rePLay : A Hardware Framework for Dynamic Program Optimization

Sanjay J. Patel Steven S. Lumetta
Center for Reliable and High-Performance Computing
The University of Illinois at Urbana-Champaign

{sjp, steve}@crhc.uiuc.edu

CRHC Technical Report #CRHC-99-16

December 1999

Abstract

In this paper, we propose a new framework for enhancing application performance through execution-guided optimization. The rePLay Framework uses information gathered at run-time to optimize an application's instruction stream. Some of these optimizations persist temporarily for only a single execution, others persist between runs. The heart of the rePLay Framework is a trace-cache like device called the frame cache, used to store optimized regions of the original executable. These regions, called *frames*, are large, single-entry, single-exit regions spanning many basic blocks in the program's dynamic instruction stream. Optimizations are performed on these frames by a flexible optimizer contained within the processor.

A rePLay configuration with a 256-entry frame cache, using realistically-sized frame constructor and frame sequencer achieves an average frame size of 88 instructions with 68% coverage of the dynamic istream, an average frame completion rate of 97.81%, and a frame predictor accuracy of 81.26%. 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.

1 Introduction

Several underlying trends (transistor counts, wire delay effects, memory latency, implementation complexity) direct those working on processor microarchitecture to examine new and creative ways of increasing the performance of general applications on general-purpose computer systems. As a result, future processor solutions are likely use novel techniques for high-performance.

In this paper, we propose a new framework for enhancing application performance through execution-guided optimization. The rePLay Framework uses information gathered at run-time to optimize an application's instruction stream. Some of these optimizations persist temporarily for only a single execution, others persist between runs. The heart of the rePLay Framework is a trace-cache like device called the frame cache, used to store optimized regions of the original executable. These regions, called *frames*, are large, single-entry, single-exit regions spanning many basic blocks in the program's dynamic instruction stream. Frames

are larger regions than those considered by previous trace cache research. Optimizations are performed on these frames by a flexible optimizer contained within the processor.

The rePLay Framework consists of four elements : a frame constructor to identify and create atomic (i.e., single-entry, single-exit) frame regions, an optimization engine to optimize each newly created frame, a frame cache to store these optimized frames, and a sequencing mechanism to predict when a particular frame will visited again in the dynamic instruction stream. The integration of these rePLay components with a processor's fetch and execution engines is shown in Figure 1

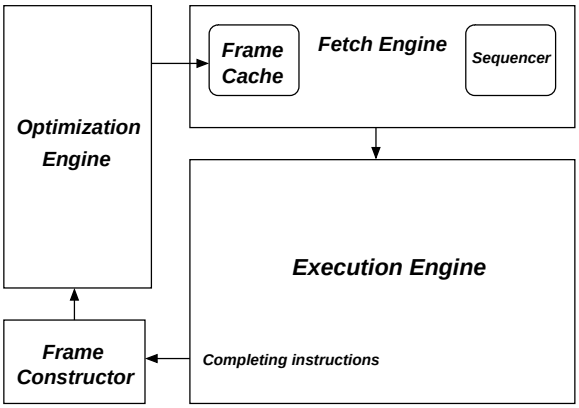

Figure 1: The rePLay Framework integrated into a generic processor microarchitecture.

The main contribution of this paper is a demonstration of the basic phenomenon that drive rePLay. The rePLay Framework exploits run-time stability – repeating patterns of control and data – to boost program performance. We demonstrate that large frames, spanning many control-flow instructions, can be created dynamically with simple hardware mechanisms. It is upon these frames that the rePLay optimizations are performed. We demonstrate that these frames are frequently executed, and can be effectively cached after optimization. We present a sequencing mechanism, based on a path-based predictor, that provides control speculation between frames. Finally, we present a sampling of optimizations which are possible within the rePLay Framework, many of which are best done knowing run-time behavior rather than the limited behavioral information available to a compiler.

2 The rePLay Framework

Run-time information is beginning to play a larger role in boosting 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 information is useful in a way that is different from information available at compile time. A compiler can very effectively capitalize on stable behavior (e.g., a highly biased branch) apparent from profile executions of a program. However, if the program behaves differently from the way the compiler expected, optimizations based on the expected behavior may degrade performance. Furthermore, periodic variations in behavior during execution (e.g., a conditional branch which is highly biased for first half of the program,

and then highly biased in the other direction for the second half) are difficult for a compiler to capitalize upon.

2.1 Approach

The rePLay Framework exposes stable run-time behavior to a hardware-based optimizer that performs lightweight code optimizations on sections of the istream. In a sense, a piece of an optimizing compiler is embedded within the processor hardware. The optimizer operates on *frames* that consist of many basic blocks in the original control flow. These frames are dynamically constructed and maintain a *single-entry, single-exit* semantic despite containing many branch instructions. Like traces in a trace cache, frames may consist of code that is not physically sequential in the static image of a program, but is dynamically converted into straight-line code. Once optimized, these frames are stored in the frame cache.

The rePLay Framework allows for local optimizations in situations in which compiler actions would have been difficult or impossible. Old binaries, transformations across dynamic library calls, applications not amenable to profiling or aggressive compilation are all situations where optimizations at compile-time alone may be ineffective. Furthermore, rePLay allows for types of optimizations that cannot effectively be done at compile-time. Such optimizations include ones based on the run-time stability of data values. We provide several examples in Section 7.

This work builds upon several concepts recently explored in computer architecture research. We leverage techniques developed for the trace cache and apply them in the context of an execution-guided optimization system. Central to frame construction is the concept of Branch Promotion originally developed to increase the size of traces stored in the trace cache. Previous trace cache work also suggests that the latency of trace construction and optimization does not impact performance because they occur off the critical processing path. This indicates that latency in the frame constructor and optimizer within the rePLay Framework may not be a large factor on overall performance.

2.2 Frames

The first concept of rePLay is the concept of the atomic region or *frame*. Similar in nature to the *trace* developed by a trace scheduling [5] compiler or the *block* within the Block-Structured ISA [15, 9], a frame is a dynamically-generated group of instructions in which all instructions in the group execute or none of them do. A frame has a single entry point and a single exit point. This *atomic* property of frames increases the level of dynamic optimization that can be performed on them.

The distinction between a frame and a trace as described by previous trace cache research is subtle, but important. A frame is a more specific entity than the a trace – it is a trace with a single entry point and a single exit point. The concept of an atomic trace was not central to previous trace cache research. It is however central to a frame within rePLay. Either the entire frame executes, or none of it executes.

2.3 Frame construction

The objective of frame construction is to create long frames that span many basic blocks. Such long frames increase the potential of the dynamic optimizer in finding opportunities for optimization that were missed at compile time. In order for frames to span multiple basic blocks while maintaining atomicity (i.e., single

entry, single exit), branches in the original instruction stream must be somehow removed when packaged within a frame. In rePLay, these branches are removed via *branch promotion*.

With branch promotion [18], branches which behave in a highly regular manner are dynamically converted into non-branching ASSERT instructions that verify that the branching conditions still hold. If an ASSERT's branching conditions do not hold, then the ASSERT fires and control is directed to a recovery point. Specifically, branches that have gone to the same target for the n consecutive previous occurrences are targeted for promotion into ASSERTs. As original blocks from the program finish execution and are retired, the corresponding control instructions are checked using a hardware structure called a bias table to determine if they should be promoted.

Within the frame construction logic, promoted control instructions (ASSERTs) cause a pending frame to keep growing. Regular, non-promoted control instructions terminate a pending frame. Figure 2 shows how original control flow is reorganized into a frame. For the frame ABCDE, a firing ASSERT — or *fault* — causes program control to return to the original block A. In such a case, none of the instructions of the frame are committed to architectural state.

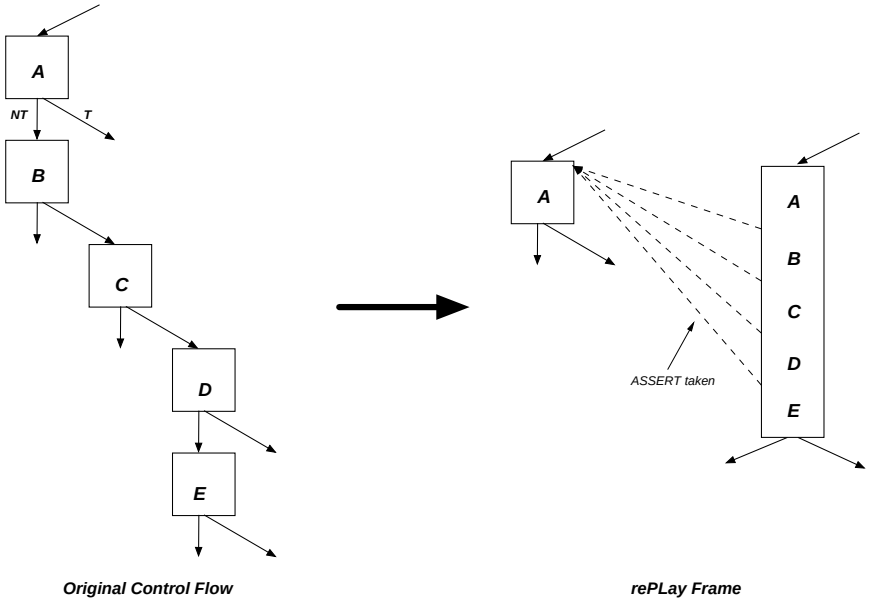

Figure 2: This figure demonstrates how a single-entry, single-exit frame is constructed from a control for spanning several branches.

Like traces, frames may contain taken branches from the original code. In other words, frame construction dynamically remaps non-sequential control flow into straight-line code.

While the frame constructor's primary function is to create long frames, it must balance this objective with two others: high likelihood of frame completion and high degree of frame coverage of the dynamic instruction stream. Frames with a high likelihood of completion incur less penalty due to false starts; a firing ASSERT causes the partial execution of a frame to be discarded and diminishes the benefit of creating the frame in the first place. Higher frame coverage of the instruction stream increases the opportunity of rePLay Framework for delivering dynamically optimized code to the execution engine. In summary, the frame constructor can be evaluated using three metrics: the average length of a frame (in instructions,

or basic blocks), how often the average frame is likely to completely execute, and how much of the total instruction stream is covered by these dynamically constructed frames.

While not examined in this study, the frame construction hardware can potentially access hardware performance counters to direct frame construction to execution hot spots [16].

2.4 Optimization engine

Once a frame is created, it is dynamically optimized by the optimization engine. The optimization engine is a flexible optimization datapath that can be software-programmed to tailor optimizations towards an application, or towards a section of code. The range of optimization includes classical compiler optimizations, extended basic block optimizations [10], and also encompasses software-based dynamic optimizations systems [1]. Furthermore, the coupling of dynamic optimizations, execution rollback mechanisms, and rePLay's assertion instruction architecture allows for new classes of optimizations such as ones that speculate on the value of particular input data. These optimizations assume particular live-in data values are known values upon entry to a frame and pre-propagate those values throughout the frame. For each assumed value, a data ASSERT 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 instruction reuse. 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.

In this paper, we discuss rePLay optimizations briefly in Section 7. Due to the complex nature of the optimization engine, we expect the frame latency through the optimization engine to take many cycles. However, we use the data gathered from an earlier study [6] as an indication that the negative effects of this latency on performance are manageable.

2.5 Frame cache

Once a frame has been processed by the optimization engine, it is stored in the frame cache. The frame cache is similar to the trace cache except that it is capable of delivering very long sequences of instructions spanning multiple traditional cache lines. A frame cache line may also span several issue-widths wide. For example, a particular frame might consist of 80 instructions, span 5 cache lines (16 instructions per cache line), and take 5 cycles to be fetched and issued on a 16-wide fetch/issue processor. This caching mechanism is described in more detail in Section 6.

2.6 Sequencer

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 chains.) The rePLay Framework uses a speculative sequencing mechanism to predict when a frame should be initiated and when it should not. For example, consider Figure 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. In Section 5, we examine this sequencing mechanism in more detail.

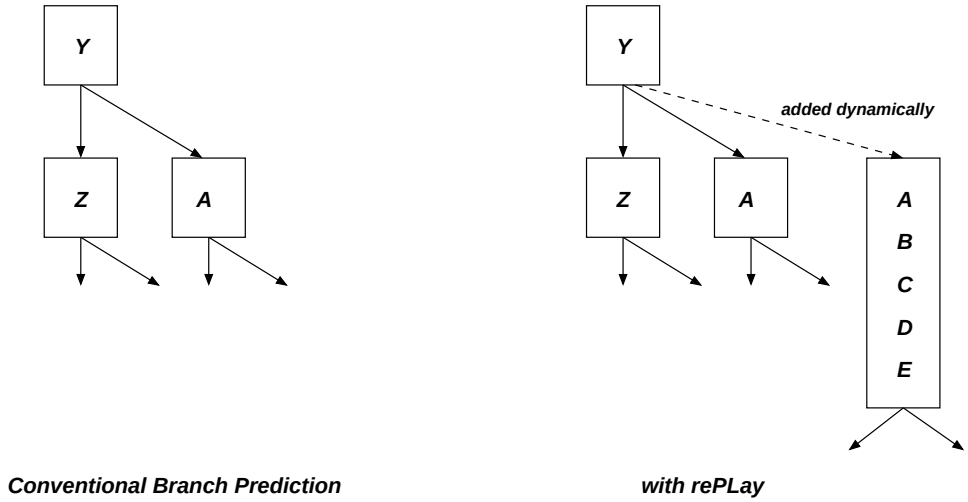

Figure 3: This figure shows two version of control flow originating from the same static branch. Unlike with a conventional branch predictor, the number of targets selected amongst by the rePLay sequencer varies dynamically, even for the same static branch. As a frame is created (ABCDE, in this case), it is added as a potential target from the prior block (Y, in this case). The sequencer decides when ABCDE should be selected versus, say, simply A.

In order 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, for the few cycles, a frame streams out of the frame cache in fetch-width sized packets.

3 Related Work

Much of this work builds upon previous trace cache research [20, 21, 19], in particular that of Branch Promotion [18] and that of dynamic trace optimizations [7, 13], and also that of the Trace Predictor [12]. Similar in nature to this work is the DIF cache [17] which dynamically creates statically scheduled instruction words.

The concept of dynamic compilation and optimization is an emerging area. The desire to have multi-ISA compatibility has spawned work in dynamic compilation systems [3], which then have spawned work in dynamic optimization systems [1, 4, 8]. Almost all of this work has focused on low-overhead software systems. The rePLay Framework can leverage these software techniques by providing a very low overhead mechanism for implementing dynamic optimization that monitors dynamic behavior at a microarchitectural level.

4 Frame Construction

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 semantic allows the run-time optimizer to perform very aggressive optimizations upon the generated frames.

Frames can span many basic blocks of a program. To prevent branches that terminate these blocks from also terminating frames, we use the Branch Promotion technique to dynamically convert highly-biased branches into unconditional branches that generate a hardware fault if they switch direction. A fault causes the instructions of the frame to be discarded (i.e., not committed to architectural state) and the control flow to be redirected to the proper recovery point. See Figure 2 for an example. In this manner, Branch Promotion is used to convert branches internal to a frame into ASSERT instructions. The promotion technique was originally applied to only conditional branches, but here we apply it also to indirect branches and returns.

With Branch Promotion, the frame construction mechanism is simple: a each completing branch accesses an entry in the branch bias table. The bias table is a hardware structure that counts the number of times a branch has had the same outcome consecutively. We improve upon the original promotion technique by accessing the bias table with path history, in addition to the branch address. Each time a branch (plus path) 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 and is converted into an ASSERT. 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 branch prediction research has demonstrated that such a technique is an effective way to identify highly-biased branches [2]. We increase the accuracy of the mechanism by adding path history. Figure 4 is a diagram of the Branch Promotion mechanism.

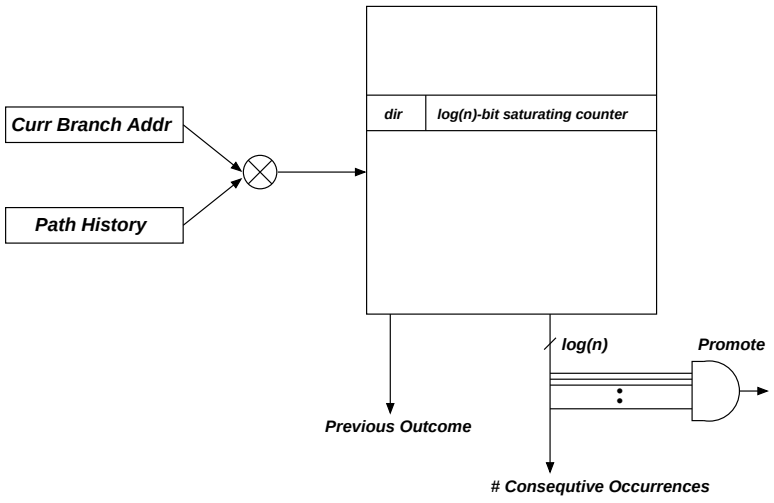

Figure 4: The bias table mechanism for Branch Promotion with promotion threshold n.

Frames are constructed by extending a basic block across promoted branches. For each arriving promoted branch, the pending frame grows by the corresponding block. The pending frame terminates when an unpromoted branch is added. In other words, a frame consists of a sequence of instructions (including unconditional branches and ASSERTs) ending at a single unpromoted branch. Even though the original

instructions may have been non-contiguous, they are converted into a contiguous frame.

Larger frames are beneficial for the rePLay Framework. Larger frames spanning more basic blocks present the rePLay optimizer with a greater opportunity performing effective optimizations. For the measurements presented in the remainder of the paper, the frame constructor only maintains frames containing 5 or more dynamic basic blocks or containing 32 or more instructions. Smaller frames are discarded.

4.1 The ideal

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

Figure 5 demonstrates the potential of this frame construction technique by showing average frame size in instructions on the 8 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 using profile-guided code placement to reduce the impact of taken branches. Also, link-time code placement optimizations using the OM executable editor were performed on these binaries.

Figure 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. The data indicates that a large number of instructions can be incorporated into a frame. On average, for a history length of 6, a frame consists of about 106 instruction.

Also shown on this graph is the region size if an ideal static predictor were used in place of the dynamic bias table. Here, the ideal static mechanism classifies a branch as promoted if it has a 95% bias towards a particular target during the profile run. The mechanism is considered ideal 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. For all schemes, the dynamic mechanism generated longer frames than the ideal static, sometimes significantly. Although, not shown on the graph, history information can also be factored in at compile-time using code replication techniques described by Young et al [23]. Longer histories (which are beneficial) can be difficult to incorporate at compile-time because of the multiplicity of paths that need to be maintained.

As the trends demonstrate, history is an important ingredient for enlarging frames. History helps the promotion mechanism refine its classification of branches. Figure 6 isolates the effectiveness of Branch Promotion (with $n = 32$) on the benchmark perl. In this graph, each bar represents the total number of dynamic branches. Each dynamic branch is either classified as a normal (unpromoted branch), or promoted branch which obeyed its promoted direction, or a promoted branch which faulted (i.e., the ASSERT fired). In this graph, a bar is generated for each path history length. As the amount of history information is increased, the number of branches which are classified as promoted increases. Furthermore, the fault rate of promoted branches is extremely low. With no history, approximately 65% of all dynamic branches are classified as promoted. With a path history of length 6, 85% are classified as promoted. Fault rate is below 1% of all promoted branches. The net effect is that branch promotion using path history is removing 85% of the branches (conditional, indirect, returns) from the dynamic instruction stream.

The next experiment measures the coverage of the dynamic instruction stream using the described frame

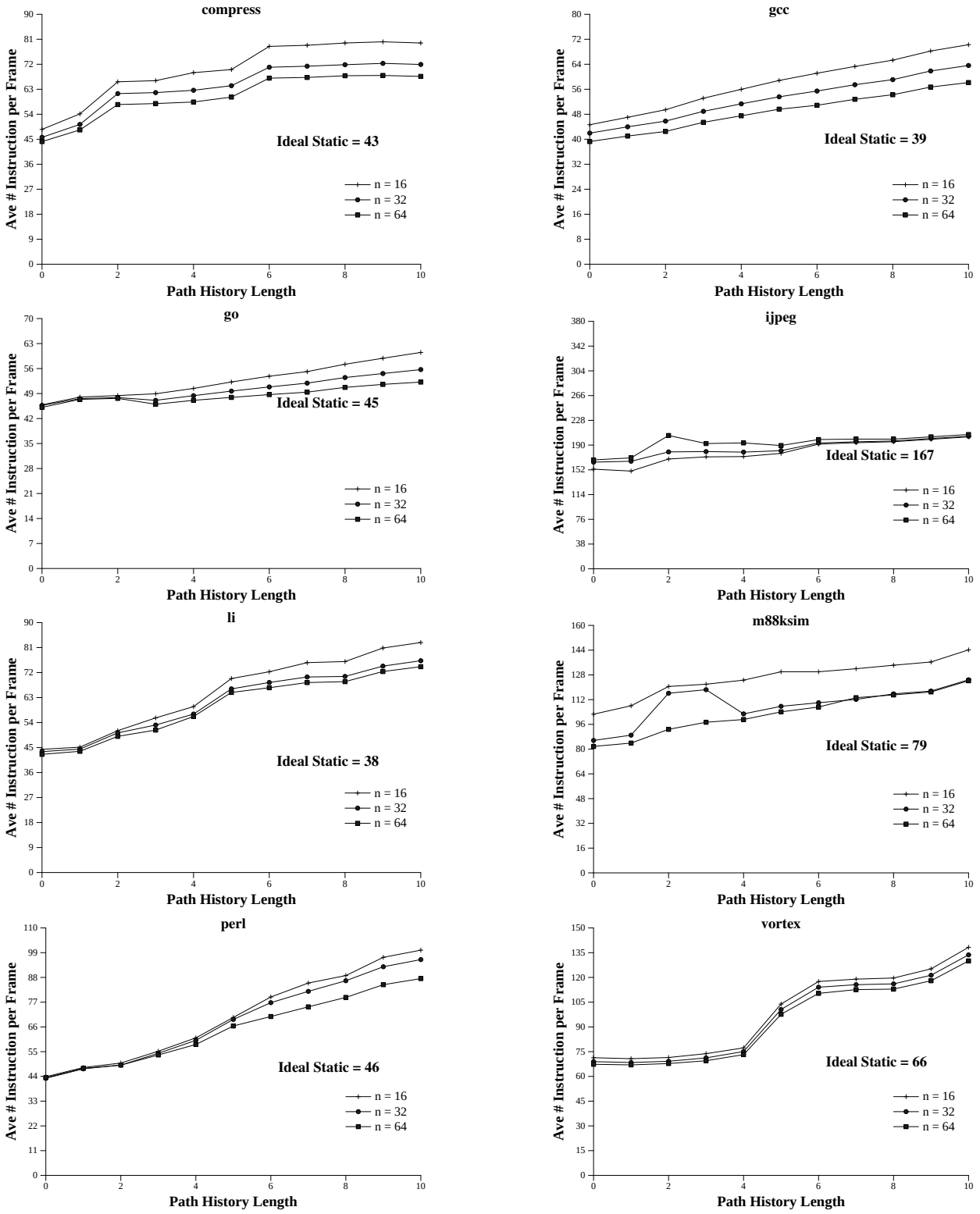

Figure 5: The average size of a frame gathered using an interference-free bias table indexed with a given path history length. Path history length indicates the number of branch targets which appear in the history. For each benchmark, an ideal static number is provided to show static frame length obtained by best-case compiler analysis.

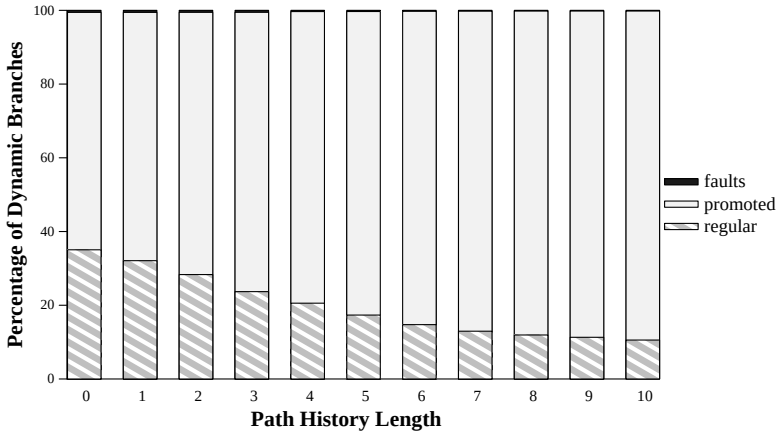

Figure 6: Promotion effectiveness as history length is increased.

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, then the corresponding instructions are tallied as covered. If the frame doesn't exist, then 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 5 basic block/32 instruction minimum size threshold, then a small fraction of the istream would be covered.

Figure 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 istream than the ideal static mechanism. With the dynamic mechanism, at history length 10, almost 90% of the dynamic istream on average is covered by frames. With the ideal static mechanism, average frame coverage is around 54%.

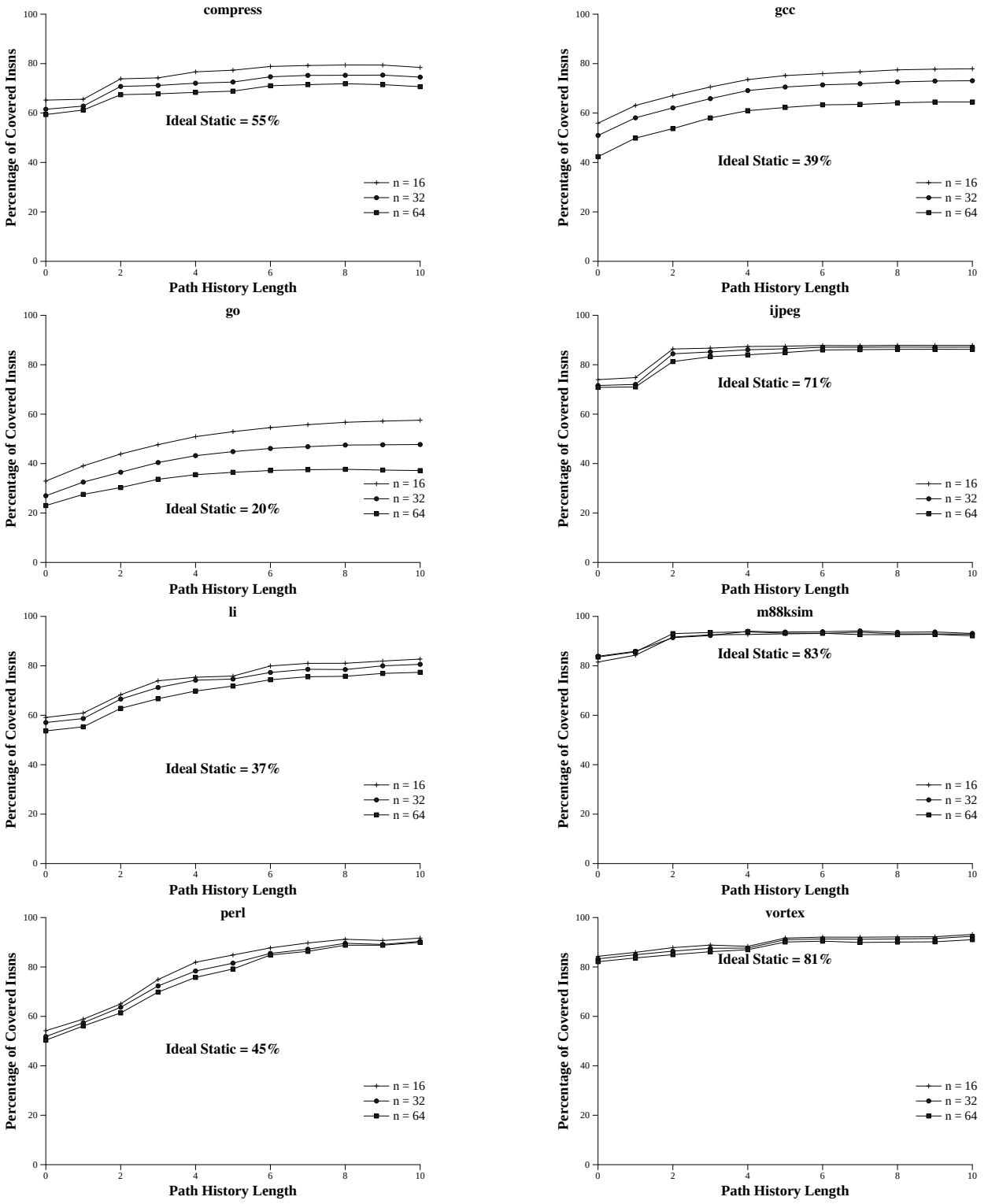

Figure 7: The percentage of dynamic instructions which occur within a frame identified by this technique.

Table 1 shows the percentage of occurrences that a frame executes completely, i.e., no ASSERT within that frame triggered. When an ASSERT instruction triggers, the corresponding frame is flushed and the fetch redirected to the recovery point. Essentially, any progress made in executing instruction within the frame is lost. For this reason, assertions can be costly and therefore high frame completion rates are desirable. Table 1 lists the ratio of frame completion versus frame initiation. In other words, the table lists how often a frame is completely executed once it has been started. The data is presented for all three promotion thresholds, with the path history length at 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

Table 1: The average frame completion rate at path history length = 6.

4.2 Finite hardware

We now demonstrate that a straightforward finite-storage implementation of the promotion mechanism can also achieve very good results. Figure 8 shows the average frame length and Figure 9 shows the percent coverage of the dynamic istream using a 64KB branch bias 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.

We also use a 12KB 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 target along with the bias counter. The promotion signal is given only if the same target is used a threshold number of times. ASSERTs for indirect branches require that the promoted target be encoded along with ASSERT. The assertion is made that the target taken by the control flow is the one assumed by the ASSERT.

For both bias tables, the path history, along with the address of the current branch is hashed 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 [22]. 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. Figure 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

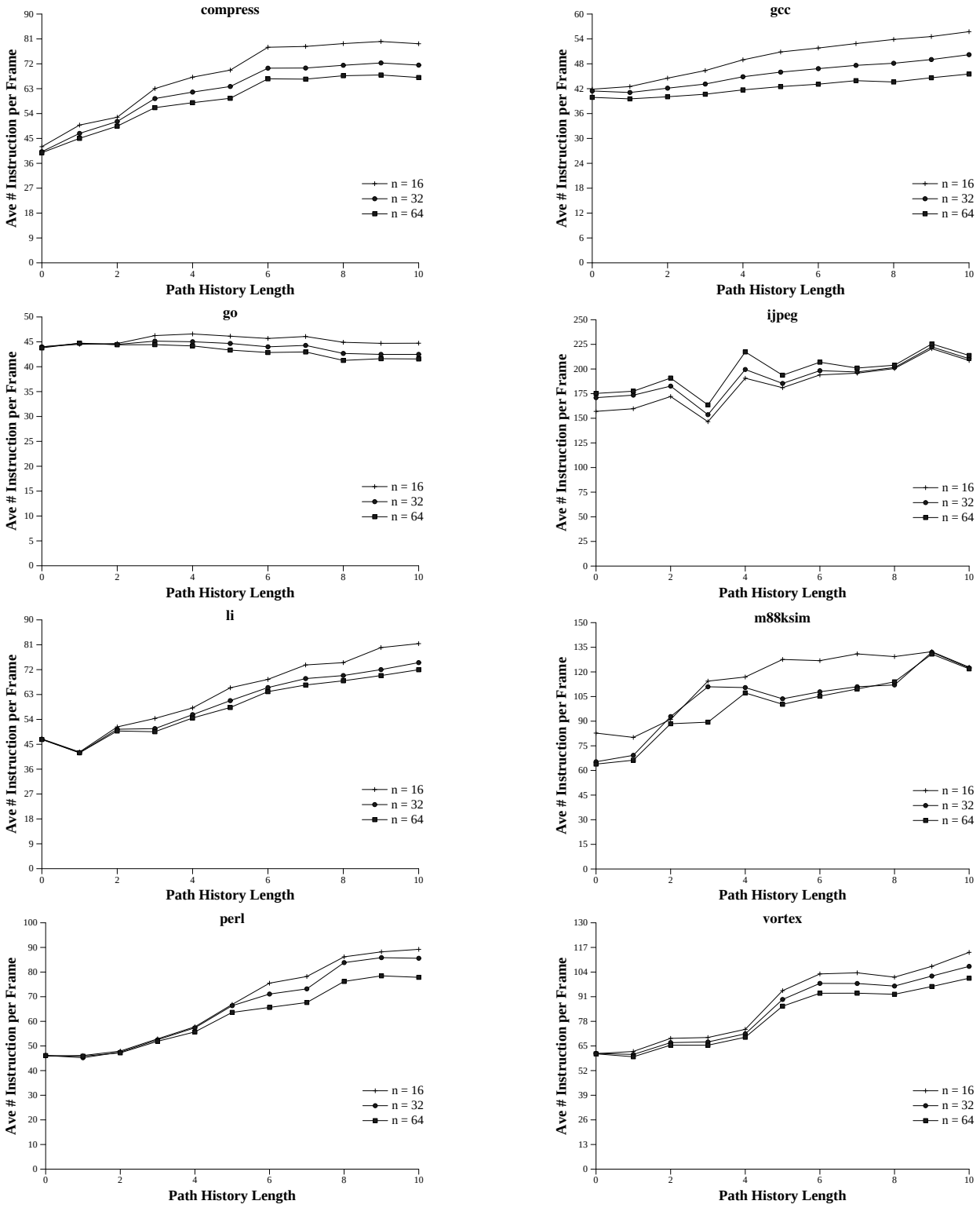

Figure 8: The average frame size using a 64KB bias table.

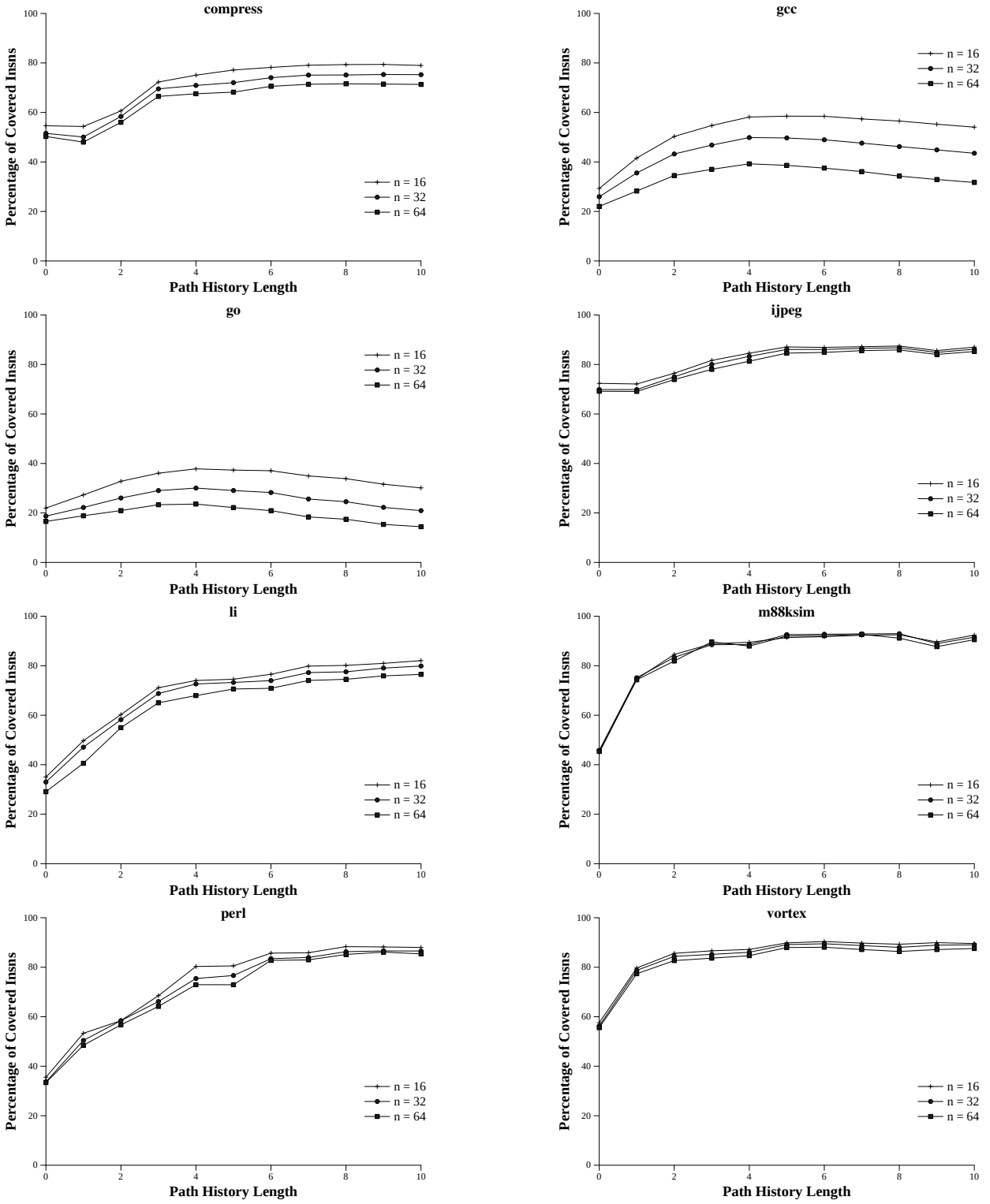

Figure 9: The percentage of dynamic instructions which are covered by frames identified using a 64KB bias table.

of this hashing scheme can be pipelined over several cycles. A pipelined version is provided by Stark et al.

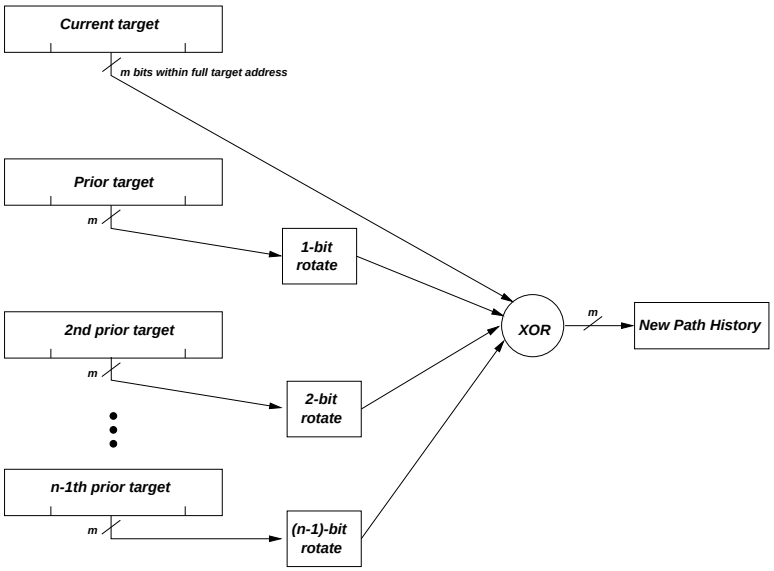

Figure 10: A conceptual diagram of the path history generation scheme.

Finally, the frame completion rates using this 64KB bias table (accessed using a path history of 6 branch targets) are shown in Table 2. The real completion rates are about the same as those attainable by idealized hardware. To summarize some of the data: with the 64KB+12KB 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%, and a completion rate of 98%. Recall that the frame construction mechanism is only saving frames that span at least 5 basic blocks or are at least 32 instructions long.

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

Table 2: The average frame completion rate at path history length = 6, using a bias table of 64KB.

Three things need to be noted here. First, this Branch Promotion 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 to performance. However, the promotion mechanism does require supporting the average branch execution bandwidth of the processor, i.e., if the execution engine completes three branches each cycle on average, then 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 towards 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 entered versus when a conventional fetch should be performed.

As mentioned in Section 2, the frame sequencing happens alongside a conventional branch predictor which sequences through the original control flow of the program (or sequences amongst 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 Figure 11. A selection mechanism selects between the conventional mechanism and the frame predictor. The selector mechanism can be history-based mechanism similar to the selector used for a hybrid branch predictor [14], or can be a confidence-based mechanism [11].

The frame predictor works similarly to the trace predictor described by Jacobson et al [12]. 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 WXYZABCDE, then a hash of the addresses WXYZ forms the path history of the frame ABCDE. The frame predictor is updated at the entry corresponding to the hash of WXYZ. This way, whenever the current fetch target is Z and the path history is ...WXY, then, in theory, the fetch sequencer outputs the address for frame ABCDE.

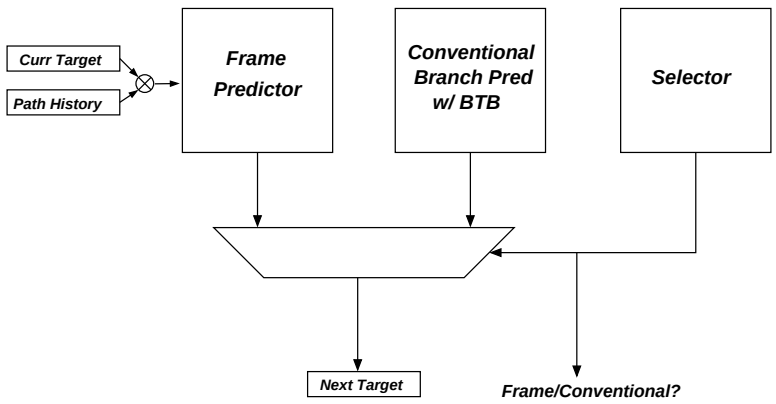

Figure 11: The Frame Sequencer for the rePLay Framework.

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 then the frame predictor is tallied a correct prediction. Otherwise, an incorrect prediction is assessed. If the next region of the istream is not a frame, then the frame prediction is dropped.

The results in Table 3 show the effectiveness of a frame predictor of 16K entries using 4, 6, or 8 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+12KB (as described in Section 4.2) are used for frame construction with promotion threshold of 32.

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

Table 3: Accuracy of a 16K entry frame predictor.

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 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 ASSERTs, the actual number of predictions required by the frame predictor and branch predictor is significantly reduced to 25% 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 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, then a frame containing 80 instructions would span five cache lines. These five cache lines would be read from the cache one line at a time, in five consecutive cycles. To enable this, all lines associated with a particular frame would be tagged with the same address – the starting address of the frame. Also, each entry contains a continuation field to hold the frame cache index of the next line of the frame. If the current line is the last line of the frame, then terminal bit is set to indicate that the output of the sequencer is used to initiate the next fetch. Finally, in order

to make replacement easier, a frame is stored within the same set of a set-associative cache. Doing this, however, requires a high degree of frame cache set-associativity in order to avoid a substantial increase in cache misses due to set conflicts. Using this scheme, both long frames and short traces can be cached in the same structure (i.e., the frame cache and the trace cache can be the same structure.) Better schemes for caching frames are under development.

To measure the effectiveness of the frame cache, 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.

Figure 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 the effectiveness of caching. Our objective here is to show that frames do exhibit locality and can indeed be cached effectively. Figure 13 shows the frame coverage using various sized frame caches. For all these measurements, a 64KB bias table (with 12KB 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 istream with large frames.

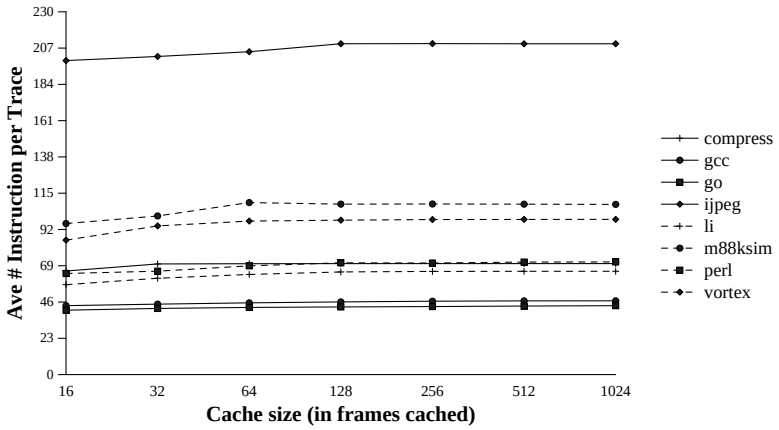

Figure 12: The average size of a cached frame with varying cache size.

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 six conditional branches. The completion rates using fixed size caches are marginally smaller than with perfect caches (See Table 1).

For a configuration consisting of a frame cache capable of caching 256 frames, a 64KB conditional branch bias table, 12KB 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% of the dynamic istream, an average frame completion rate of 97.81%, and a frame predictor accuracy of 81.26%.

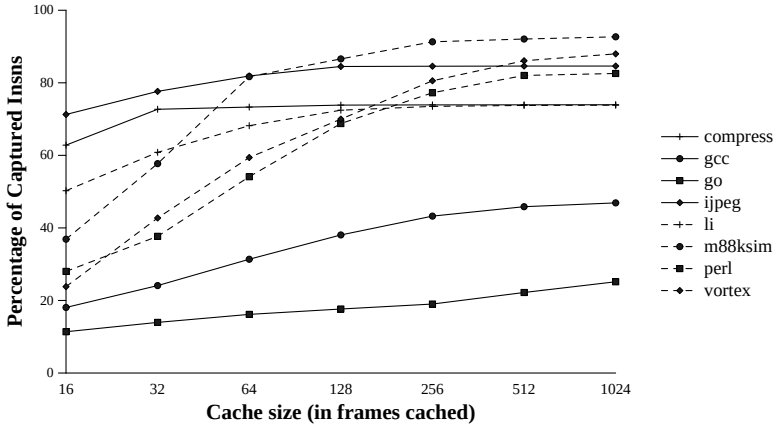

Figure 13: The coverage of the istream using a fixed size frame cache.

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

Table 4: The average frame completion rate at path history length = 6, using a bias table of 64KB, using various sized frame caches.

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 rePLay optimization engine is still under construction, 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.

7.1 Memory Allocation Example

Figure 14 details the first frame, a segment of memory allocation code from gcc, in unoptimized and optimized forms. This frame was the one most frequently initiated during an execution of gcc on the rePLay Framework (256 entry frame cache, 64KB+12KB bias table, 16K entry frame predictor). During one execution of gcc, rePLay initiated the frame 17,942 times and completed it 17,900 times, a completion rate of 99.77%. The

```
; 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
```

Figure 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").

frame represents a little more than 0.4% 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 non-empty. 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 thirteen.

The right column in the figure illustrates a few straightforward optimizations: superfluous register restore instructions and motion 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.

7.2 Data Copying Example

Figure 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%. The frame represents more than 2.9% 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 pre-loop component and an iteration of 16 instructions executed eight times. The number of bytes 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

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

```
; 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)
```

Figure 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.

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%.

7.3 Summary

Using the most frequently initiated frames from rePLay executions as samples, this section highlighted possible strategies for the rePLay optimization engine. Many traces contain interprocedural linkage code 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 optimization by eliminating potential data dependencies; if the rare aliasing conditions ever occur, the frame faults and normal instruction execution resumes. Coupled with the high coverage of frames achieved through rePLay's frame construction approaches, the success of these optimizations indicates the potential value of the rePLay Framework.

8 Conclusion

We have described a new framework for enhancing the performance of an application using execution-guided optimizations. The rePLay Framework centers on the concept of a frame, a single-entry, single-exit, linearized sequence of instruction drawn dynamically from an executing program. These frames are typically much larger than the traces considered by previous work on trace caches, but are constructed in large part through technology developed for trace caches. 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 frame cache, a 64KB+12KB bias table, 16K entry frame predictor, and a path history length of 6, achieves an average frame size of 88 instructions with 68% coverage of the dynamic istream, an average frame completion rate of 97.81%, and a frame predictor accuracy of 81.26%. 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, this section 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 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.

We have deliberately left the execution architecture slightly vague in this paper, as we do not wish to constrain the idea of constructing and dynamically optimizing frames to a particular implementation. However, given the very low register input and output counts on the sample traces (three inputs and nine or thirteen outputs for 41 and 136 instructions, respectively), an approach similar to that taken by processor architectures which speculate at the thread level might provide interesting opportunities to speculatively execute frames and to reduce the synchronization implied by the relatively low frame prediction rates. Regardless of the specific implementation, however, we believe that the concept of frames, along with the mechanisms and strategies that we outlined in this paper, will play an important role in processor architectures of the future.

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. Technical Report HPL-1999-78, Hewlett-Packard Laboratories, June 1999.

[2] Po-Yung Chang, Marius Evers, and Yale N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. In *Proceedings of the 1996 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1996.

[3] Anton Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, B. Yadavalli, and J. Yates. Fx!32: a profile-directed binary translator. *IEEE Micro*, 18(2), March 1998.

[4] C. Consel and F. Noël. A general approach for run-time specialization and its application to c. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 145 – 156, 1996.

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

[6] Daniel H. Friendly, Sanjay J. Patel, and Yale N. Patt. Alternative fetch and issue techniques from the trace cache fetch mechanism. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[7] Daniel H. Friendly, Sanjay J. Patel, and Yale N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31th Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[8] Brian Grant, Marcus Mock, Matthai Phillipose, Craig Chambers, and Susan J. Eggers. Dyc: An expressive annotation-directed dynamic compiler for c. Technical Report UW-CSE-97-03-03, University of Washington, May 1999.

[9] Eric Hao, Po-Yung Chang, Marius Evers, and Yale N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. *International Journal of Parallel Programming*, 26(4):449–478, August 1998.

[10] W. W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(9-50), 1993.

[11] Erik Jacobsen, Eric Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 142–152, 1996.

[12] Quinn Jacobson, Eric Rotenberg, and James E. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

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

[14] Scott McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

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

[16] M. Merten, A. Trick, C. George, J. Gyllenhaal, and Wen mei Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 136–149, 1999.

[17] Ravi Nair and Martin 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] Sanjay J. Patel, Marius Evers, and Yale 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] Sanjay J. Patel, Daniel H. Friendly, and Yale N. Patt. Evaluation of design options for the trace cache fetch mechanism. *IEEE Transactions on Computers*, 48(2):435–446, February 1999.

[20] Alexander Peleg and Uri Weiser. Dynamic flow instruction cache memory organized around trace segments independant of virtual address line. U.S. Patent Number 5,381,533, 1994.

[21] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.

[22] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch predcition. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 170 – 179, 1998.

[23] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, 1994.