rePLay: A Hardware Framework for Dynamic Optimization

Sanjay J. Patel, *Member*, *IEEE*, and Steven S. Lumetta, *Member*, *IEEE*

Abstract—In this paper, we propose a new processor framework that supports dynamic optimization. The rePLay Framework embeds an optimization engine atop a high-performance execution engine. The heart of the rePLay Framework is the concept of a *frame*. Frames are large, single-entry, single-exit optimization regions spanning many basic blocks in the program's dynamic instruction stream, yet containing only a single flow of control. This atomic property of frames increases the flexibilty in applying optimizations. To support frames, rePLay includes a hardware-based recovery mechanism that rolls back the architectural state to the beginning of a frame if, for example, an early exit condition is detected. This mechanism permits the optimizer to make speculative, aggressive optimizations upon frames. In this paper, we investigate some of the underlying phenomenon that support rePLay. Primarily, we evaluate rePLay's region formation strategy. A rePLay configuration with a 256-entry frame cache, using 74KB frame constructor and frame sequencer, achieves an average frame size of 88 Alpha AXP instructions with 68 percent coverage of the dynamic istream, an average frame completion rate of 97.81 percent, and a frame predictor accuracy of 81.26 percent. These results soundly demonstrate that the frames upon which the optimizations are performed are large and stable. Using the most frequently initiated frames from rePLay executions as samples, we also highlight possible strategies for the rePLay optimization engine. Coupled with the high coverage of frames achieved through the dynamic frame construction, the success of these optimizations demonstrates the significance of the rePLay Framework. We believe that the concept of frames, along with the mechanisms and strategies outlined in this paper, will play an important role in future processor architecture.

Index Terms—High-performance microarchitecture, dynamic optimization, trace caches.

1 Introduction

Dynamic optimizations techniques are rapidly gaining the attention of computer systems researchers as an effective means for boosting application performance. Code optimizations made dynamically (i.e., while the program is running) can exploit the stable, possibly phased [26], behavior exhibited by an application during execution 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 more amenable to dynamic optimization.

In this paper, we propose a hardware framework that supports dynamic optimization. The rePLay Framework is a microarchitecture that combines a high-performance execution engine with a programmable optimization engine, allowing program optimization to occur simultaneously with program execution. Coupled with the execution engine is a hardware recovery mechanism that enables the optimization engine to perform optimizations *speculatively*.

RePLay consists of five key components:

1. A component called the frame constructor for constructing candidate code regions for optimization,
2. A programmable optimization engine for optimizing these regions,
3. A frame cache for caching these regions on-chip,
4. A component for sequencing between regions, and
5. A mechanism to recover architectural state if speculatively applied optimizations prove incorrect.

These components are integrated into a processor's fetch and execution engine, as shown in Fig. 1.

In rePLay, the candidate regions of optimization are called *frames*. They are the soul of rePLay. They serve the same purpose that traces within a trace scheduling compiler or superblocks within an aggressive ILP compiler serve. They are the scope in which optimizations are applied.

Frames are *logically atomic*. They have a single entry point and a single exit point (execution of a frame starts at a single instruction and ends at a single instruction). Furthermore, frames encapsulate only a single flow of control. If any instruction within a frame executes, all instructions within the frame execute. A basic block is an example of a frame, albeit a small one. This atomicity property provides more flexibility for applying optimizations than if the frame were not atomic: Instructions within a frame are not control dependent on one another and can be moved freely within the frame within the confines of data dependencies. Atomicity also reduces the complexity of the optimization algorithms. These algorithms need not consider multiple paths of execution as there are no side exits, side entrances, or divergent flows of control within a frame.

To support frame atomicity, the rePLay Framework includes a hardware recovery mechanism that reverts the architectural state to the beginning of a frame when it is detected that a frame does not completely execute (e.g.,

• *The authors are with the Center for Reliable and High-Performance Computing, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: {sjp, steve}@crhc.uiuc.edu.*

Manuscript received 7 Aug. 2000; revised 31 Jan. 2001; accepted 15 Feb. 2001. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 113691.

0018-9340/01/$10.00 © 2001 IEEE

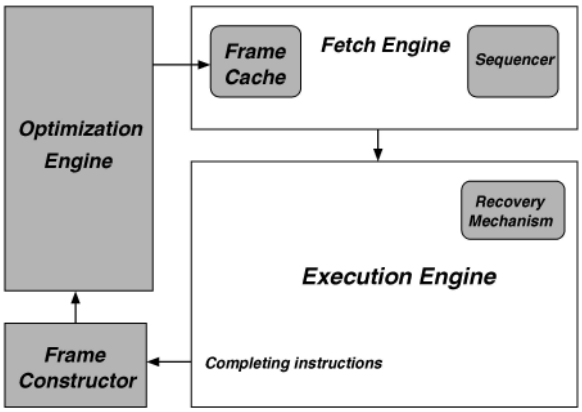


Fig. 1. The rePLay Framework integrated into a generic processor microarchitecture.

because an early exit condition was detected). This enables the rePLay mechanisms to join together and aggressively optimize basic blocks that are likely to execute together into a single frame without generating the recovery code necessary for several other approaches to optimization. In order for frame-based optimizations to have a significant impact on performance, frames must consist of many instructions and span multiple basic blocks. The longer the frame, the greater the opportunity for optimization and the greater the boost to overall fetch bandwidth.

In this paper, we make several contributions. First, we describe the rePLay Framework, including the mechanisms to create frames, to optimize frames, to sequence between them, and to recover when frames are incorrectly initiated (Section 2). Second, we describe and evaluate an effective dynamic technique for creating long frames that have good caching properties and very low likelihood of requiring recovery action (Sections 4, 5, and 6). Third, we present a sampling of optimizations that are possible within the rePLay Framework, many of which are most effective when based on runtime behavior rather than the limited behavioral information available to a static optimizer (Section 7).

2 The rePLay Framework

Dynamic information is beginning to play a significant role in boosting processor performance. Mechanisms such as branch prediction, dynamic scheduling, and hardware memory disambiguation are central to high-performance processing today. Techniques such as trace caches, value prediction, and instruction reuse are likely to appear in tomorrow's processors. All of these techniques leverage the stable patterns that occur during execution to reduce a program's running time.

Run-time behavior may exhibit stabilities that are hard to identify or hard to exploit statically. A compiler can very effectively capitalize on behavior that is stable across an entire execution, such as highly biased branches apparent from profile executions of a program. However, if the program behaves differently from the way the compiler expected, optimizations based on expected behavior may degrade performance. Similarly, phased variations in behavior during execution—for example, a conditional branch that is highly biased for the first half of the program and then highly biased in the other direction for the second half—are difficult for a compiler to exploit. Furthermore, without addressing the cumbersome task of generating recovery code or relying on hardware support for speculation, compilers can reap only limited benefits from behavior that is typical but not universal.

The rePLay Framework dynamically identifies stable runtime behavior and exposes this behavior to a hardware-based optimizer that performs lightweight code optimizations on sections of the dynamic instruction stream. The optimizer operates on regions called *frames* that consist of many basic blocks in the original control flow. These regions are based on the original control flow of the program and are created by a hardware frame constructor. Once optimized, these frames are stored in a hardware cache for low-latency access. While the program executes, a correlation-based frame sequencer decides appropriate instances for frame dispatch. Coupled with the frame execution mechanism is a hardware-based recovery mechanism that reverts architectural state to the beginning of a frame if it is detected that a frame should not have executed. With this hardware recovery mechanism, the optimizer can make speculative, potentially unsafe, optimizations that are based on assumptions about control flow or data values. This section provides a more detailed overview of each component and subsequent sections explore the specific design trade-offs and experiments that motivate rePLay.

2.1 Frame Speculation and Recovery

The central concept of rePLay is the concept of the atomic region or *frame*. A frame is similar to a *trace* in a trace-scheduling compiler [6] or a *block* in the Block-Structured ISA [10], [18]. As explained later, 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 execute or none of them do. This atomicity simplifies both frame scheduling and the optimization algorithms used for dynamic optimization.

Control instructions within a frame are changed to *assertion* instructions. An 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, selecting either the taken target of the branch instruction if the condition is true or the instruction following the branch if the condition is false.

In contrast, an assertion has no effect on the address of subsequent instructions and has no effect whatsoever if the condition is true. If the condition is false, the assertion fires, triggering a recovery action that discards all instructions in the frame and redirecting control flow to the original address for the instruction at the beginning of the frame. Instructions after a given assertion can thus be executed speculatively, making the assumption that the assertion condition is true.

When an assertion fires during a frame's execution, the hardware must roll architectural state back to the beginning

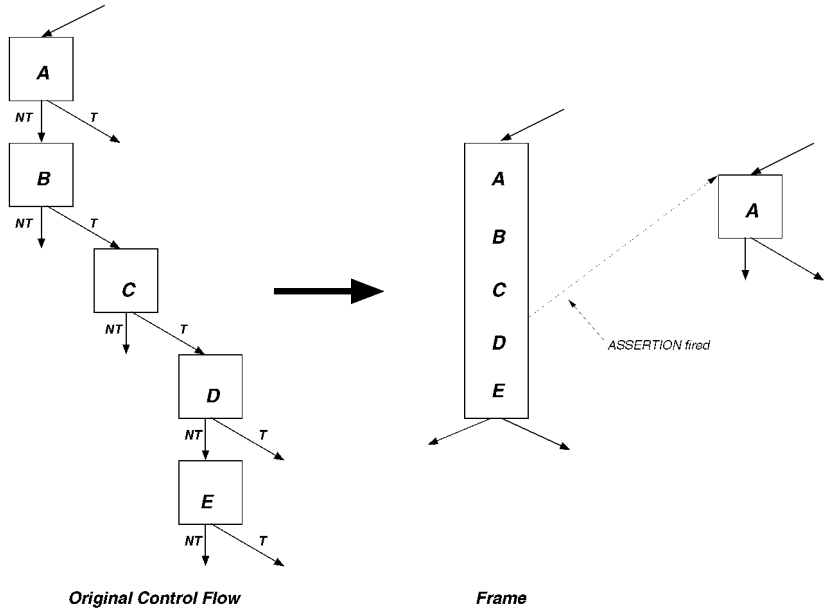


Fig. 2. Construction of a frame. Although the original code spans several branches, the resulting frame has only a single entry and a single exit.

of the frame. Such restoration implies that any state generated during frame execution must be buffered until all assertions within the frame have executed successfully (not fired). Once all assertions within a frame have been checked, state changes generated within the frame can be committed.

Buffering of architectural state in rePLay is accomplished using recovery mechanisms similar to those used by dynamically scheduled processors with speculative execution. As required by these processors, we require a means to commit register values and memory writes only once the associated computation is known to have completed without exception and with no intervening misspeculations.

RePLay requires 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 it is known that their associated frame commits, at which point all values that are live-out of the frame proceed to the architectural register file in a single cycle. If the frame does not commit, all values corresponding to the frame (and all values from subsequent frames) are flushed from the reorder buffer, also in a single cycle. This recovery action is similar to that required for a branch misprediction. Because of the potentially high number of values that are in-flight while executing a frame, rePLay's register recovery mechanism requires a deeper buffer.

Similarly, store values are kept in a pending store buffer until the corresponding frame is committed. Again, a dynamically scheduled processor utilizes a similar mechanism to recover from branch mispredictions, allowing misspeculated stores to be flushed in a single cycle. The corresponding rePLay mechanism requires a larger number of store values to be buffered: potentially, one for every store instruction in a frame. Alternatively, the number of store instructions in a frame can be limited during construction to accommodate the physical size of the pending store buffer.

2.2 Frame Construction

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 many basic blocks. Long frames increase the potential for finding optimization opportunities not exploited at compile time. As mentioned previously, branches and other control instructions at the boundaries between basic blocks in a frame are converted to assertion instructions. In rePLay, branches are converted via a technique called *branch promotion*.

With branch promotion [21], branches that behave in a highly regular manner are dynamically promoted to assertions. Many different schemes for promoting branches to assertions are possible. In this paper, we investigate a hardware-based scheme in which branches that have gone to the same target for n consecutive previous occurrences are candidates for promotion. The promotion process can be applied not only to conditional branches, but also to indirect branches and return instructions by generating appropriate assertion conditions based on the expected target addresses.

Our frame construction algorithm is quite simple: 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 that is not promoted into an assertion, the pending frame is terminated. If a newly formed frame is large enough, it 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.

Fig. 2 shows how original control flow is reorganized into a frame. For the frame ABCDE, a firing assertion causes program control to return to the original block A. In such a

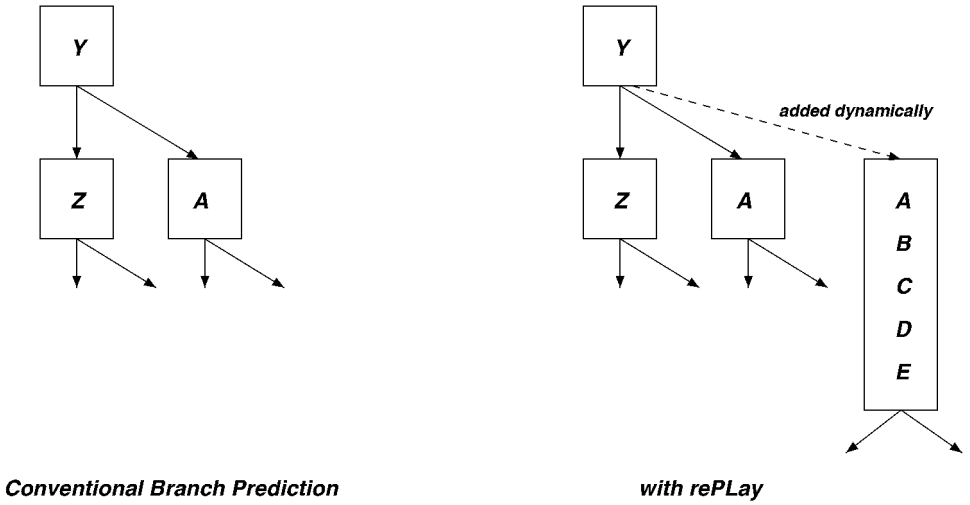


Fig. 3. Two versions of control flow originating from the same static branch. The number of targets for a rePLay sequencer varies dynamically, even for the same static branch. When frame ABCDE is created, it becomes a target for the prior block, Y. The sequencer decides when ABCDE is selected rather than one of the static targets.

case, no instruction in the frame is committed to architectural state.

Frames may contain taken branches from the original code. In other words, the frame construction process dynamically remaps nonsequential control flow into straight-line code in a manner similar to trace construction within a trace cache mechanism or the dynamic code realignment mechanism proposed by Merten et al. [19]. The frame ABCDE in the example above is stored as a sequential block of instructions despite the presence of nonsequential blocks in the original code.

2.3 Optimization Engine

Frames generated by the frame constructor are dynamically optimized by an optimization engine. The optimization engine is a flexible datapath that can be software-programmed using its own native instruction set architecture and separate local memory. The optimization engine also has access to the execution state (including micro-architectural state) of the program being optimized, such as branch behavior, load-store dependence information, or to intermediate data values.

Typical optimizations performed by the optimization engine include classical compiler optimizations, extended basic block optimizations [11], and various optimizations performed by other dynamic optimizations systems [1]. The programmability of the optimizer allows for optimizations to be tailored toward an application or toward a particular section of code.

Furthermore, the coupling of dynamic optimizations, execution rollback mechanisms, and rePLay's assertion instruction architecture allows for low-latency implementation of speculative optimizations: optimizations that may not be valid in every operating scenario. For example, an optimization that speculates on input data values can be performed. These optimizations prespecialize a frame for particular input values (or subset of input values) that are detected to be stable at runtime. For each assumed value, a data assertion is added to the frame to ensure the live-in value is the expected value. This type of optimization draws upon the same phenomenon that drives value prediction and computation reuse [4], [27]. Examples of both simple optimizations and aggressive value speculation for pointer aliasing appear in Section 7. Other classes of possible optimization include optimizations that tune the instruction stream to the details of the execution microarchitecture, such as instruction scheduling and instruction placement for clustered functional units.

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 except that it delivers very long sequences of instructions, spanning multiple traditional cache lines. For example, a particular frame might consist of 80 instructions, span five cache lines (at 16 instructions per cache line), and take five cycles to be fetched and issued on a 16-wide fetch/issue processor. The frame cache must support frames of varying sizes 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. This caching mechanism is described in more detail in Section 6.

2.5 Sequencer

The sequencer has the difficult task of chaining the fetch mechanism between one frame and the next. As frames are created dynamically, they must be added to the fetch stream. As is the case with the fetch engines of all pipelined processors, the sequencer operates ahead of the execution engine and, thus, must speculatively select frames for fetching. The penalty associated for incorrectly initiating an optimized frame may be quite severe, depending on the depth of the firing assertion within the frame's dependency tree. The rePLay Framework uses a speculative sequencing mechanism to predict when a frame should be initiated and when it should not. For example, consider Fig. 3. At block Y in the original control flow, there are two choices: block Z and block A. A conventional branch predictor selects one of the two targets based on information collected about past

program behavior. With rePLay, a third choice is possible: the dynamically constructed frame ABCDE. The rePLay sequencing mechanism consists of a conventional branch predictor, which selects between A and Z, and the frame sequencer, which selects between the fetch of block A or block Z versus an initiation of the frame ABCDE. Section 5 examines this mechanism in more detail.

To sustain adequate instruction fetch bandwidth, the rePLay Framework uses an instruction delivery mechanism consisting of a standard instruction cache or trace cache to supplement the frame cache. Based on the sequencer mechanism described above, the fetch address is directed to either the conventional caches or to the frame cache. If it is directed to the frame cache and the frame cache responds with a miss, the conventional caches are accessed in a subsequent cycle. If the frame cache responds with a hit, a frame streams out of the frame cache in fetch-width sized packets over the next few cycles.

3 Related Work

Almost all of the previous work on dynamic optimization has centered around software systems where the dynamic optimizer is part of the runtime system [15], [1], [8]. For many schemes, such as Dynamo [1] and Transmeta's Code Morphing System [15], the original program runs under control of a software interpreter. The interpreter gathers information about the program's runtime behavior and builds optimized regions. When a PC is encountered for which an optimized region exists, the optimized code is directly executed.

All software schemes suffer from diminished gains due to optimization overheads. Essentially, while the optimizer is running, the application is not and, therefore, the latency of the optimizer (and interpreter, if it exists) is *exposed* in the execution time of the application. Furthermore, probing latency is incurred in order to collect runtime information. Also, the entire optimization system uses the same caches and translation lookaside buffers as the running program, exacerbating memory systems performance. RePLay offers the potential of reducing this overhead by moving the portions of dynamic optimization process into hardware. Furthermore, rePLay's hardware recovery scheme enables the optimizer to perform speculative optimization without requiring recovery code.

A few, preliminary investigations into hardware support for dynamic optimization have been made [3], [5], [7], [20]. Because of the limited scope (i.e., small regions) to which the optimizations were applied in these previous studies, many hardware optimization schemes showed only modest gains.

RePLay is an evolution of these hardware schemes. It builds upon several recent developments in computer microarchitecture, specifically that of trace caches, correlation-based branch prediction, value prediction, and computation reuse, in order to enlarge the scope of optimization and to create opportunities for speculative optimization. The programmable optimization engine has the potential of allowing the optimization techniques used by software-based dynamic optimizers to be applied with less overhead.

Much of this work builds upon previous trace cache research [24], [25], [22], in particular, that of branch promotion [21] and that of dynamic trace optimizations [7], [14], [3], and also that of the trace predictor [13]. Similar in nature to this work is the DIF cache [20], which dynamically creates statically scheduled instruction words.

Frames bear resemblance to other types of optimization regions. Superblock [9] and hyperblock formation [16], and trace scheduling [6], collapse sequences of frequently executed blocks into a straight-line, possibly predicted, code entity, relegating the infrequent blocks to side exits. Frames are different in that they contain no side exits and no internal control flow. Furthermore, the hardware recovery mechanism allows frames to be scheduled and optimized without requiring recovery code for misspeculations.

While not examined in this study, the frame construction hardware can direct its efforts to hot regions of execution, thereby focusing the efforts of the optimizer to frames that have high probability of repeated execution. Merten et al. [19] have demonstrated how hot-spot detection can effectively boost instruction fetch bandwidth by dynamic code realignment of hot execution regions.

4 Frame Construction Techniques

A critical component of the rePLay Framework is the frame construction mechanism. The objective of frame construction is to create atomic regions consisting of many instructions that are very likely to execute completely. The resulting single-entry, single-exit, single-path semantic allows the runtime optimizer to perform very aggressive optimizations upon these constructed frames.

Frames can span many basic blocks of a program. To prevent branches that terminate these blocks from also terminating frames, the rePLay frame constructor dynamically converts each highly biased branch into an assertion that generates a recovery action if the original branch switches direction. See Fig. 2 for an example. The target block of the original branch appears inline with the assertion. Furthermore, because of rePLay's hardware-assisted recovery mechanism, instructions within a frame are not control dependent on any of the frame's assertions. The promotion of branches to assertions can be applied to all types of multitarget branches (i.e., conditional, indirect, returns).

The frame construction mechanism is simple: Instructions are added to the frame constructor as they retire. The pending frame continues to grow as long as branches are promoted. Once a nonpromotable branch is encountered, the pending frame is considered complete and is handed off to the optimizer.

The promotion mechanism is also simple: The constructor accesses an entry for each completing branch in a *branch bias table*. The bias table is a hardware structure that counts the number of consecutive times a branch has had the same outcome. Each time a branch has the same outcome as previously, the counter field of the bias table entry is incremented. Once the counter reaches a threshold, the branch is promoted into an assertion. In other words, the bias table promotes a branch if it has n (where n is the counter threshold) outcomes in the same direction. Previous

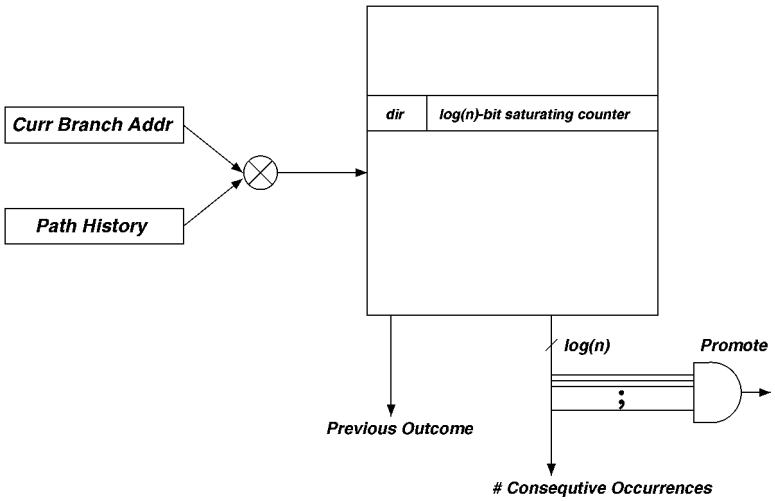


Fig. 4. The bias table mechanism for Branch Promotion with promotion threshold *n*.

branch prediction research has demonstrated that such a technique is an effective way to identify highly biased branches [2].

We also *demote* assertions back into branches when we detect that their behavior has changed. As every assertion is committed, it also is checked in the branch bias table (using its original basic block address) to determine if it should be demoted. While many demotion schemes are possible, we adopt one in which an assertion is demoted when it fires twice consecutively. This policy allows a branch at the end of a loop to remain as an assertion even though the singular fall through case causes the assert to fire. A demoted assertion causes the frame containing it to be invalidated in the frame cache.

The frame constructor's ability to identify candidate branches for promotion is greatly enhanced with the use branch correlation, as shown later. Fig. 4 is a diagram of the branch bias mechanism augmented with path history. Path history is XORed with a branch's address in order to generate the bias table index. This way, a specific static branch is divided into multiple instances based on the control path leading up to the branch.

The starting path history of each frame (i.e., the committed history at the first block in the frame) is kept with each frame. This history is essentially a prefix that identifies the instance of each promoted branch within a frame. For example, if the frame history of frame ABCDE is XYZ, XYZ was used to decide whether or not to promote branch A, YZA was used to decide the promotion of B, and so forth. The starting history XYZ forms a *context* for the frame and specifies when the frame should be invoked. Whenever the current history contains XYZ and the current fetch address is A, the frame sequencing mechanism attempts to fetch the frame ABCDE.

The crux of this frame construction technique hinges on the observation that a branch can be separated into instances based on the path leading up to the branch. Once separated this way, a greater number of branches tend to exhibit biased behavior. This is the same phenomenon exploited by two-level branch predictors. Said another way, the outcome of a branch tends to be highly correlated to the outcomes of branches, or path, before it. The history used in the promotion decision helps separate branches into these biased instances.

Larger frames are beneficial for the rePLay Framework. Larger frames spanning more basic blocks present the rePLay optimizer with a greater opportunity for performing effective optimizations. Larger frames also increase the processor fetch mechanism's ability to supply instructions at a high rate. For the measurements presented in the remainder of the paper, the frame constructor only maintains frames containing five or more dynamic basic blocks or containing 32 or more instructions. Smaller frames are discarded. Also, to limit the space required for a frame in the hardware frame cache, each frame has a maximum limit of 256 instructions.

4.1 The Ideal

In this section, the rePLay frame construction mechanism is evaluated on three bases: frame length, coverage of the instruction stream, frame completion rate.

Fig. 5 demonstrates the potential of this frame construction technique by showing average frame size in instructions on the eight SPECint95 benchmarks. All measurements were taken on a trace-driven simulator (based on the SimpleScalar tool set) simulating the Alpha AXP ISA. All benchmarks were compiled using the Compaq/Digital C Compiler V3.5 at the maximum optimization level (including loop unrolling). Profile-guided code placement was used to reduce the impact of taken branches. Also, link-time code placement optimizations using the OM executable editor were performed to further improve the code-layout. In [23], we present a more condensed version of the measurements presented in this paper on the SPEC2000 benchmarks.

Fig. 5 demonstrates the average size of frames in instructions using a branch bias table of unlimited size (i.e., interference-free). The horizontal axis of the graph represents the number of branch targets incorporated into the path history. The path history is used to index into the branch bias table to determine whether a branch should or should not be promoted.

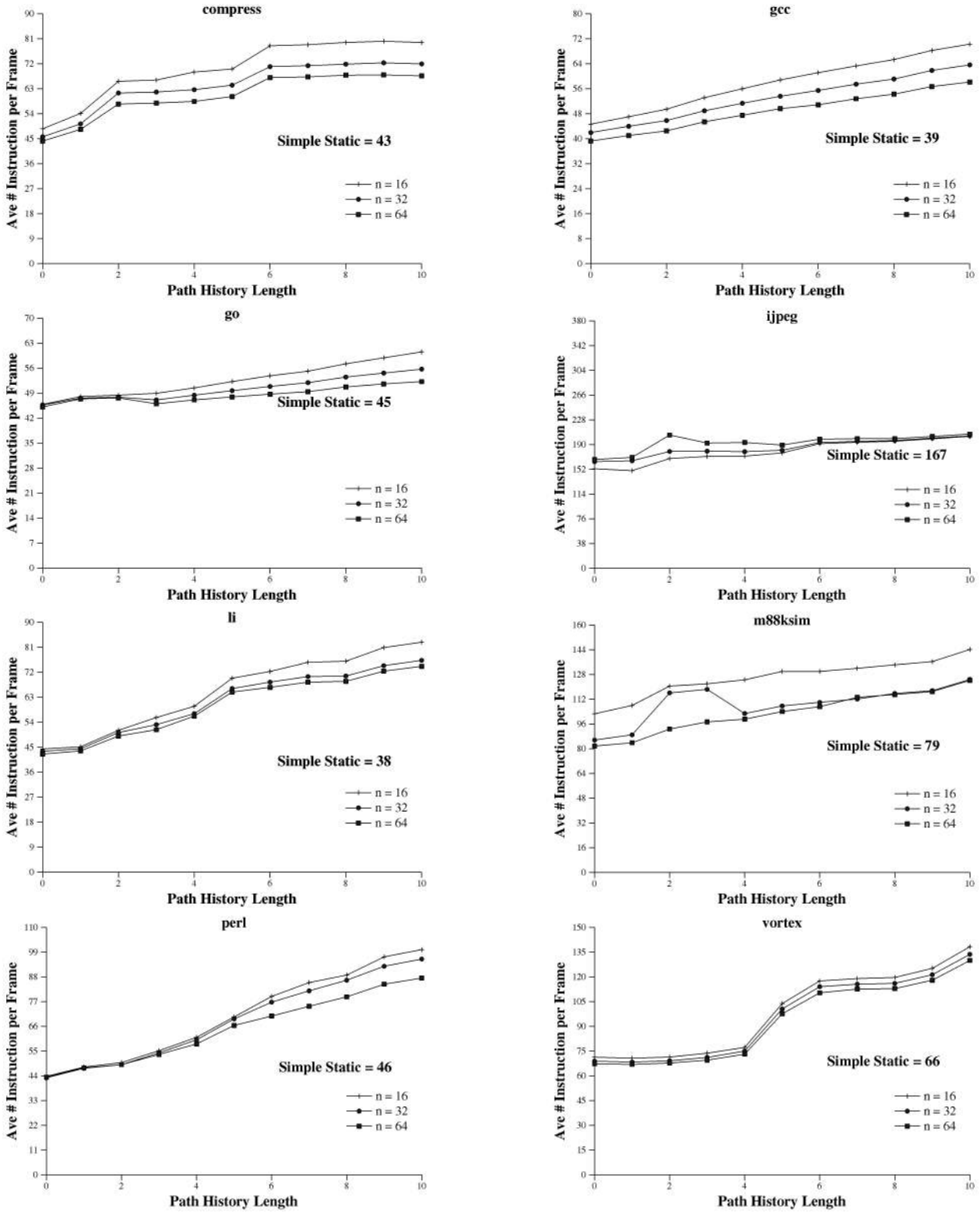


Fig. 5. Average frame size gathered using an interference-free bias table indexed with a given path history length. The three lines (n = 16, 32, 64) correspond to different promotion thresholds. For each benchmark, a simple static number is provided to show static frame length obtained by best-case compiler analysis.

Also shown on these graphs is the average frame size if a simple static predictor were used in place of the dynamic bias table. Here, the static mechanism classifies a branch as promoted if it has a 95 percent bias toward a particular target during the profile run. The mechanism is idealized because the measurement data set is the same as the profile data set. All branches are considered to be promotable, including indirect jumps and returns.

The data presented in these graphs is promising. Whether generated by a static means or by using the

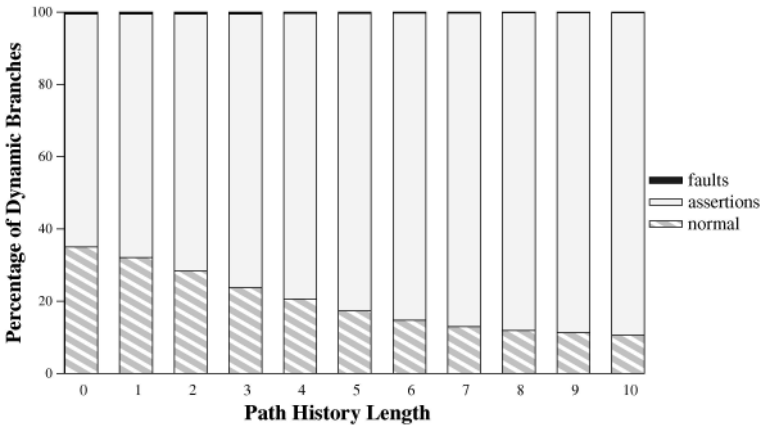


Fig. 6. Promotion effectiveness with various history lengths for perl. Promotion threshold is 32.

dynamic bias table, frames can be large. The dynamic means is able to make use of the additional branch characterization provided by path history to further increase average frame size. On average, for a history length of 6, a frame consists of about 106 instructions.

Branch correlation effects can be incorporated into the static scheme by using the code replication techniques described by Young and Smith [29]. They demonstrate that compile-time branch assumptions can be made more accurate by considering the paths leading up to a branch. Blocks can be specialized for the particular paths in which their behavior is more regular. Such a scheme, however, is not able to adapt to phased behavior as easily as the dynamic scheme.

One phenomenon that frame construction may be capturing is simple loop unrolling. For all data presented thus far, the loop unrolling option was enabled when the benchmarks were compiled using the Compaq Alpha compiler. Frame construction was able to boost atomic region size beyond the loop unrolling performed by a production C compiler.

As the trends demonstrate, history is an important ingredient for enlarging frames. History helps the promotion mechanism refine its classification of branches. Fig. 6 isolates the effectiveness of Branch Promotion (with $n = 32$) on the benchmark perl. In this graph, each dynamic branch is classified either as a normal branch or an assertion that did not fire or an assertion that fired. In this graph, each bar represents a different path history length. As the amount of history information is increased, the number of branches that appear as assertions increases. Furthermore, the fault rate of these assertions is extremely low. With no history, approximately 65 percent of all dynamic branches are classified as assertions. With a path history of length 6, 85 percent are classified as assertions. Fault rate is below 1 percent of all cases. The net effect is that branch promotion using path history removes 85 percent of the branches (conditional, indirect, returns) from the dynamic instruction stream. While we only present the data for the benchmark perl, all other benchmarks exhibit similar behavior.

The next experiment measures the coverage of the dynamic instruction stream using the described frame construction technique. In essence, this experiment measures the fraction of the instruction stream delivered by a perfect frame cache that is capable of caching every constructed frame. Whenever a new frame is created, the perfect cache is checked. If the frame exists, the corresponding instructions are tallied as covered. If the frame does not exist, those instructions are not covered. With the caching effects factored out, the effects of the frame construction algorithm can be more closely examined. For example, if the frame constructor is creating very small frames (say, if branches are rarely promotable) that rarely exceed the five basic block/32 instruction minimum size threshold, a small fraction of the instruction stream will be covered.

Fig. 7 shows the fraction of the dynamic instruction stream covered by frames created using this technique. The coverage attainable by the ideal static mechanism is also provided. Again, the interference-free dynamic mechanism is able to capture a larger fraction of the instruction stream than the ideal static mechanism. With the dynamic mechanism, at history length 10, almost 90 percent of the dynamic instruction stream on average is covered by frames. With the simple static mechanism, average frame coverage is around 54 percent.

In Table 1, we present data that demonstrate that frames almost always completely execute. The table lists the percentage of executions where a frame completely executed once it was initiated, i.e., no assertions within that frame fired. When an assertion fires, the corresponding frame is flushed. Essentially, any progress made in executing instruction within the frame is discarded. For this reason, assertions can be costly and, therefore, high frame completion rates are desirable. The data is presented for all three promotion thresholds, with the path history length at 6. In general, lowering the threshold increased the assertion rate, but generated frames that were larger and covered more of the instruction stream. Increasing the threshold lowered assertion rate, but decreased frame size and coverage.

4.2 Finite Hardware

We now demonstrate that a straightforward finite-storage implementation of the construction mechanism can also achieve very good results. Fig. 8 shows the average frame length and Fig. 9 shows the percent coverage of the dynamic instruction stream using a 64KB direct-mapped

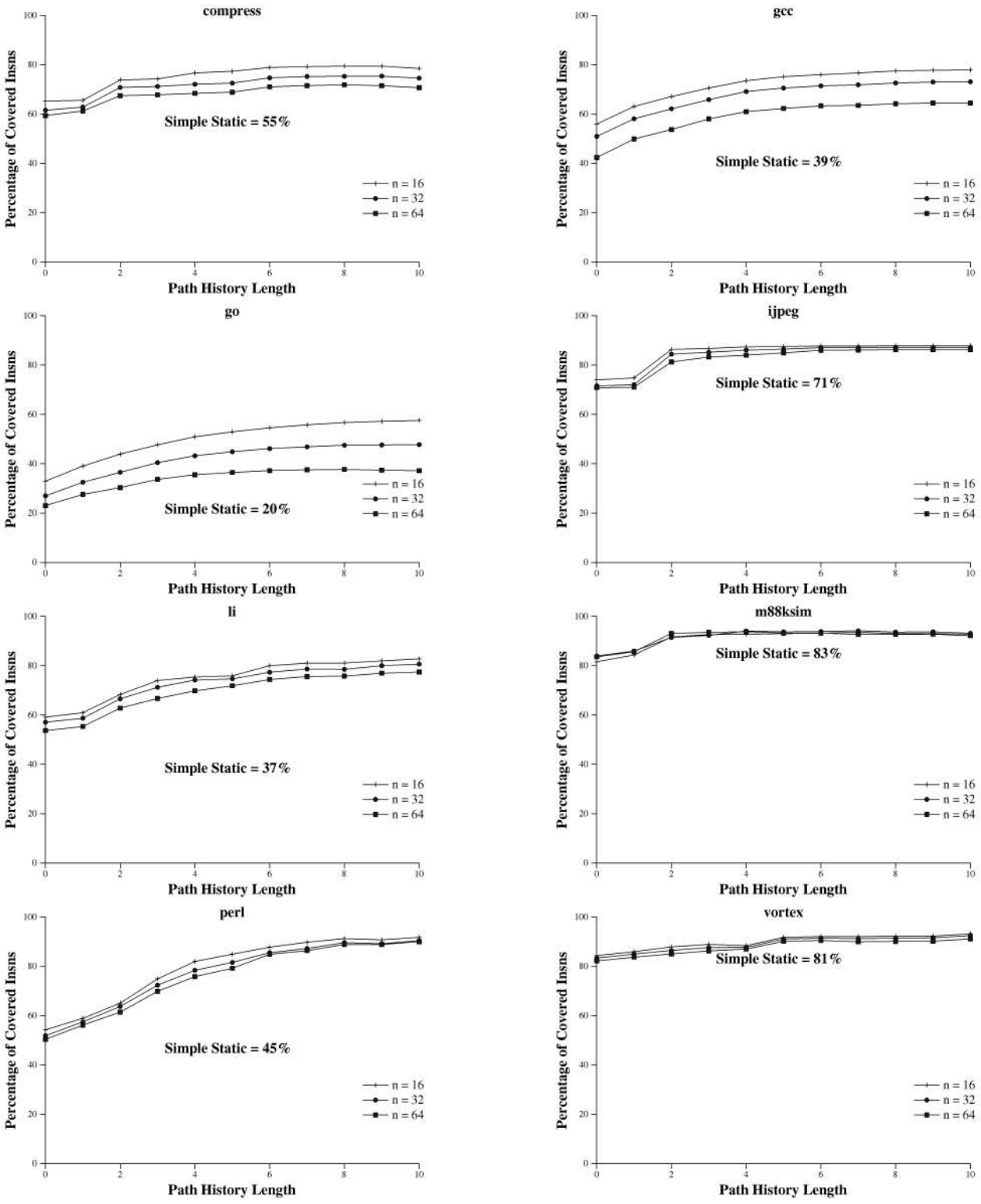


Fig. 7. The percentage of dynamic instructions that occur within a frame identified by this technique.

branch bias table for conditional branches. Each entry in this table is 8 bits, allowing 64K branch instances to concurrently reside in the table.

The degradation from the ideal is substantial in some cases, minor in others. An interesting note: As path history is increased for gcc and go, coverage begins to decline. This is due to the shear number of control paths followed by these benchmarks. Increasing history results in a steep increase in the number of paths needed to be maintained within the bias table. For this reason, we pick a history length of 6 as our base for further evaluation. This path history length strikes a good balance between the positive and negative effects of longer path history.

TABLE 1
The Average Frame Completion Rate at Path History Length = 6

Benchmark	n = 16	n = 32	n = 64
compress	94.95	98.35	99.44
gcc	93.82	96.88	98.43
go	94.12	97.17	98.43
ijpeg	98.56	99.19	99.54
li	93.00	96.27	97.79
m88ksim	94.55	98.65	99.70
perl	97.50	98.55	99.59
vortex	97.46	98.22	98.31
Average	95.50	97.91	98.90

We also use a 10KB direct-mapped bias table for promoting returns and indirect branches. The main difference between the conditional branch bias table and the indirect branch table is that each entry contains a 32-bit target along with an 8-bit bias counter (the table stores up to 2K branch instances). The promotion signal is given only if the same target is used a threshold number of times. Assertions for indirect branches require that the promoted target be encoded along with assertion.

For both bias table sizes, path history is hashed with the address of the current branch in order to form an index into the corresponding tables. We use a path history hashing technique similar to that described by Stark et al. [28]. The technique maintains path history by rotating the old history prior to XORing in a new target. This way, the history pattern encapsulates the ordering of targets within the history, while allowing for a larger number of bits of the target address to be expressed in the history. Fig. 10 demonstrates how this mechanism works conceptually. In this figure, n targets are hashed together to form an m-bit path history. Note that this diagram is a conceptual diagram; the actual implementation of this hashing scheme can be pipelined over several cycles. A pipelined version is provided by Stark et al.

Finally, the frame completion rates using this 64KB bias table (accessed using a path history of six branch targets) are shown in Table 2. The real completion rates are about the same as those attainable by ideal hardware. To summarize the data: With the 64KB+10KB finite bias table, at a history length of 6 and a promotion threshold of 32, we attain an average frame size of 96 instructions, with a coverage of 82 percent and a completion rate of 98 percent. Recall that the frame construction mechanism is only capturing frames that span at least five basic blocks or are at least 32 instructions long. Frames are truncated at the 256th instruction.

Three things need to be noted here. First, the frame constructor's bias table mechanism does not exist in the front end of the processor, therefore, single-cycle access of the bias tables is not essential. The bias tables exist in the frame constructor, where latency is likely not a major factor of performance. The bias table, however, does require supporting the average branch execution bandwidth of the processor, i.e., if the execution engine completes three branches each cycle on average, the promotion hardware needs to support three lookups per cycle. Second, since the promotion information is maintained on the completed branch stream, checkpointing of the associated structures is not required. Third, many techniques developed to reduce the interference within dynamic branch predictors can be applied here to increase the effectiveness of the bias table mechanism toward that of the interference-free case. We have only explored a simple bias table scheme to demonstrate that large frames can be formed effectively using dynamic information.

5 Sequencing Model

We only want to initiate frames at the right time. The frame execution percentages of Tables 1 and 2 indicate that, once a frame is correctly initiated, it has a very high chance of fully executing. However, there are also penalties associated with incorrectly initiating a frame in the first place (similar to the penalties of a regular branch misprediction). To help avoid these penalties, we use a sequencing technique that predicts when a frame should be initiated versus when a conventional fetch should be performed.

As mentioned in Section 2, the frame sequencing happens alongside a conventional branch predictor that sequences through the original control flow of the program (or sequences among traces if a trace cache is used). Whenever the conditions for sequencing to a frame are present, the frame sequencer overrides the prediction generated by the conventional branch predictor.

The sequencer datapath is shown in Fig. 11. A selection mechanism selects between the conventional mechanism and the frame predictor. The selector mechanism can be a history-based mechanism similar to the selector used for a hybrid branch predictor [17] or can be a confidence-based mechanism [12].

The frame predictor works similarly to the trace predictor described by Jacobson et al. [13]. Each entry in the table contains a frame starting address. Entries are accessed using a hashed path history containing the current fetch target. Whenever a frame is added to the frame cache, the frame predictor is updated by adding the frame's address at the entry corresponding to its path history (i.e., the path history used to determine whether or not to promote the first branch in the frame). For example, say frame ABCDE is just created and optimized by the rePLay pipeline. This frame also has an associated path history: If the stream of retiring target addresses was XYZABCDE, a hash of the addresses XYZ forms the path history of the frame ABCDE. The frame predictor is updated at the entry corresponding to the hash of XYZ. This way, whenever the current fetch target is Z and the path history is ...XY, then, in theory, the fetch sequencer outputs the address for frame ABCDE.

For our initial evaluation, we present the effectiveness of a hardware implementation of the frame predictor assuming the selector mechanism operates ideally. We measure frame predictor accuracy by comparing the predicted frame with the next region of instructions encountered in the dynamic instruction stream. If the next region is a frame, the frame addresses are compared. If they match, the frame predictor is tallied a correct prediction. Otherwise, an incorrect prediction is assessed. If the next region of the

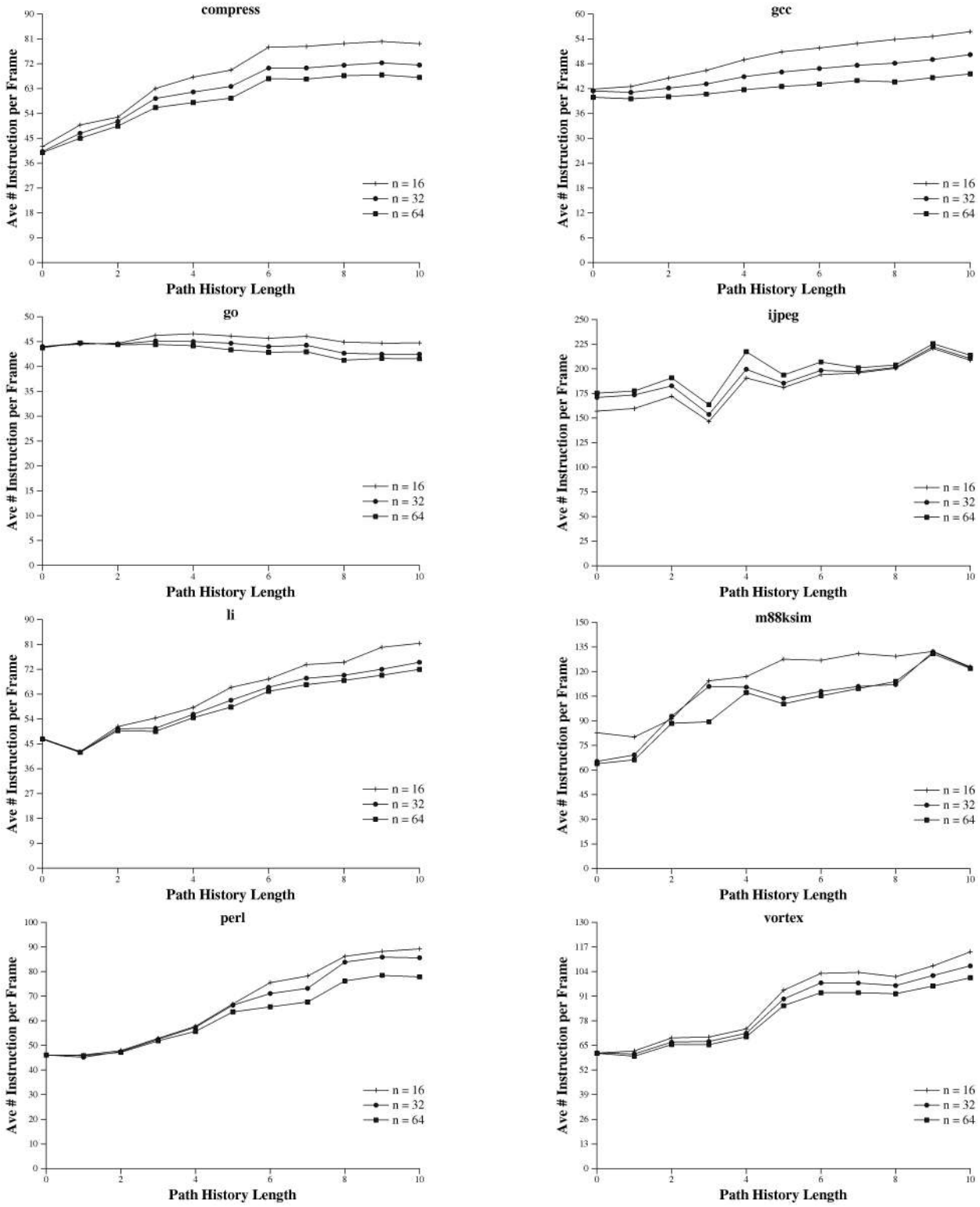


Fig. 8. The average frame size using a 64KB bias table.

instruction stream is not a frame, the frame prediction is dropped.

The results in Table 3 show the effectiveness of a frame predictor of 16K direct-mapped entries using four, six, or eight previous path targets (one of which is the current fetch address) hashed together into a 14-bit index. For all runs, branch bias tables of 64KB + 10KB (as described in Section 4.2) are used for frame construction with promotion threshold of 32.

One thing must be noted here. The low rates of frame prediction reflect the effectiveness of the frame construction algorithm. With the frame constructor, a large fraction of the regularly behaving branches have been collected into frames. The frame predictor's job (and the

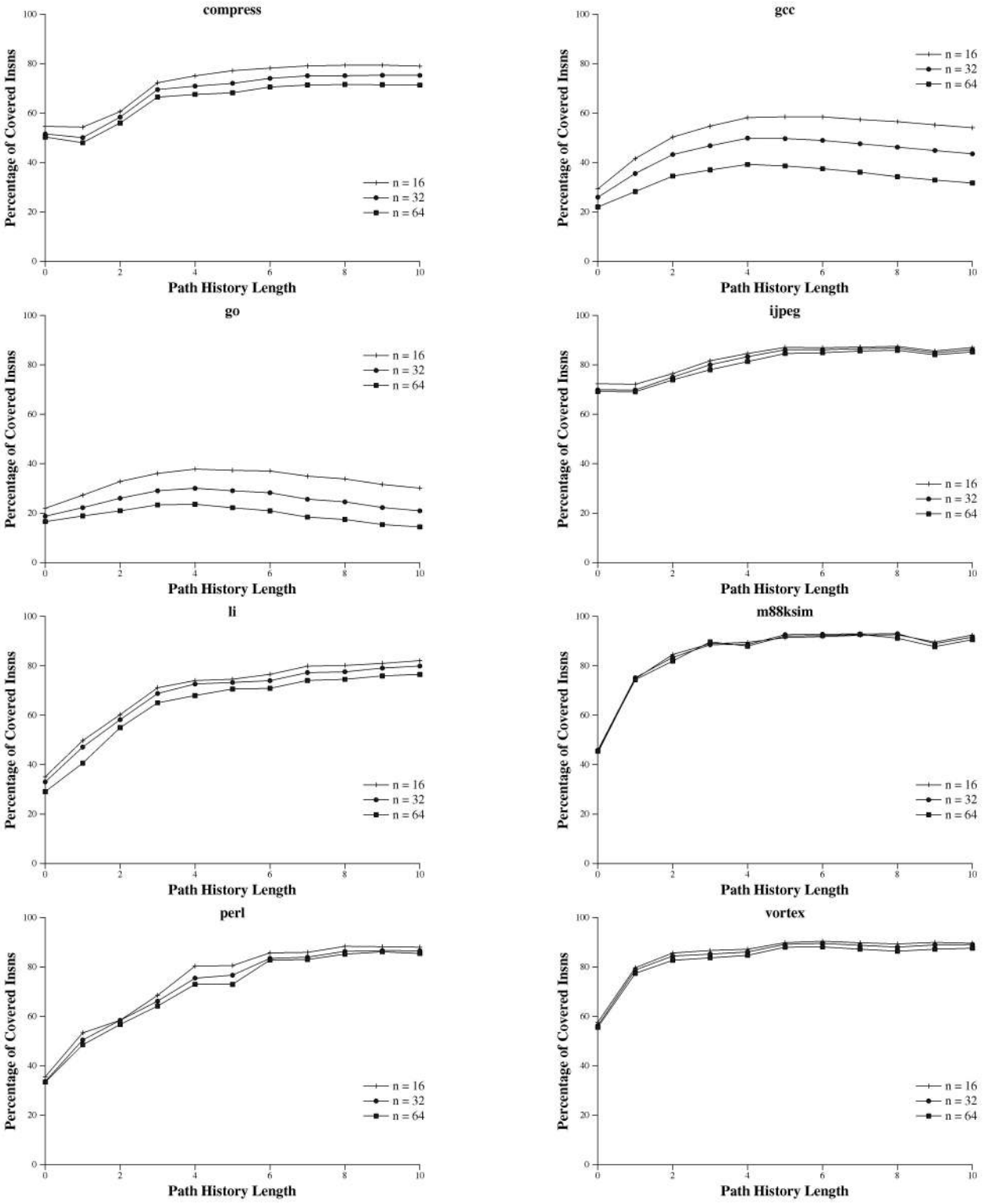


Fig. 9. The percentage of dynamic instructions that are covered by frames identified using a 64KB bias table.

branch predictor's as well) is to now predict the most difficult branches within the program. This difficulty is reflected in the rather low prediction rates shown in Table 3. However, because a significant fraction of branches have been removed from the dynamic instruction stream and converted into assertions, the actual number of predictions required by the frame predictor and branch predictor is significantly reduced to approximately 1/4 of the original dynamic branch count.

6 The Frame Cache

We have now demonstrated that frames span many instructions and, if perfectly cached, can cover a large

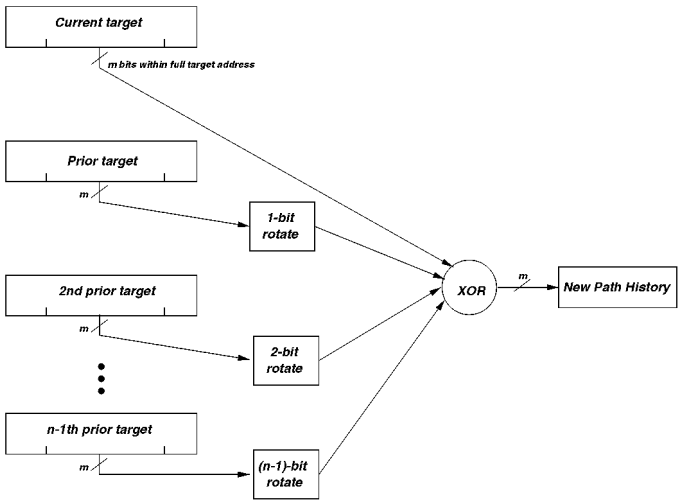


Fig. 10. A conceptual diagram of the path history generation scheme.

fraction of the dynamic instruction stream. In this section, we establish the caching effectiveness of frames with finite sized caches.

A frame cache is essentially a trace cache with the added ability of frames to span multiple cache lines. For instance, if the frame cache had a line size of 16 instructions, a frame containing 80 instructions spans five cache lines. These five cache lines are read from the cache one line at a time, in five consecutive cycles. To enable this pipelining, all lines associated with a particular frame are tagged with the same address—the starting address of the frame. The individual lines of a frame are stored in subsequent sets of the cache. For example, if a frame's address indicates that the initial line of the frame maps to index 30, subsequent lines are found in set 31 onward. The last line of the frame is marked with a termination bit to indicate that the output of the sequencer is used to initiate the next fetch. Frame replacement must occur atomically, i.e., if any line of the frame is to be evicted, the entire frame must be evicted. This happens by using the tag of a line to be evicted to find the first line of the frame. Once the initial line is found, the entire frame can be evicted. A by-product of this scheme is that writes to the frame cache can take many cycles. A major benefit of this scheme is that the frame cache can effectively store both long frames and short traces without wasting cache storage space.

TABLE 2
The Average Frame Completion Rate at Path History Length = 6, Using a Bias Table of 64KB

Benchmark	n = 16	n = 32	n = 64
compress	94.88	98.35	99.43
gcc	94.99	97.57	98.85
go	95.82	98.06	98.98
ijpeg	98.58	99.15	99.45
li	92.37	95.60	97.10
m88ksim	94.12	98.67	99.70
perl	97.55	98.62	99.61
vortex	96.93	98.00	98.27
Average	95.65	98.00	98.92

To measure the effectiveness of caching frames, we use the two metrics used in Section 4: frame length and frame coverage. Frame length is simply a measure of the average number of instructions contained in frames fetched from the frame cache. Frame coverage is the percentage of the dynamic instruction stream covered by frames fetched from the frame cache. Here, a miss in the frame cache results in no frame fetch.

Fig. 12 displays the average frame length for various cache sizes. Here, the frame cache is measured in the number of frames it can hold. Since frames can be of different sizes, this is not a direct measurement of the cache size, but a general indicator of caching effectiveness. In other words, the cache we measured is able to store an entire frame, regardless of its size, in a single entry. These entries are organized in a 4-way set associative manner. Our objective here is to show that frames do exhibit locality and can indeed be cached effectively. The effective size of a frame depends on the optimizations that are performed upon it. Optimizations are likely to reduce the size of a frame from its original length. As we are not currently performing optimizations upon the frames we create, measuring cache space in bytes would portray the benefits of rePLay too conservatively.

Fig. 13 shows the frame coverage using various sized frame caches. For all these measurements, a 64KB bias table (with 10KB bias table for indirect branches) is used for frame construction, accessed using a path history of length 6. The promotion threshold is set to 32. These graphs demonstrate that even with fixed hardware, we get a substantial coverage of the instruction stream with large frames.

Finally, in order for the rePLay Framework to be effective, frames must have a high completion rate. Table 4 shows the frame completion rates, i.e., the probability of completing a frame once it has been started, given various sized frame caches. The rates are high, considering that an average frame contains the equivalent of six conditional branches. The completion rates using fixed size caches are marginally smaller than with perfect caches (see Table 1).

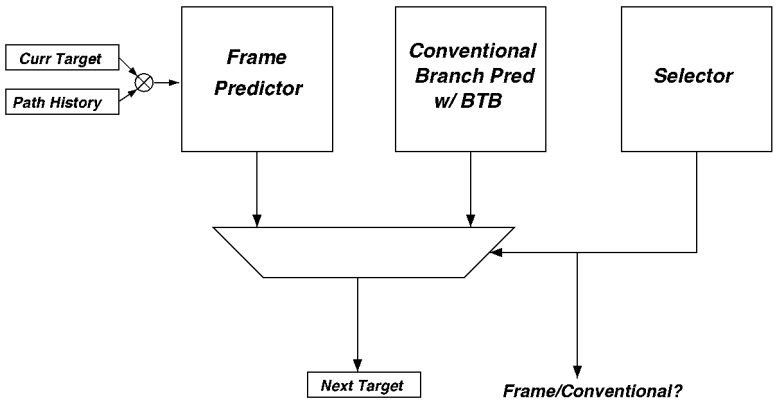


Fig. 11. The Frame Sequencer for the rePLay Framework.

TABLE 3
Accuracy of a 16K Entry Frame Predictor

Benchmark	hist length = 4	hist length = 6	hist length = 8
compress	85.46	86.61	86.17
gcc	84.64	84.28	85.58
go	84.80	87.28	87.69
ijpeg	85.57	90.25	86.41
li	74.42	79.42	76.98
m88ksim	83.38	86.52	86.82
perl	66.50	73.91	74.29
vortex	61.94	61.77	63.99
Average	78.34	81.26	80.99

For a configuration consisting of a frame cache capable of caching 256 frames, a 64KB conditional branch bias table, 10KB indirect branch bias table, 16K entry frame predictor, all using a path history length of 6, we achieve the following results: average frame size of 88 instructions, with these frames covering an average of 68 percent of the dynamic instruction stream, an average frame completion rate of 97.81 percent, and a frame predictor accuracy of 81.26 percent.

7 Sample Frame Optimizations

As apparent from the preceding sections, dynamic frame construction is a powerful tool for partitioning the instruction stream into more predictable pieces. In this section, we consider two typical frames generated by executions of SPEC95 benchmarks on rePLay and discuss potential optimizations for these frames. As the design of the rePLay optimization engine is still open, we select frames that offer fairly obvious opportunities related to interprocedural optimization and dynamic loop unrolling. The optimization engine will analyze frames more carefully and systematically and will uncover opportunities less obvious to the untrained eye. By varying the aggressiveness of the optimizations between the two sample frames, this section provides qualitative insight on the potential value of frames in improving control-related performance.

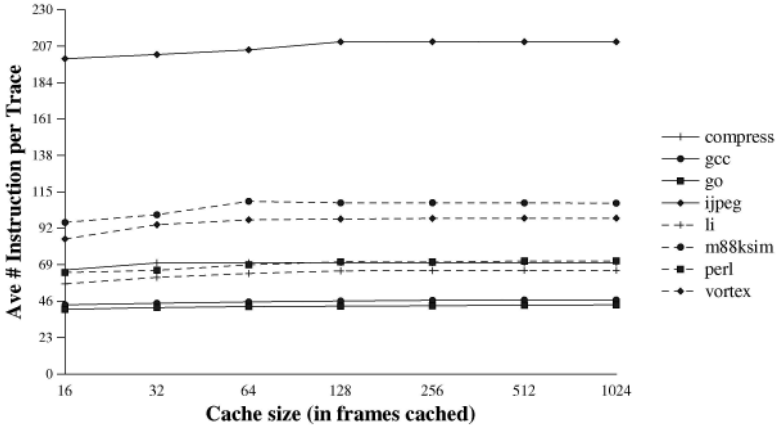


Fig. 12. The average size of a cached frame with varying cache sizes.

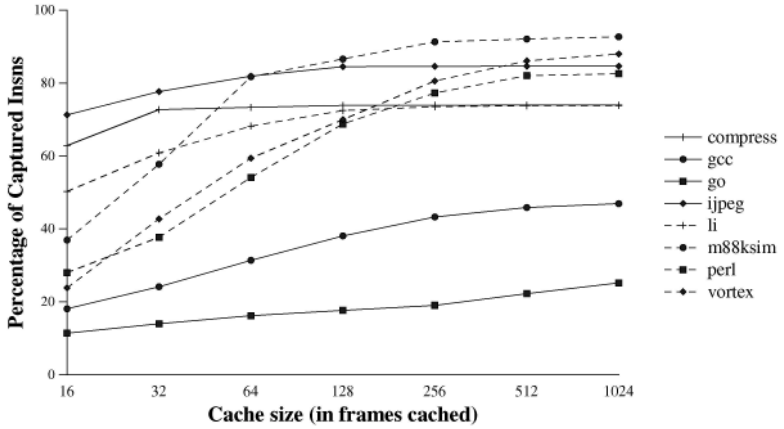


Fig. 13. The coverage of the instruction stream using a fixed size frame cache.

7.1 Memory Allocation Example

Fig. 14 details the first frame, a segment of memory allocation code from gcc, in unoptimized and optimized forms. The format is pseudoassembly code based on Alpha instructions. This frame was the one most frequently initiated during an execution of gcc on the rePLay Framework (256 entry frame cache, 64KB + 10KB bias table, 16K entry frame predictor). During the execution, rePLay initiated the frame 17,942 times and completed it 17,900 times, a completion rate of 99.77 percent. The frame represents a little more than 0.4 percent of all dynamic instructions in the execution.

The left column of the figure lists a sequence of 41 instructions corresponding to the tail end of a call to `alloca`. The frame begins in `malloc` once an appropriate hash bucket has been chosen for an allocation. Such buckets must be refilled periodically, but are typically nonempty. The function unlinks a chunk of memory from the bucket and fills in a private header, then returns to its caller, `xmalloc`. `xmalloc` checks the return value, which is always acceptable, as the **return** (instruction 21) never returns NULL. `xmalloc` next returns to `alloca`, which fills in a private header and returns an adjusted pointer, ending the frame.

A simple scan of the frame reveals that only three registers are live into the frame: gp, the global data pointer; sp, the stack pointer; and s3, the index of the memory allocation hash bucket to be used. Many other registers[1] are overwritten without use. The return value (v0) and stack pointer (sp) are live out of the frame, as are s0, s1, s2, and s3. Considering only the instructions in the frame, registers ra, t0, t1, t5, t6, t7, and t11 must also be treated as live out of the frame, although they are not preserved across C function boundaries. In total, the frame reads three registers and writes 13.

The right column in the figure illustrates a few straightforward optimizations: Superfluous register restore instructions are eliminated, branches and returns are changed to assertions, and stack pointer arithmetic is condensed into a single operation. Optimizations related to register naming and scheduling were not performed, although some are obvious: Renaming register ra to s0 in instructions `06-10` eliminates false dependencies with later instructions and rewriting instruction `13` to add to t7 after `04` reduces the dependency height of the frame, although another scratch register (or a recalculation) must be used to avoid changing t7's value out of the frame. Finally, instruction `23` can be removed, as a value of 0 generates an exception in instruction `07`.

TABLE 4
The Avarage Frame Completion Rate at Path History Length = 6, Using a Bias Table of 64KB, Using Various Sized Frame Caches

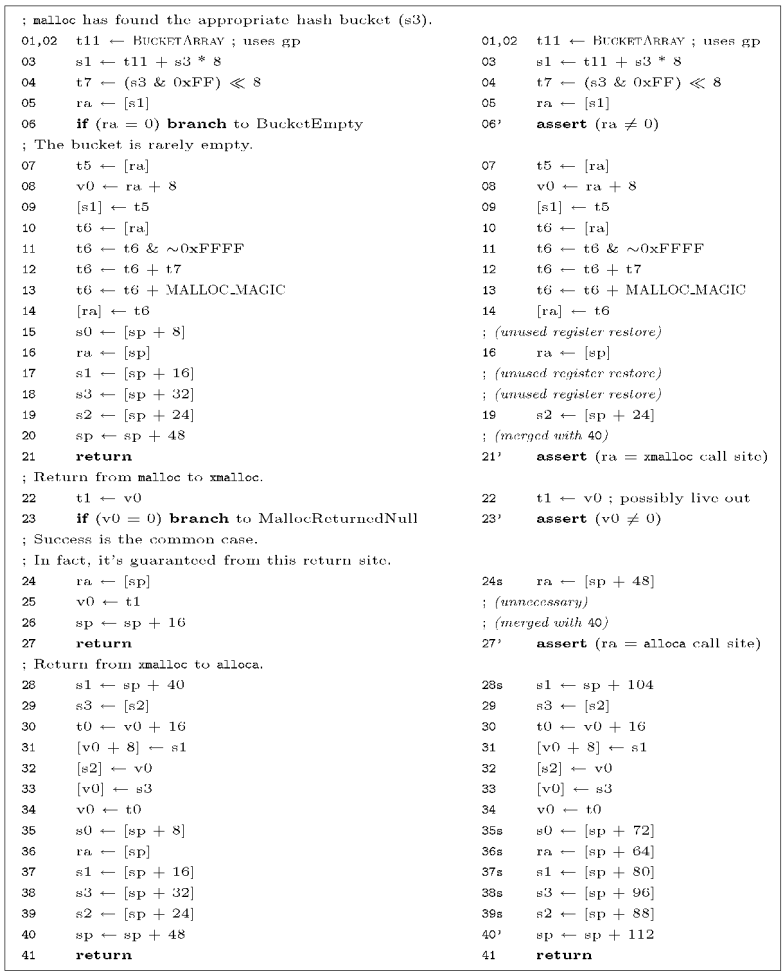
Benchmark	Cache Size (in Frames) 128	256	512
compress	98.34	98.34	98.34
gcc	96.93	97.27	97.41
go	97.00	97.20	97.57
ijpeg	99.17	99.17	99.17
li	95.54	95.57	95.58
m88ksim	98.57	98.65	98.66
perl	98.34	98.52	98.59
vortex	97.46	97.78	97.92
Average	97.67	97.81	97.91

7.2 Data Copying Example

Fig. 15 shows unoptimized and aggressively optimized versions of a second sample frame, a piece of memory copy code from compress. Of frames exhibiting character akin to loop unrolling (i.e., with a repeated component), this frame was the one most frequently initiated during an execution of compress on the rePLay Framework described in Section 7.1. During one execution of compress, rePLay initiated and completed the frame 25,671 times, a success rate of 100 percent. The frame represents more than 2.9 percent of all dynamic instructions in the execution.

The frame corresponds to part of a basic block in the `output` function in which decompressed data are copied to an output buffer. The left column of the figure lists the unoptimized instructions, a total of 136 instructions including an 8-instruction preloop component and an iteration of 16 instructions executed eight times. The number of bytes

1. Registers s0, s1, s2, t0, t1, t5, t6, t7, t11, and v0 are all overwritten.



```
; malloc has found the appropriate hash bucket (s3).
01,02  t11 ← BucketArray ; uses gp
03     s1 ← t11 + s3 * 8
04     t7 ← (s3 & 0xFF) ≪ 8
05     ra ← [s1]
06     if (ra = 0) branch to BucketEmpty
; The bucket is rarely empty.
07     t5 ← [ra]
08     v0 ← ra + 8
09     [s1] ← t5
10     t6 ← [ra]
11     t6 ← t6 & ∼0xFFFF
12     t6 ← t6 + t7
13     t6 ← t6 + MALLOC_MAGIC
14     [ra] ← t6
15     s0 ← [sp + 8]
16     ra ← [sp]
17     s1 ← [sp + 16]
18     s3 ← [sp + 32]
19     s2 ← [sp + 24]
20     sp ← sp + 48
21     return
; Return from malloc to xmalloc.
22     t1 ← v0
23     if (v0 = 0) branch to MallocReturnedNull
; Success is the common case.
; In fact, it's guaranteed from this return site.
24     ra ← [sp]
25     v0 ← t1
26     sp ← sp + 16
27     return
; Return from xmalloc to alloca.
28     s1 ← sp + 40
29     s3 ← [s2]
30     t0 ← v0 + 16
31     [v0 + 8] ← s1
32     [s2] ← v0
33     [v0] ← s3
34     v0 ← t0
35     s0 ← [sp + 8]
36     ra ← [sp]
37     s1 ← [sp + 16]
38     s3 ← [sp + 32]
39     s2 ← [sp + 24]
40     sp ← sp + 48
41     return
```

```
01,02  t11 ← BucketArray ; uses gp
03     s1 ← t11 + s3 * 8
04     t7 ← (s3 & 0xFF) ≪ 8
05     ra ← [s1]
06'    assert (ra ≠ 0)
07     t5 ← [ra]
08     v0 ← ra + 8
09     [s1] ← t5
10     t6 ← [ra]
11     t6 ← t6 & ∼0xFFFF
12     t6 ← t6 + t7
13     t6 ← t6 + MALLOC_MAGIC
14     [ra] ← t6
; (unused register restore)
16     ra ← [sp]
; (unused register restore)
; (unused register restore)
19     s2 ← [sp + 24]
; (merged with 40)
21'    assert (ra = xmalloc call site)
22     t1 ← v0 ; possibly live out
23'    assert (v0 ≠ 0)
24s    ra ← [sp + 48]
; (unnecessary)
; (merged with 40)
27'    assert (ra = alloca call site)
28s    s1 ← sp + 104
29     s3 ← [s2]
30     t0 ← v0 + 16
31     [v0 + 8] ← s1
32     [s2] ← v0
33     [v0] ← s3
34     v0 ← t0
35s    s0 ← [sp + 72]
36s    ra ← [sp + 64]
37s    s1 ← [sp + 80]
38s    s3 ← [sp + 96]
39s    s2 ← [sp + 88]
40'    sp ← sp + 112
41     return
```

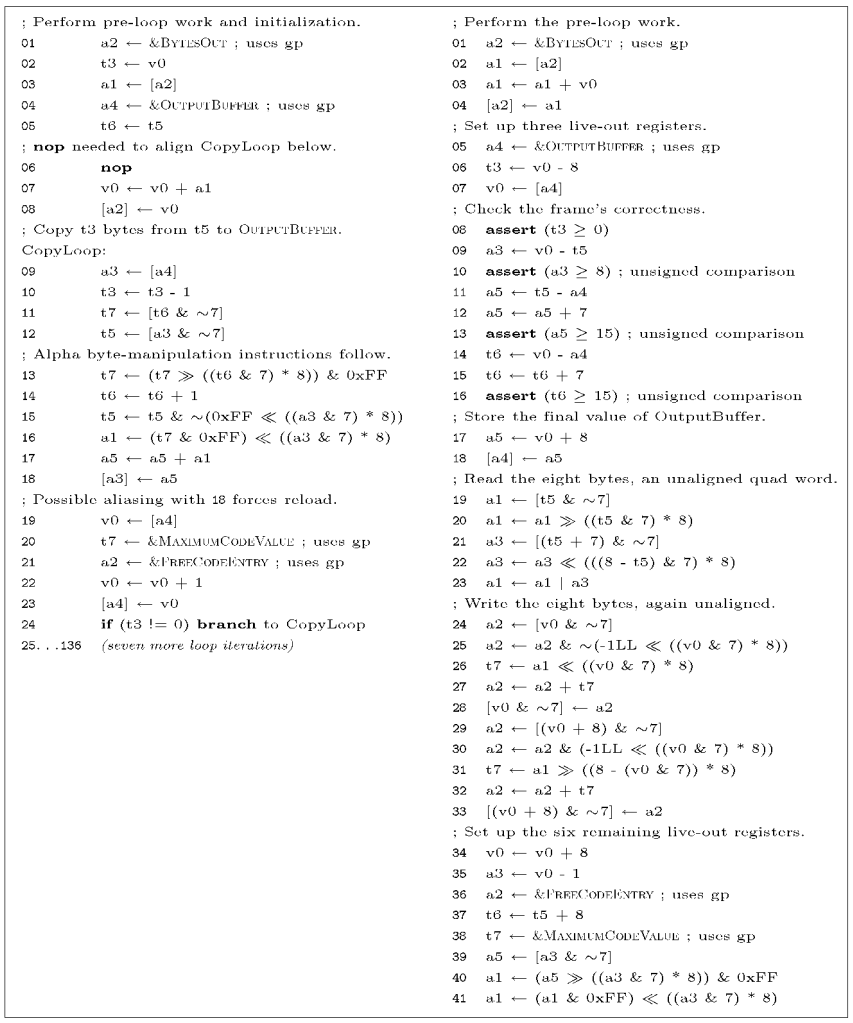
Fig. 14. A sample frame based on memory allocation code from the SPEC95 `gcc` benchmark. The left column is the unoptimized frame. In the right column, some instructions have been eliminated, modified or replaced (primed), or adjusted to reflect the reassociation of stack pointer manipulations (marked with "s").

copied by the loop depends on the current code length, but is always nine or more, thus the frame never faults.

An analysis of this frame is both more complex and more rewarding. The live input registers are again three: gp, the global data pointer; v0, the number of bytes to copy; and t5, the buffer from which bytes are to be copied. The frame overwrites a1-a5, t3, t6, and t7 without reading them and changes v0 in the loop iteration. Register inputs to the loop include a4, a pointer to the storage location for the pointer to the destination buffer; t3, the number of uncopied bytes; and t6, a pointer to the current source byte. Potentially live output registers include all those that the frame overwrites or changes (a1-a5, t3, t6, t7, and v0). Overall, the frame reads three registers and writes nine.

Pointer analysis is one of the more difficult aspects of optimization and it remains a stumbling block for frame optimization. However, we can augment optimized frames with assertions to support likely but unprovable pointer aliasing relationships. For example, in addition to the control assertion that the frame requires at least eight bytes to copy (instruction `08`), we assert the following: The destination does not overlap forward to the source (instructions `09-10`), the source bytes do not overlap with the storage for the destination buffer pointer (instructions `11-13`), and neither do the destination bytes (instructions `14-16`).

Leveraging these assertions, the optimization engine can rewrite the Alpha byte manipulation instructions as unaligned quad-word manipulations, reducing eight iterations to a single load-store combination (instructions `19-33`). The effect of this optimization is to reduce the number of instructions required for the frame from 136 to 41, a factor of more than three. Although the magnitude of this benefit is enhanced by the fortuitous length of the sample frame (we selected it based solely on frequency and its loop-unrolling nature), the optimization is not limited to frames with exactly eight iterations. More or fewer iterations can be grouped into multiple or smaller (e.g., 32-bit word) load-store combinations, generally with a significant savings in dynamic instruction count. The optimized form shown in the figure also makes no attempt to optimize register allocation or instruction scheduling, but rather breaks the instructions into conceptual blocks to improve readability. For the execution discussed here, optimization of one frame reduces the total dynamic instruction count by 2.1 percent.



```
; Perform pre-loop work and initialization.
01        a2 ← &BytesOut ; uses gp
02        t3 ← v0
03        a1 ← [a2]
04        a4 ← &OutputBuffer ; uses gp
05        t6 ← t5
; nop needed to align CopyLoop below.
06        nop
07        v0 ← v0 + a1
08        [a2] ← v0
; Copy t3 bytes from t5 to OutputBuffer.
CopyLoop:
09        a3 ← [a4]
10        t3 ← t3 - 1
11        t7 ← [t6 & ∼7]
12        t5 ← [a3 & ∼7]
; Alpha byte-manipulation instructions follow.
13        t7 ← (t7 ≫ ((t6 & 7) * 8)) & 0xFF
14        t6 ← t6 + 1
15        t5 ← t5 & ∼(0xFF ≪ ((a3 & 7) * 8))
16        a1 ← (t7 & 0xFF) ≪ ((a3 & 7) * 8)
17        a5 ← a5 + a1
18        [a3] ← a5
; Possible aliasing with 18 forces reload.
19        v0 ← [a4]
20        t7 ← &MaximumCodeValue ; uses gp
21        a2 ← &FreeCodeEntry ; uses gp
22        v0 ← v0 + 1
23        [a4] ← v0
24        if (t3 != 0) branch to CopyLoop
25...136  (seven more loop iterations)
```

```
; Perform the pre-loop work.
01   a2 ← &BytesOut ; uses gp
02   a1 ← [a2]
03   a1 ← a1 + v0
04   [a2] ← a1
; Set up three live-out registers.
05   a4 ← &OutputBuffer ; uses gp
06   t3 ← v0 - 8
07   v0 ← [a4]
; Check the frame's correctness.
08   assert (t3 ≥ 0)
09   a3 ← v0 - t5
10   assert (a3 ≥ 8) ; unsigned comparison
11   a5 ← t5 - a4
12   a5 ← a5 + 7
13   assert (a5 ≥ 15) ; unsigned comparison
14   t6 ← v0 - a4
15   t6 ← t6 + 7
16   assert (t6 ≥ 15) ; unsigned comparison
; Store the final value of OutputBuffer.
17   a5 ← v0 + 8
18   [a4] ← a5
; Read the eight bytes, an unaligned quad word.
19   a1 ← [t5 & ∼7]
20   a1 ← a1 ≫ ((t5 & 7) * 8)
21   a3 ← [(t5 + 7) & ∼7]
22   a3 ← a3 ≪ (((8 - t5) & 7) * 8)
23   a1 ← a1 | a3
; Write the eight bytes, again unaligned.
24   a2 ← [v0 & ∼7]
25   a2 ← a2 & ∼(-1LL ≪ ((v0 & 7) * 8))
26   t7 ← a1 ≪ ((v0 & 7) * 8)
27   a2 ← a2 + t7
28   [v0 & ∼7] ← a2
29   a2 ← [(v0 + 8) & ∼7]
30   a2 ← a2 & (-1LL ≪ ((v0 & 7) * 8))
31   t7 ← a1 ≫ ((8 - (v0 & 7)) * 8)
32   a2 ← a2 + t7
33   [(v0 + 8) & ∼7] ← a2
; Set up the six remaining live-out registers.
34   v0 ← v0 + 8
35   a3 ← v0 - 1
36   a2 ← &FreeCodeEntry ; uses gp
37   t6 ← t5 + 8
38   t7 ← &MaximumCodeValue ; uses gp
39   a5 ← [a3 & ∼7]
40   a1 ← (a5 ≫ ((a3 & 7) * 8)) & 0xFF
41   a1 ← (a1 & 0xFF) ≪ ((a3 & 7) * 8)
```

Fig. 15. A sample frame based on code byte copying from the SPEC95 compress benchmark. The loop body appears eight times in the frame. With an aggressive optimization engine and pointer aliasing assertions, the frame can be reduced from 136 to 41 instructions, as shown on the right.

7.3 Summary

This section has presented a range of possible optimizations on frames, from simple single-pass elimination of redundant and useless instructions to value speculation to eliminate pointer aliasing. We believe that the former class of optimizations, although fairly simple and straightforward, can provide substantial benefits when applied to large regions of code. The dynamic identification of the control path allows instructions intended for alternative paths or for the possibility of multiple paths to be eliminated. Register spill code for unexecuted code, for example, can be transformed to stores or eliminated entirely in some cases. Finally, in the case of unoptimized code, many of the optimizations normally performed by a compiler can be handled by rePLay.

The more aggressive class of optimizations is perhaps even more promising. Use of hardware-based speculation to exploit high-probability (or low-probability) relationships between pointers passed into a frame can result in very effective optimizations, as demonstrated by the memory copy example. A similar technique can be employed to capitalize on predictable data values. As the optimization architecture will contain programmable elements, interaction with compilers and profilers to further improve the focus and impact of these optimizations is also possible.

8 Conclusion

We have described a new hardware framework for enhancing application performance by using dynamic optimizations. The rePLay Framework centers on the concept of a frame, a logically atomic sequence of instructions drawn dynamically from an executing program. The rePLay mechanism allows a programmable optimization engine to make speculative optimizations upon frames, such as ones based on assumptions about control flow behavior or data values. In the unlikely event that these assumptions are incorrect, a hardware-based recovery mechanism rolls back state to the beginning of a frame. This coupling between a hardware-based optimizer and recovery mechanism can potentially reduce the overheads suffered by software-based dynamic optimizers.

These frames are typically much larger than the regions considered by previous work on dynamic optimization. Branch promotion, in particular, plays a key role in the

rePLay frame construction strategy. Once rePLay has constructed a potentially useful frame, the frame undergoes online optimization and is stored in a frame cache. Branches, returns, and indirect calls are transformed into instructions that assert the prediction implied by the linear instruction sequence in the frame. The rePLay sequencer then fetches and initiates the frame when the branch path history indicates a high likelihood of completion.

A rePLay configuration with a 256-entry 4-way set associative frame cache, a 64KB direct-mapped conditional branch bias table, a 10KB direct-mapped indirect branch bias table, 16K entry frame predictor, and a path history length of 6, achieves an average frame size of 88 instructions with 68 percent coverage of the dynamic instruction stream, an average frame completion rate of 97.81 percent, and a frame predictor accuracy of 81.26 percent. These results soundly demonstrate that the frames upon which the optimizations are performed are large and stable.

Using the most frequently initiated frames from rePLay executions as samples, we highlighted possible strategies for the rePLay optimization engine. Many traces contain interprocedural linkage that can easily be stripped away to reduce dynamic instruction count. Loop unrolling and reoptimization based on dynamic iteration counts also seems promising. Finally, the use of assertions about infrequent pointer aliasing can significantly improve the level of frame aliasing conditions that ever occur, the frame faults and normal instruction execution resumes. Coupled with the high coverage of frames achieved through the dynamic construction approaches outlined in earlier sections, the success of these optimizations demonstrates the significance of the rePLay Framework.

REFERENCES

[1] V. Bala, E. Duesterwald, and S. Banerjia, "Transparent Dynamic Optimization: The Design and Omplementation of Dynamo," Technical Report HPL-1999-78, Hewlett-Packard Laboratories, June 1999.

[2] P.-Y. Chang, M. Evers, and Y.N. Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," *Proc. 1996 ACM/IEEE Conf. Parallel Architectures and Compilation Techniques,* 1996.

[3] Y. Chou and J.P. Shen, "Instruction Path Coprocessors," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* 2000.

[4] D.A. Connors and W.W. Hwu, "Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results," *Proc. 32nd Ann. ACM/IEEE Int'l Symp. Microarchitecture,* 1999.

[5] K. Ebcioglu and E.R. Altman, "Daisy: Dynamic Compilation for 100% Architectural Compatibility," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* 1997.

[6] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers,* vol. 30, no. 7, pp. 478-490, July 1981.

[7] D.H. Friendly, S.J. Patel, and Y.N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," *Proc. 31st Ann. ACM/IEEE Int'l Symp. Microarchitecture,* 1998.

[8] M. Gschwind, E.R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "Dynamic and Transparent Binary Translation," *Computer,* vol. 33, no. 3, pp. 54-59, Mar. 2000.

[9] R.E. Hank, S.A. Mahlke, R.A. Bringmann, J.C. Gyllenhaal, and W.W. Hwu, "Superblock Formation Using Static Program Analysis," *Proc. 26th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* pp. 247-255, 1993.

[10] E. Hao, P.-Y. Chang, M. Evers, and Y.N. Patt, "Increasing the Instruction Fetch Rate via Block-Structured Instruction Set Architectures," *Int'l J. Parallel Programming,* vol. 26, no. 4, pp. 449-478, Aug. 1998.

[11] W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *J. Supercomputing,* vol. 7, pp. 9-50, 1993.

[12] E. Jacobsen, E. Rotenberg, and J.E. Smith, "Assigning Confidence to Conditional Branch Predictions," *Proc. 29th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* pp. 142-152, 1996.

[13] Q. Jacobson, E. Rotenberg, and J.E. Smith, "Path-Based Next Trace Prediction," *Proc. 30th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* 1997.

[14] Q. Jacobson and J.E. Smith, "Instruction Pre-Processing in Trace Processors," *Proc. Fifth IEEE Int'l Symp. High Performance Computer Architecture,* 1999.

[15] A. Klaiber, "The Technology behind Crusoe Processors," technical report, Transmeta Corp., Jan. 2000.

[16] 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," *Proc. 25th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* pp. 45-54, 1992.

[17] S. McFarling, "Combining Branch Predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[18] S. Melvin and Y. Patt, "Enhancing Instruction Scheduling with a Block-Structured ISA," *Int'l J. Parallel Programming,* vol. 23, no. 3, pp. 221-243, June 1995.

[19] 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," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* 2000.

[20] R. Nair and M.E. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 13-25, 1997.

[21] S.J. Patel, M. Evers, and Y.N. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing," *Proc. 25th Ann. Int'l Symp. Computer Architecture,* 1998.

[22] S.J. Patel, D.H. Friendly, and Y.N. Patt, "Evaluation of Design Options for the Trace Cache Fetch Mechanism," *IEEE Trans. Computers,* vol. 48, no. 2, pp. 435-446, Feb. 1999.

[23] S.J. Patel, T. Tung, S. Bose, and M.M. Crum, "Increasing the Size of Atomic Instruction Blocks Using Control Flow Assertions," *Proc. 33rd Ann. ACM/IEEE Int'l Symp. Microarchitecture,* 2000.

[24] A. Peleg and U. Weiser, "Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line," US Patent Number 5,381,533, 1994.

[25] E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* 1996.

[26] M.D. Smith, "Overcoming the Challenges of Feedback-Directed Optimization," *Proc. ACM SIGPLAN Workshop Dynamic and Adaptive Compilation and Optimization,* 2000.

[27] A. Sodani and G.S. Sohi, "Dynamic Instruction Reuse," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* 1997.

[28] J. Stark, M. Evers, and Y.N. Patt, "Variable Length Path Branch Prediction," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* p. 170-179, 1998.

[29] C. Young and M.D. Smith, "Improving the Accuracy of Static Branch Prediction Using Branch Correlation," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 232-241, 1994.

Sanjay J. Patel received the PhD degree from the University of Michigan, Ann Arbor, in 1999. He received the BS and MS degrees from the University of Michigan and has done hardware verification, logic design, and performance modeling at Digital Equipment Corporation, Intel, and HAL Computer Systems, and has consulted for Transmeta and Jet Propulsion Laboratory. He is an assistant professor of electrical and computer engineering at the University of Illinois at Urbana-Champaign. He is coauthor of a unique bottom-up introduction to computing titled "Introduction to Computing Systems: From Bits and Gates to C and Beyond." His research interests include processor microarchitcture, computer architecture, and high performance and reliable computer systems. He is a member of the IEEE and the IEEE Computer Society.

Steven S. Lumetta received the AB degree in physics in 1991, the MS degree in computer science in 1994, and the PhD degree in computer science from the University of California at Berkeley in 1998. He is an assistant professor of electrical and computer engineering and a research assistant professor in the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. He has worked on a wide range of problems in scalable parallel computing, including languages (Split-C), tools (Mantis debugger), algorithms, and runtime systems, culminating in his dissertation on multiprotocol, user-level communication on clusters of SMP's. Dr. Lumetta's research interests are in optical networking, high-performance networking and computing, hierarchical systems, and parallel runtime software. He is a member of the IEEE and the IEEE Computer Society.

▷ **For further information on this or any computing topic, please visit our Digital Library at** http://www.computer.org/publications/dlib.