an aggressive standard superscalar microarchitecture.

1 Introduction

Dynamic optimization is gaining the attention of computer systems researchers as an effective means for boosting application performance. Code optimizations made dynamically can exploit the stable, possibly phased, behavior exhibited by a running application and thereby utilize information not available to a static optimizer. Furthermore, new code deployment techniques, such as dynamically-linked libraries, create barriers for traditional optimizers, but are amenable to dynamic optimization.

One of the key challenges for effective dynamic optimization is to perform aggressive code transformations at low overhead. The rePLay Framework [19] addresses this challenge by providing dynamic optimization support at the microarchitectural level. In this framework, a high-performance execution engine is augmented with a programmable optimization engine, allowing optimization to occur concurrently with program execution. The rePLay microarchitecture facilitates optimization by providing the optimizer with long, atomic code regions upon which optimizations, possibly *speculative*, can be performed. A hardware recovery mechanism recovers architectural state in the event that assumptions made during optimization—such as speculation of likely control paths—are invalid during execution. Performing speculative optimizations without the need to generate recovery code potentially increases the aggressiveness of optimizations.

In this paper, we present a quantitative and qualitative analysis of a rePLay-based microarchitecture that performs several simple optimizations. We provide an analysis of factors that affect an implementation of rePLay. We investigate some intrinsic characteristics of code that shed light on the optimization potential of rePLay and other path-based optimizers. We demonstrate that a rePLay configuration performing a small suite of optimizations can attain a 13% reduction in execution cycles (a 16% increase in effective IPC) on the SPEC2000 integer benchmarks over a rePLay configuration not performing optimizations, and a 21% reduction in cycles (27% increase in IPC) over an aggressive standard superscalar microarchitecture.

2 The rePLay Microarchitecture

In order to lay a solid foundation for the subsequent sections of this paper and to illustrate rePLay's potential benefits, we describe a prototypical rePLay implementation in this section. Further details on this design appear in [20, 19, 5].

The rePLay framework consists of five key components: (1) a frame constructor for creating candidate optimization regions, (2) a programmable engine for optimizing these regions, (3) a frame cache for storing these regions on-chip, (4) a component for sequencing between regions, and (5) a

mechanism to recover architectural state if speculative optimizations prove incorrect. These components are integrated into a processor as shown in Figure 1.

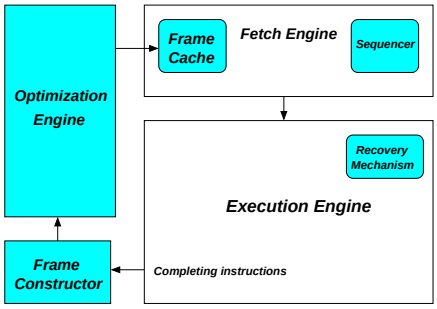

Figure 1. The rePLay mechanism is coupled to a high-performance microarchitecture.

2.1 Frames, assertions, and the recovery mechanism

Central to rePLay is the concept of the atomic region or *frame*. A frame is similar to a *trace* in a trace-scheduling compiler [7] or a *block* in the Block-Structured ISA [11, 15]. All control dependencies within a frame are removed, ensuring that all instructions within the frame are mutually control independent. In particular, either all instructions in a frame commit their results, or none of them do. This atomicity simplifies the optimization algorithms used and supports high-bandwidth instruction fetch.

To facilitate the creation of frames, rePLay includes the concept of an *assertion*. Assertion instructions enable multiple basic blocks to be combined into an atomic entity, or frame. A control flow assertion instruction is similar to a conditional branch in that both test a condition. They are different, however, in the actions taken after the condition is tested. The outcome of a conditional branch instruction determines the address of the next instruction. In contrast, an assertion has no effect on the address of subsequent instructions, and no effect whatsoever if the condition being asserted is true. If the condition is false, the assertion fires, triggering a recovery action that discards all instructions in the frame and redirects control flow to the original address for the instruction at the beginning of the frame (*i.e.*, the address of the first original basic block).

When an assertion fires (or an exception such as a TLB miss occurs) during a frame's execution, the hardware must roll architectural state back to the beginning of the frame. Such restoration implies that any state generated during frame execution must be buffered until all instructions within the frame have executed successfully, at which point state changes can be committed.

Buffering and recovery of architectural state in rePLay is accomplished using recovery mechanisms similar to those used by dynamically-scheduled processors with speculative execution. RePLay uses a reorder buffer-type mechanism to allow values generated within a frame to be used by subsequent instructions. Values are kept in the reorder buffer until the associated frame commits, at which point all values that are live at frame exit proceed to the architectural register file. If a frame does not commit, all values corresponding to the frame (and all values from subsequent frames) are flushed from the reorder buffer. This recovery action is similar to that required for a branch misprediction. Because of the potentially high number of values in-flight while executing a frame, rePLay's register recovery mechanism requires a deeper buffer. Similarly, values stored to memory are kept in a pending store buffer until the corresponding frame is committed.

2.2 Frame Constructor

While the rePLay Framework allows both compile time and dynamic frame formation, we focus here on a dynamic technique based on a hardware structure called a frame constructor. The frame constructor uses the committed instruction stream from the execution engine to build frames for optimization. Its objective is to create long frames that span multiple basic blocks. Long frames increase the potential for finding optimization opportunities not exploited at compile time. Branches and other control instructions at the boundaries between basic blocks in a frame are converted to assertion instructions.

The constructor works as follows: as instructions are retired by the execution engine, they are passed to the constructor. The constructor adds each arriving instruction into a frame construction buffer, causing the pending frame to grow. Whenever a control flow instruction is encountered, the constructor decides whether or not to convert it to an assertion via a technique for identifying highly biased branches called *branch promotion* [18]. An arriving control instruction not promoted into an assertion causes the pending frame to be finalized. A newly formed frame is passed to the optimization engine for optimization. With this technique, only highly biased branches are incorporated into frames; those that are not strongly biased form the terminal branches of frames.

As presented in [20], our path-history based frame construction technique generates long frames (on average 50 Alpha instructions) that have a very high probability of complete execution. On the average benchmark, these constructed frames provide 76% of all dynamic instructions.

2.3 Optimization Engine

The optimization engine can perform classical compiler optimizations, extended basic block optimizations, and various other optimizations performed by other dynamic optimization systems [1]. The rePLay optimizer can also schedule code, for instance if the underlying execution architecture is statically-scheduled. Moreover, the coupling of dynamic optimizations, execution rollback mechanisms, and rePLay's assertion instruction architecture allows for implementation of speculative optimizations (*i.e.*, optimizations that may not be valid in every execution scenario) without the need to generate recovery code.

The optimization engine is a flexible datapath that can be software-programmed using a specialized instruction set architecture, coupled with specialized optimization hardware that assists in performing critical optimization operations at low latency. The tight coupling between the execution and optimization engines provides the optimizer access to the microarchitectural execution state of the program, such as branch behavior, load-store dependence information, and intermediate data values.

While the prototype rePLay system embodies the optimizer as a separate hardware entity, other implementations are possible. For example, the optimizer can be a separate special microthread that shares the execution hardware with the application [3]. This thread would use special hardware to facilitate optimization, such as the ability to efficiently write into the the application thread's istream.

While our current rePLay microarchitecture does not save optimized frames into persistent storage, it is possible to write useful frames into an unused section of the the application's code segment, in a similar fashion to that proposed in [16].

Previous approaches to hardware-based dynamic optimization [8, 12, 4] have focused on simple microarchitectural optimizations. Such optimizations tuned a small trace of instructions (typically around 16 instructions) to the particulars of the execution microarchitecture. Example optimizations included instruction fusion (*e.g.*, combining shifts with adds), cluster assignment, and trace formation. Compiler style optimizations—optimizations that reduce the extent of computation—proved of little value because of the limited scope of a trace. Because frames are atomic and because rePLay is able to construct long frames, the potential of compiler optimizations increases with rePLay.

The rePLay framework differs from software-based optimizers such as the Transmeta Code Morphing System[13], HP Dynamo[1], and DyC [9] primarily in the use of the hardware support for optimization functions. The hardware support helps to reduce overhead in two ways: (1) the optimizer does not use the same execution hardware as the application, and (2) the hardware recovery mechanism allows for speculative optimizations on atomic code regions without recovery code.

2.4 Frame Cache

Frames processed by the optimization engine are stored in a frame cache. The frame cache is similar to a trace cache designed to deliver very long sequences of instructions, spanning multiple traditional cache lines. For example, a particular frame might consist of 100 instructions, span 13 cache lines (at 8 instructions per cache line), and take 13 cycles to be fetched and issued on an 8-wide fetch/issue processor. The frame cache must support frames of varying sizes (previous work reported that the distribution of frame sizes across applications is very broad), and must prevent a fetch from starting from the middle of a frame. Furthermore, the frame cache must treat each cached frame as an atomic entity: if any portion of a frame is to be evicted, all of it must be evicted.

Based on our evaluation of several frame cache [5] designs, the frame cache is partitioned into two structures. The first structure, the *head partition*, is a set-associative cache structure reserved for caching the first line of each frame. The other structure, the *body partition*, is a direct-mapped cache reserved for the remainder of the frames. Figure 2 illustrates how a frame is stored in the cache.

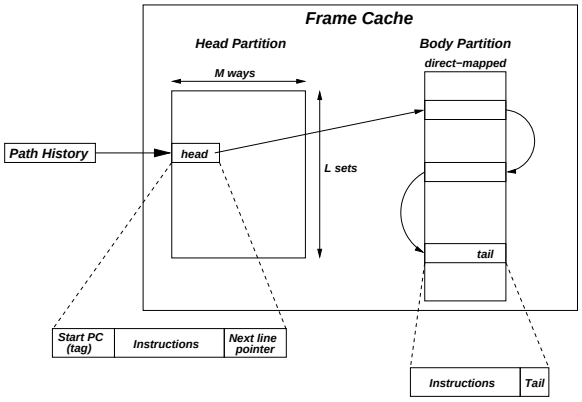

Figure 2. The frame cache consists of two structures: one for caching the first cache line of frames, and one for caching the remainder.

We've determined that, given this general structure for the frame cache, a good rule of thumb is to make the body parition roughly twice the capacity of the head partition. The particular partition sizes used in this study are presented in Section 4.3.

2.5 Frame Sequencer

While the program executes, a correlation-based frame sequencer decides appropriate situations for frame dispatch. The sequencer determines whether, for a given path history and current fetch address, a frame starting at the current fetch address with matching path history exists in the frame cache. If so, the frame is fetched and dispatched. Otherwise, the standard instruction cache provides the instructions for the fetch. For example, if the current path history is XYZ (meaning the last three branch targets were X, Y, and Z) and the current fetch address is A, the sequencer attempts to look for a frame starting at A with context XYZ.

The sequencer is coupled to a conventional branch predictor that decides which target to fetch in any particular cycle. The current path history is used to index into the frame cache. A comparison is made using the fetch address and the tag of the entries fetched from the frame cache. A match indicates that a fetch of the corresponding frame should be initiated, with the first line having already been fetched. A miss indicates that no frame exists in the cache for that particular path history and starting address.

3 Dynamic Optimizations

This section describes several simple optimizations targeted for rePLay's optimizer. While the space of possible optimizations is quite large, we select these optimizations because they are simple and powerful, and help to demonstrate the potential of optimization with rePLay. Some of the optimizations are common optimizations found in a standard compiler, while others are unique to rePLay. Specifically, we examine:

Dead code removal. Instructions whose results are never used and overwritten within a frame can be removed. Because a frame is a collection of basic blocks that embody a single control path, a rePLay optimizer can remove code that is dead on a particular control path. This optimization is quite effective, removing between 5% and 19% of dynamic instructions. In Section 7, we examine the potential of this optimization.

NOPs and unconditional branches are also removed by this optimization. NOP instructions are added by the compiler for cache alignment and instruction slotting and account for nearly 10% of the dynamic istream. This optimization removes these instructions, allowing for a net increase in effective fetch bandwidth.

Constant propagation. Constant values are often loaded into registers for subsequent computation. Such computation can sometimes be pre-calculated, reducing the amount of computation during execution. Because a path through multiple basic blocks is embodied as an atomic block, the rePLay optimizer can propagate a constant through a longer span of the code.

Reassociation. Often, multiple associative operations (*e.g.*, additions) are performed during the course of computation, for example to generate memory addresses or loop indices. When a sequence of these operations involving immediate values is detected within a frame, the optimizer can reassociate these values to reduce the computation tree height. The potential of this optimization increases substantially when applied across multiple basic blocks.

Common sub-expression removal. Code sequences sometimes produce the same value through identical computation. In the case of such redundant expressions, it is sometimes possible to remove one of the sets by forwarding the output value from the first to the second.

Sub-routine inlining. This optimization removes the stack and pointer operations associated with a call/return sequence when the sequence is contained within a frame.

Fetch Scheduling. Fetch scheduling attempts to improve the order in which instructions are fetched. Although many current processors employ dynamic scheduling, they are still constrained to fetching instructions in the order specified by the compiler. With rePLay, the optimizer can take advantage of having exact knowledge of the processor's parameters, such as fetch width, and can thus reduce inefficiencies, such as fetching two data-dependent instructions in the same cycle. Additionally, instructions on the critical path in a frame can be fetched earlier.

For example, we can schedule the fetch of assertions earlier to reduce the penalty associated with recovery. Of more critical importance, the position of the frame-ending branch is not relevant to correctness because the end of the frame is explicitly marked by the frame cache. We can move the branch and associated condition-generating instructions to the beginning of the frame. In doing so, we can reduce the branch resolution time, even if the underlying hardware uses dynamic scheduling. This particular optimization results in a net 3% decrease in branch resolution cycles.

Miscellaneous. The optimizer also performs a very limited amount of strength reduction (converting multiplies of powers of 2 into left shifts), and some remapping of common code idioms into more efficient sequences. The optimization engine also performs the previously-investigated trace cache optimization performing zero cycle register moves using existing register renaming logic [8].

To facilitate all optimizations, the rePLay architecture maintains a save/restore register mask with each frame. This mechanism enables the optimizer to use architectural registers that are not used in the frame as frame temporaries. The original value of a register used as a frame temporary is saved upon entry into a frame and restored upon exit. It should be noted that this process need not physically involve data movement in the implementation: the renaming mechanism can be augmented to provide this support by as-

signing temporary physical registers to frame temporaries.

We again note that the simple compiler optimizations performed here are different in nature from the microarchitectural optimizations performed in previous studies [8, 12]. Most of the optimizations investigated previously are complementary to those presented here, and if combined would result in an additional boost to performance. Also, as we demonstrate in Section 6.3, the longer nature of frames increases the ability of compiler style optimizations to make an impact on performance.

4 Experimental Setup

4.1 Benchmarks

We used the SPEC2000 integer benchmarks for performance characterization. Table 1 shows the number of simulated instructions for each benchmark. In most cases, the SPEC input sets were modified to enable the benchmarks to simulate all parts of the program in reasonable simulation time. All benchmarks executed to completion except vpr, which we capped at one billion instructions.

Benchmark	InstCount	Benchmark	InstCount
bzip2	289M	mcf	413M
crafty	620M	parser	508M
eon	132M	perlbmk	154M
gap	490M	twolf	595M
gcc	283M	vortex	265M
gzip	870M	vpr	1000M

Table 1. Benchmarks used in simulations.

The benchmarks were compiled using the Compaq Alpha C compiler, Compaq C V5.9, with optimization level 4 (the benchmark eon was compiled using g++ also at optimization level 4). At this level of optimization, the Compaq C compiler performs in-lining, loop unrolling, and code replication to eliminate branches. We also report several performance characterizations based on unoptimized binaries in order to provide another point of reference on the potential of the rePLay optimizer.

4.2 Simulation Environment

Our simulation framework is built upon the Alpha instruction-level simulator provided as the core of the SimpleScalar 3.0 tool set. We developed a timing simulator to model per-instruction processing delays associated with a dynamically-scheduled pipeline, including wrong path (branch misprediction) effects. To measure the effects of frame construction and optimization on performance, we coupled models of the rePLay frame constructor, optimization engine, and frame cache to the simulator.

We augmented the Alpha ISA to include assertion instructions corresponding to conditional branches, indirect branches, and return instructions. The rePLay instructions are inserted into frames by the frame constructor.

In order to establish the correctness of our optimizations, we compare the architectural state generated by the rePLay simulator with that of a reference simulator at every frame boundary. This technique ensures that rePLay optimizations do not cause incorrect program behavior.

4.3 Processor Configuration

We evaluate rePLay in the context of an 8-wide dynamically scheduled processor and compare it against baseline models in which the superscalar core is coupled with an instruction cache and with a trace cache. The specifics of the common core are provided in Table 2.

Pipeline	8-wide fetch/issue/retire, 7 cycles (min) for BR resolution
Inst Window	1024 instructions
ExeUnits	6 IALUs, 2 IMULs, 2 FLTs
L1 DCache	64kB, 1 cycle, 3 read/write ports
L2 Cache	1MB, 8 cycle
Memory	50 cycles

Table 2. Configuration of Superscalar Processor Core.

The baseline rePLay processor contains a 48kB Frame Cache and a 16kB instruction cache. As mentioned in Section 2.4, the cache consists of two partitions. The head partition is a 4-way set associative structure with 128 sets, meaning that the cache can store up to 512 frames. The body partition contains 1024 cache lines, direct-mapped. The line size of both the head and body partitions is 8 instructions (32B). The frame constructor contains a 32kB conditional branch bias table and a 4kB indirect branch bias table, each of which is accessed using a path history containing four previous branch targets. Branches are promoted after 32 consecutive outcomes to the same target. The default latency of the optimization engine is 1000 cycles (in Section 6.1 we examine the impact of varying this latency). The rePLay fetch mechanism uses a 8kB path-history based branch predictor with a standard 4kB branch target buffer.

The baseline Instruction Cache configuration extends the superscalar core with a simple 64kB Instruction Cache capable of performing split-line fetches (*i.e.*, fetching consecutive cache lines). The branch predictor is an 8kB gshare predictor with a 4kB BTB. The gshare predictor was tuned

to optimize the performance of this configuration.

We compare rePLay with a simple Trace Cache configuration capable of fetching traces of up to 16 instructions (delivered eight per cycle). The Trace Cache configuration consists of a 32kB Trace Cache and a 32kB Instruction Cache, with an 8kB gshare predictor. The Trace Cache model also performs some very simple optimizations, specifically the register move optimization described in [8]. With this optimization, instructions that move a value from one register to another are performed by the register renaming mechanism with no extra decoding time.

We have selected the execution core based on our projections of feasible organizations in one or two processor generations. The Instruction Cache and Trace Cache configurations represent standard microarchitectures utilizing this core. We evaluate the basic rePLay (*i.e.*, without optimization) configuration as an alternative to these conventional microarchitectures. Compared to the ICache and Trace Cache, the basic rePLay configuration does require extra storage for branch bias tables, but these tables are not on a latency sensitive section of the processor. In Section 5, we demonstrate that, from a performance point of view, basic rePLay is competitive with the standard ICache and Trace Cache organizations. The rePLay configuration with optimizations represents an enhancement beyond basic rePLay.

5 Performance Measurements

Our evaluation of rePLay involves a comparison between rePLay with (**RPO**) and without (**RP**) optimizations and the Instruction Cache (**IC**) and Trace Cache (**TC**) processor organizations described in Section 4. Figures 3 and 4 display the measured execution cycles for each of the SPEC 2000 integer benchmarks on these configurations.

5.1 Performance on optimized binaries

Figure 3 presents performance comparisons between the four configurations on statically optimized binaries.

In order to assess the factors that contribute to execution time, we subdivided each bar into categories based on how the fetch mechanism spends each cycle. For example, if in a particular cycle, the fetch mechanism produces some instructions on the correct execution path, the cycle is tallied as either a Normal Cycle, if the ICache or Trace Cache provided the instructions or a Frame Cycle, if the Frame Cache provided the instructions. The classification of cycles is made as follows: If the resulting fetch will be ultimately discarded because of a frame assertion, the fetch is tallied as an Assert Cycle. Otherwise, if all of the fetched instructions are on the wrong execution path (due to a mispredicted and yet unresolved branch), the fetch is tallied as a Wrong Path Cycle. Otherwise, the cycle is tallied as a Fetch Miss Cycle if there was a cache miss, or a Stall Cycle if the pipeline is stalled (due to a full scheduling window), or a Normal or Frame Cycle.

The data in Figure 3 has some notable trends: rePLay with optimizations outperforms the ICache configuration by an average of 21%, the Trace Cache by 18%, and basic rePLay by 13%. The percentage numbers included on the graph represent the improvement between rePLay with optimizations and basic rePLay.

For the benchmarks eon, gap, mcf, parser, and perlbmk, the optimizations delivere sizeable wins when coupled with the gains in fetch bandwidth delivered by the basic rePLay mechanism (on average 34% better than the ICache). For the benchmarks crafty, gcc, perlbmk, and vortex, the frame cache (and to some degree, the Trace Cache) suffers from cache misses due to effects such as redundancy (examined in Section 6.2).

The net loss due to assertions is remarkably low. On average this loss accounts for only 0.92% of all cycles.

The net loss due to branch mispredictions is a major factor on performance, in some cases contributing to over half the execution time. The rePLay mechanism is able to reduce the cost of mispredicted branches (*e.g.*, gap and perl) by reducing interference in the branch predictor—assertions, which account for over 80% of all branches, require no dynamic prediction. To a smaller extent, the optimizations themselves reduce resolution time. Optimizations such as reassociation reduce dependency height of branch computations, while the fetch scheduling optimization allows branch computation to be started earlier.

Basic rePLay itself performs only slightly better than the Trace Cache and ICache configurations when coupled with the 8-wide superscalar execution engine. Even the TC configuration barely outperforms the IC configuration. Several factors contribute to this: foremost, on average, the underlying machine is execution bandwidth-limited and increasing fetch bandwidth does little to improve performance. Secondly, we implemented a simple trace cache—enhancements such as those proposed in [18] and redundancy reduction techniques such as those proposed in [2, 21] can potentially boost the performance of the TC configuration. Furthermore, the microarchitectural optimizations evaluated in previous trace cache optimization studies can improve the performance of TC, RP, and RPO configurations.

Table 3 provides the average size of each fetch produced by the fetch mechanism for each of the four configurations (on Normal or Frame Cycles) for the average benchmark. Notice that the average fetch size drops slightly between the RP and RPO configurations. This reflects the optimizer's ability to remove instructions thus slightly reducing frame size. On average, the optimizer removes 16% of dynamic instructions from the optimized binaries. On average, 76%

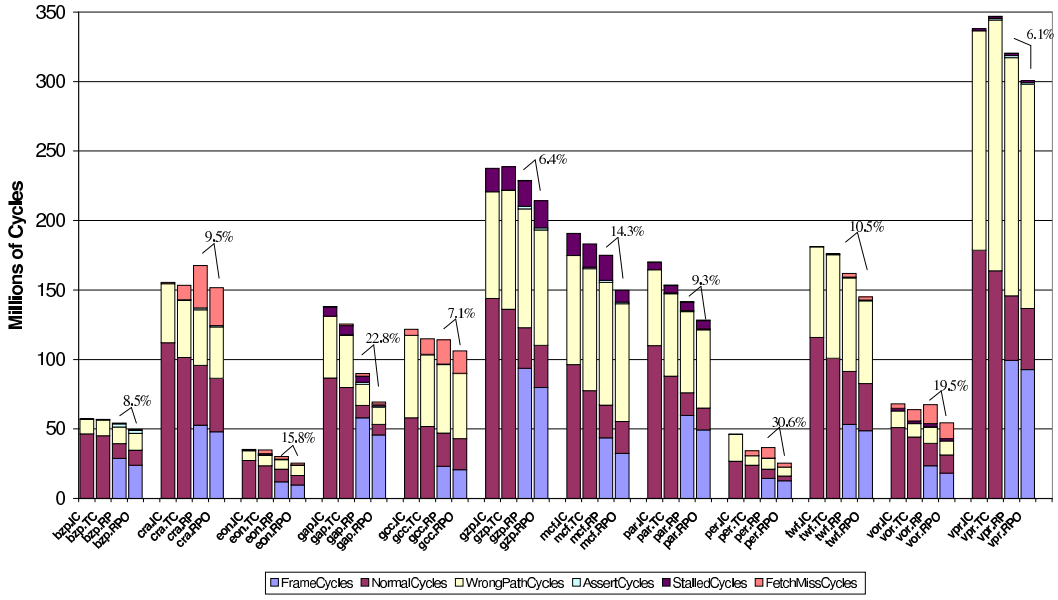

Figure 3. Performance of four superscalar processor systems: ICache, Trace Cache, rePLay, and rePLay with Optimizations on optimized binaries.

of dynamic instructions are in frames for the RPO configuration.

	IC	TC	RP	RPO
Inst/Fetch	5.28	5.94	6.69	6.55

Table 3. Average Fetch Size

5.2 Performance on unoptimized binaries

Figure 4 provides a comparison similar to Figure 3 for statically unoptimized binaries. The rePLay optimizer shows a remarkable performance improvement for most cases. Overall, RPO performs 15% better than RP (18% increase in effective IPC), 23% better than the TC configuration (30% increase in IPC), and 27% better than IC configuration (37% increase in IPC). On average, the rePLay optimizer reduces the dynamic instruction count of unoptimized binaries by 18%.

In two cases, RPO on unoptimized code is able to outperform statically optimized binaries running on the RP configuration (gap and perlbmk). With this comparison, the majority of the fetch bandwidth improvement is factored out and the result is mainly due to the strength of the rePLay optimizations. Though it appears that vpr also outperforms its statically optimized counterpart, it did not completely execute, rendering the comparison invalid.

Commercial software is often delivered unoptimized because of production deadlines, support issues, and issues with *fragile code*, or buggy code where flaws are exposed by compiler optimization. Such code is often due to accesses to uninitialized memory, out-of-bounds accesses, and timing and synchronization issues with threaded code.

While issues with fragile code are important for any optimization system, rePLay offers a safeguard against many types of fragile code, in particular those involving uninitialized and out-of-bounds accesses. The optimizations performed by rePLay are low-level and adhere to the semantics of the binary (as opposed to the semantics of the source code). Each atomic frame is a direct implementation of the underlying basic blocks; the resulting architectural state transformations performed by the frame are the same as those performed by the original code. There is a tradeoff: by exploiting semantic information in the source code, a static compiler is able to make powerful optimizations (note the difference in performance between the optimized and unoptimized binaries in Figures 3 and 4). These optimizations, however, are done at the expense of potentially breaking fragile code. The performance boost from the rePLay optimizations is more modest, but these optimizations are less likely to break fragile code.

5.3 Performance of individual optimizations:

Figure 5 is a plot of the relative impact of each class of optimization described in Section 3. In this plot, we enable

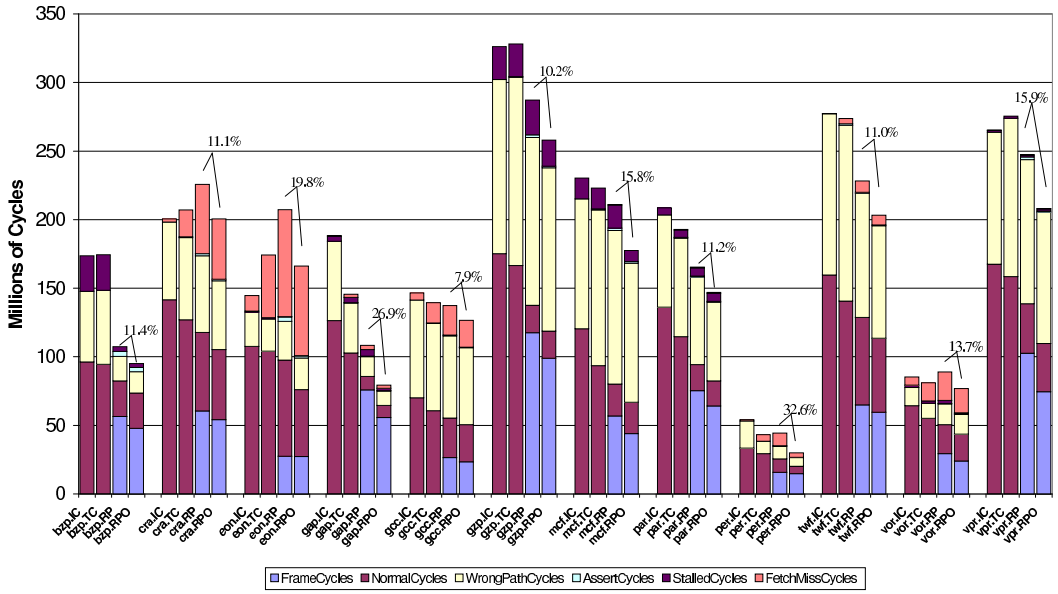

Figure 4. Performance of four superscalar processor systems on unoptimized binaries.

each optimization separately and plot its performance relative to both the RPO configuration (all optimizations are enabled) and the RP configuration (all disabled). The data were collected on the statically optimized binaries. A 1 on the y-axis represents the performance of RPO and 0 represents the performance of RP.

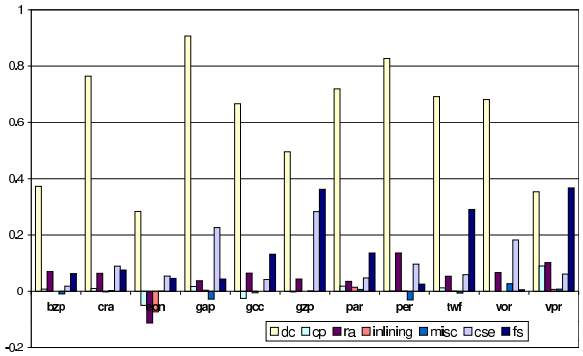

Figure 5. Performance of individual optimizations relative to the unoptimized rePLay configuration.

Of the individual optimizations, the dead code removal optimization has the most significant impact. Removing dead instructions, NOPs and unconditional branches eliminates the need to cache and fetch them.

Another single optimization of note is the branch fetch scheduling optimization. The individual performance of this optimization is 1.3% over basic rePLay, but can be as high as 6%. It is able to fetch branches on average 2.5 cycles earlier than without fetch scheduling, resulting in an average 3% decrease in the number of cycles for branch resolution.

Some optimizations, such as constant propagation and reassociation, provide little benefit in isolation, but provide substantial boosts when coupled with other optimizations. In associated experiments (not shown in this paper) we observed a synergistic effect of reassociation coupled with dead code and fetch scheduling. This combination provided almost 95% of the performance benefit of all optimizations.

6 Analysis

The previous section demonstrated the performance potential of the rePLay optimizer. In this section, we investigate the impact of implementation constraints on this performance potential.

6.1 Optimization latency and throughput

Figure 6 presents the effect of increasing the optimization latency relative to an optimizer that operates in 0 cycles. The latency of the optimizer is measured from the cycle an unoptimized frame arrives to the point the unoptimized frame is transmitted to the frame cache. This data was collected on the optimized binaries.

The latency starts to have an impact overall when it exceeds 10,000 cycles. In some cases, an increased optimizer latency improves performance. The increased latency causes the optimizer to delay the generation of a not-so-useful frame that evicts a more useful frame. At a latency of

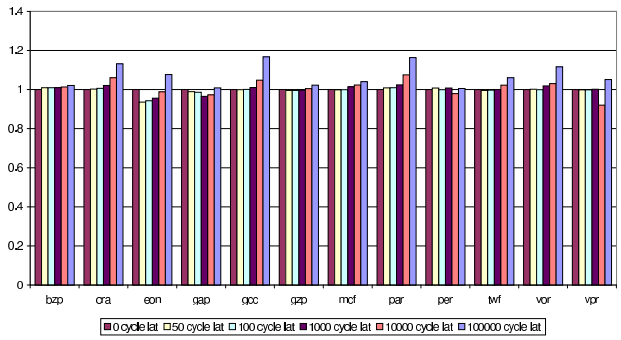

Figure 6. Slowdown relative to 0 cycle optimizer.

100,000 cycles, the net average slowdown over the 0-cycle optimizer is 7%, with a max slowdown of 17% for parser. As mentioned in Section 4.3, the typical optimizer latency is 1000 cycles.

Another important factor in the design of an effective optimizer is the throughput that the optimizer must sustain. For instance, if the optimizer operates at an approximate latency of 10 cycles per instruction in a frame and average frame size is 100 instructions, then the average frame occupies the optimizer for 1000 cycles. If the optimizer is not pipelined, then an incoming frame that arrives before the previous frame completes must be buffered or dropped. In our model, we assumed an ideal optimizer that could handle an unlimited number of in-flight frames.

In order to assess the throughput that the optimizer must support, we measured the average frame arrival rate: one frame every 110 cycles. While this rate presents a significant design challenge in the construction of the optimization engine, note that (1) many of the frames that are generated are never used (we quantify this in the next subsection), and (2) simple filter functions can eliminate the transmission of redundant and repeated frames to the optimizer. Our baseline (from which the 110 cycle number was derived) implements a very simple filter that drops a frame if the same frame was generated previously—this function eliminates approximately 1 out of 3 frames. Our future work involves a more detailed description of the optimization engine along with effective means to manage both its latency and its required throughput.

6.2 Frame Cache effects

Nearly a third of the frames that are generated by the constructor, optimized, and subsequently cached in the frame cache are never read. On average 28% of frames are dead in this sense. The benchmarks that suffer from high penalty due to cache misses (crafty, gcc, perl, vortex) also have a relatively high number of dead frames, on average 46%. An effective means of not constructing frames that are likely to be dead will boost frame cache efficiency and reduce the throughput requirements on the optimizer.

Frame cache efficiency is also impacted by the redundant nature of caching traces of the dynamic instruction stream. We measure redundancy in the frame cache by scanning the frame cache every 100k cycles during the execution and counting the number of unique basic block addresses stored in the cache. We find that on average 71% of the basic blocks cached in the frame cache are unique in any sample point. This ranges from 51% on gcc to 86% on bzip2. The results are similar for both optimized and unoptimized binaries. Nearly 30% of the instructions cached in the frame cache are duplicates. While techniques such as the block-based trace cache [2] can help control redundancy, different techniques must be adopted for rePLay because modifications by the optimizer alter the original basic blocks.

6.3 Frame length effects

In order to investigate whether longer frames are beneficial in boosting optimization potential, we examined performance while as a function of average frame size. We can control average frame size indirectly by decreasing the maximum frame size that the rePLay frame constructor is allowed to create. Figure 7 plots the relative effect on performance versus a relative drop in average frame size as measured by running each benchmark on the RPO configuration. The vertical axis represents the percentage of RPO performance where maximum frame size is 256 instructions. A benchmark configuration attaining 50% relative performance, for example, executes twice as long as the corresponding benchmark executing on a configuration where maximum frame length is 256 instructions.

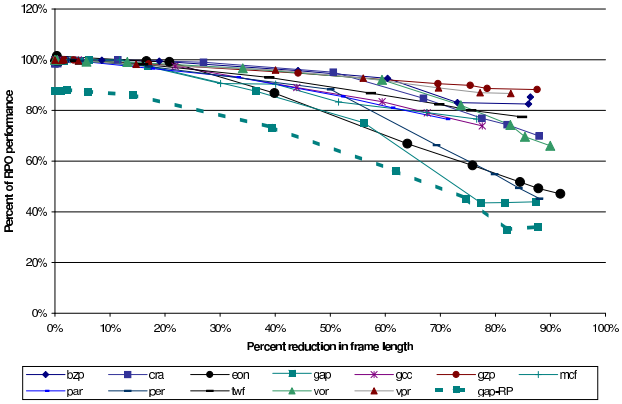

Figure 7. Performance versus frame length.

Obviously, fetch bandwidth is also a factor on performance as frame size is decreased. Also provided in Figure 7

is the effect on performance of the benchmark gap running on the RP configuration. Both trend lines for gap are emphasized. For the gap-RP trend line, the drop in performace is due purely to a drop in fetch bandwidth. When comparing gap versus gap-RP, one can deduce that the net drop frame size affects optimization potential.

7 Optimization Potential

To evaluate rePLay's success in exploiting optimization opportunities, we investigated the frequency of dead code in dynamic instruction streams. Processing a program's instruction trace offline, we broadly classify each instruction as either live or dead. Dead instructions contribute to neither control flow nor program output; their results are overwritten before they are used, or are used only by other dead instructions. Our study is similar to Rotenberg's measurements of ineffectual instructions [22], but focuses on the potential for instruction removal and on the difficulty of dynamic identification, whereas [22] studies parallel execution of ineffectual regions of code identified through profiling. We calculate optimization potential offline and explore dynamic identification of dead instructions.

To recognize a dead instruction, the rePLay optimizer must verify that all uses of the instruction's result are eventually discarded. As the optimizer sees only instructions within a frame, this requirement implies that the frame contains the dataflow graph comprising all uses of the instruction as well as all overwrites of its results. We term the number of instructions that must be visible after a dead instruction in order to classify it as dead as the *apparent lifetime* of the instruction. Even if a dead instruction's apparent lifetime lies within the scope of a frame, the optimizer may not have adequate memory aliasing information to recognize the instruction as dead.

Offline processing provides an upper bound on the potential of finding dead instructions dynamically. Program traces are processed in reverse order to reduce the complexity of classification. When processing a particular instruction, liveness and apparent lifetimes of all consumers of the instruction's result are already known.

The total live instruction counts are slightly conservative (high): as the simulator only emulates traps/system calls, we assume that any trap reads all registers and memory locations. Also, at the end of a program, all registers are considered live and all memory dead. Assuming that traps consume neither registers nor memory reduces the number of live instructions by an average of only 2.5%, thus this conservative bound is reasonable.

Results for the twelve benchmarks appear in Figure 8. In the figure, we separate dynamic instructions into nine categories, three live and six dead. The live categories are data operations, control instructions, and control operations. Data operations include all instructions that contribute directly to a program's output, *i.e.*, the dataflow trees that produce memory or register values consumed by traps. Control instructions and operations contain instructions that contribute only to control flow (and thus indirectly to program output); instructions are the actual control instructions, and control operations are non-control instructions that contribute to conditions, indirect jump targets, *etc.* The relatively small fraction of data operations relative to control may reflect algorithmic tradeoffs in the benchmarks; in particular, programmers often reduce the number of data operations at the expense of more complex control.

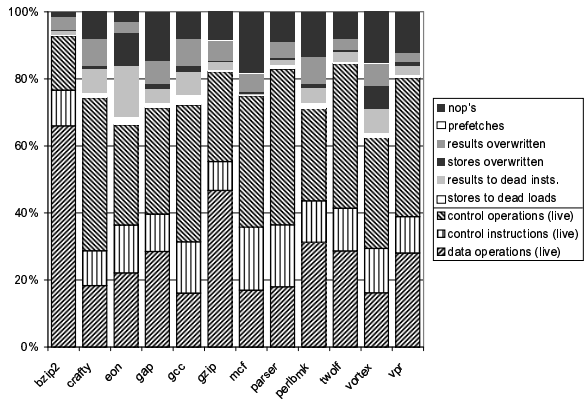

Figure 8. Breakdown of live and dead instructions. Dead categories are solid, and live ones are textured.

The six categories of dead instructions are divided roughly according to the difficulty of identifying them as dead during execution. Because of the relative storage capacities, dead values stored in memory tend to live longer than those stored in registers. Three of the six dead categories have the potential for memory storage, and thus present roughly the same difficulty for dynamic identification. Store operations used only by dead loads are the most difficult, as values must not only be tracked through their lifetime in memory, but any (dead) dataflow structures based on loads of the values must also be traced. Instructions with results used by dead instructions present a similar difficulty if their consumers include dead stores. Stores overwritten before use are the easiest of the three categories, but may still require a fairly large window to identify.

The next category, results overwritten, includes instructions that write their results into registers that are overwritten before they are used. Relative to previous categories, such instructions have small apparent lifetimes. Prefetches, the next category, make up only an insignificant part of the dynamic instruction stream (at most 0.3% across all benchmarks). As these instructions may be beneficial, there is no

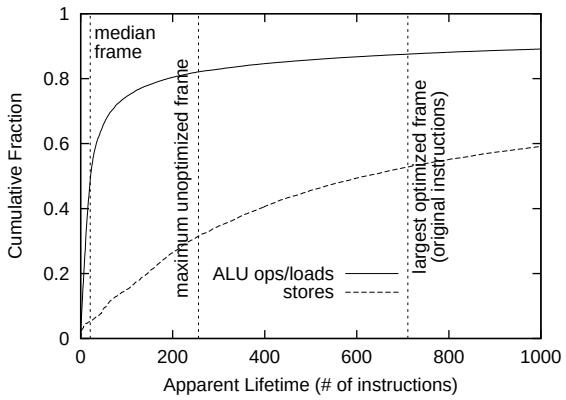

Figure 9. Cumulative distribution of apparent lifetime for dead instructions in gcc. Dead stores are longer-lived and harder to identify than instructions that write to registers.

point in removing them. NOPs account for roughly one in ten instructions in optimized code.

The proportion of dead instructions is striking, averaging 24% (from 7.1 to 37%) across the benchmarks. With gcc, for example, Figure 8 indicates that approximately 15% of dynamic, result-producing instructions in gcc are dead. For such instructions, rePLay's dead code removal eliminates only about 10.7% of dynamic instructions, however. Similarly, 4.9% of dynamic instructions are dead stores, but rePLay eliminates less than 0.038%.

Two factors account for the bulk of these discrepancies: apparent lifetimes can be long compared with frame lengths, and address disambiguation further hampers dead store identification (rePLay's dead code optimization does not even attempt to remove dead stores). As an example of this effect, consider Figure 9, which plots the cumulative distributions of apparent lifetime in gcc for dead stores and for dead ALU operations and loads. Although only a small fraction of dead registers cannot be identified within a large frame, the median frame size for gcc is only 21 instructions. By summing the probability of identifying each instruction within a frame as dead over all instructions in a frame of a particular size, and then summing the results weighted by a program's frame size distribution, we can more accurately estimate the potential for dead code elimination with rePLay. Based on this calculation for gcc, rePLay can identify at most 4.0% of dynamic instructions as dead, result-producing instructions, and 0.19% as dead stores. Dead code thus achieves 53% of its potential, and dead store removal roughly 20%. The difference in these achievements is rooted in the address disambiguation problem.

8 Related Work

The rePLay Framework bears resemblance to the generalized Trace Processor model [23] in that both microarchitectures are oriented towards the execution of long sequences of the dynamic instruction stream. In rePLay, frames are necessarily atomic to facilitate dynamic optimization. This fact allows far more aggressive optimization over an architecture that does not guarantee atomicity.

Also, rePLay represents an effective implementation of a general Instruction Path Co-Processor, or ICOP [8, 12, 4]. ICOP frameworks provide programmable hardware support for trace formation and dynamic optimization. A few, preliminary investigations into hardware support for dynamic optimization have been made [6, 17, 8, 4].

The notion of a frame is similar to other types of optimization regions, such as superblocks, hyperblocks [14], and traces (from trace scheduling) [7]. It is different in the notion that recovery is relegated entirely to hardware.

Almost all of the previous work on dynamic optimization has centered around software systems where the dynamic optimizer is part of the run-time system [13, 1, 10]. For many schemes, such as Dynamo [1], the original program runs under the control of a software interpreter. The interpreter gathers information about the program's run-time behavior and builds optimized regions. When a PC is encountered for which an optimized region exists, the optimized code is directly executed.

9 Conclusion

We have evaluated the rePLay microarchitecture as means for reducing application execution time by facilitating effective dynamic optimization. The framework contains hardware support for dynamic optimization, in the form of a programmable optimization engine and a recovery mechanism. The optimizer decreases optimization overhead by allowing optimization to occur concurrently with execution and with potentially lower latency. The recovery mechanism enables the optimizer to make speculative optimizations without the necessity of generating recovery code, potentially increasing the aggressiveness of the optimizations.

We find that, when compared to a rePLay configuration not performing optimization, the rePLay optimizer can reduce the number of execution cycles for the SPEC2000 integer benchmarks by an average of 13% on Alpha binaries already optimized by a compiler (resulting in a net effective increase of 16% in IPC), and by 15% on binaries that are not statically optimized (18% increase in IPC). Furthermore, rePLay with optimizer reduces execution cycles by 21% over an ICache and 18% over a Trace Cache when binaries are statically optimized, and 27% and 23% when they are not.

We find that the rePLay optimizer can operate effectively with optimization latencies up to the 10k cycle range, but must support a fairly high throughput. Throughput reduction techniques are possible, as nearly 30% of frames generated are never executed.

One major benefit of the optimizer comes from removing dead code, and on average, it is able to reduce dynamic instruction count by 11%. We investigated the potential for dead code elimination in optimized binaries and compared rePLay's results with that potential, concluding that rePLay realizes a substantial fraction of the optimistically-bounded potential: 50% of dead, result-producing instructions and 20% of dead stores. The limiting factor in realizing the immediate potential is disambiguation of memory addresses.

10 Acknowledgments

This work was supported in part by NSF CAREER grants NSF-CCR-00-92740 and NSF-ACI-99-84492 and by the C2S2 MARCO Center. We gratefully acknowledge the input of the other ACS members, and in particular Michael Fertig and Gregory Muthler. We also thank Advanced Micro Devices, Hewlett-Packard, and Intel for their generosity in providing equipment.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical Report HPL-1999-78, Hewlett-Packard Laboratories, June 1999.

[2] B. Black, B. Rychlik, and J. P. Shen. The block-based trace cache. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 196–207, 1999.

[3] R. Chappell, J. W. Stark, S.-W. Kim, S. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.

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

[5] M. M. Crum. Design alternatives for caching long regions of the dynamic instruction stream. Master's thesis, University of Illinois at Urbana-Champaign, 2001.

[6] K. Ebcioğlu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[8] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, 1998.

[9] B. Grant, M. Mock, M. Phillipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. Technical Report UW-CSE-97-03-03, University of Washington, May 1999.

[10] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. Dynamic and transparent binary translation. *IEEE Computer*, 33:54 – 59, Mar. 2000.

[11] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. *International Journal of Parallel Programming*, 26(4):449–478, Aug. 1998.

[12] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999.

[13] A. Klaiber. The technology behind Crusoe processors. Technical report, Transmeta Corporation, Jan. 2000.

[14] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, 1992.

[15] S. Melvin and Y. Patt. Enhancing instruction scheduling with a block-structured ISA. *International Journal of Parallel Programming*, 23(3):221–243, June 1995.

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

[17] R. Nair and M. E. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, 1997.

[18] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[19] S. J. Patel and S. S. Lumetta. rePLay : a hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):300–318, June 2001.

[20] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, 2000.

[21] A. Ramirez, J. L. Larriba-Pey, and M. Valero. Trace cache redundancy: Red and blue traces. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, 2000.

[22] E. Rotenberg. Exploiting Large Ineffectual Instruction Sequences. Technical report, North Carolina State University, November 1999.

[23] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.