Increasing the Size of Atomic Instruction Blocks using Control Flow Assertions

Sanjay J. Patel Tony Tung Satarupa Bose Matthew M. Crum

Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{sjp, tonytung, sbose, mcrum}@crhc.uiuc.edu

Abstract

For a variety of reasons, branch-less regions of instructions are desirable for high-performance execution. In this paper, we propose a means for increasing the dynamic length of branch-less regions of instructions for the purposes of dynamic program optimization. We call these atomic regions frames and we construct them by replacing original branch instructions with assertions. Assertion instructions check if the original branching conditions still hold. If they hold, no action is taken. If they do not, then the entire region is undone. In this manner, an assertion has no explicit control flow. We demonstrate that using branch correlation to decide when a branch should be converted into an assertion results in atomic regions that average over 100 instructions in length, with a probability of completion of 97%, and that constitute over 80% of the dynamic instruction stream. We demonstrate both static and dynamic means for constructing frames. When frames are built dynamically using finite sized hardware, they average 80 instructions in length and have good caching properties.

1 Introduction

An atomic region of code has the following properties: execution of the region begins at a single instruction, ends at a single instruction, and the region contains a single path of execution. The region is considered atomic because if one instruction in the region is committed to architectural state, then all instructions are committed. A basic block, for example, is an atomic region.

Atomic regions consisting of many instructions are desirable for a variety of reasons. They allow a compiler maximum flexibility for optimizations. Code scheduling in atomic regions, for example, need not account for side entrances, side exits, or divergent paths of execution. Atomic regions provide hardware with a sequential stream of instructions with no control flow. Instruction fetch mechanisms can stream out an atomic region with a single PC and a single branch prediction. Optimistic state recovery mechanisms need only save state at boundaries of atomic regions.

The nature of programs, however, is such that atomic regions typically consist of very few instructions. Basic blocks are the most familiar notion of atomic regions. The data in Table 1 shows that dynamic basic block size for a majority of the SPEC2000 integer benchmarks is below 9 instructions. The benchmarks were compiled using the Compaq Alpha compiler with a high level (-O4) of optimization including function in-lining and loop unrolling.

Benchmark	Average block size
bzip	9.17
crafty	9.23
eon	7.45
gap	8.52
gcc	6.43
gzip	11.07
mcf	5.33
parser	5.30
twolf	7.36
vortex	7.20
vpr	8.56

Table 1. Dynamic basic block size.

In this paper, we present an effective technique for generating longer atomic regions with the use of control flow *assertions*. An assertion is an instruction that verifies that the original branching conditions still hold. If the conditions are still true, then no action is taken. If they are not, then the entire region is undone and control is diverted to an

original copy of the code.

The atomic regions formed using our technique are called *frames*. A frame is a region of code where all internal branches have been promoted into assertions. Frame creation can be done statically by a profiling compiler, or dynamically with a hardware fill unit. We demonstrate that with a dynamic technique using branch correlation, frames can be very long—an order of magnitude longer than a basic block—with several properties that make them very compelling for further investigation.

In addition to the reasons mentioned earlier in the introduction, long atomic regions are useful for low-level dynamic translation and optimization, as exemplified by several recent proposals such as the rePLay Framework [10], the Transmeta Code Morphing System [5], and HP Dynamo [1]. An atomic region can serve as the basic unit of optimization. It can be as small as an instruction, but longer regions are preferred in order to give a dynamic optimizer greater opportunity for optimization. Further benefits are had if recently optimized regions occur frequently—the overhead costs of translation and optimization are amortized over each occurrence. While the frame construction techniques presented here are specifically tailored for rePLay, they can be extended for use by a variety of dynamic optimization schemes.

In this paper, we contribute the following. We present a technique for constructing logically atomic regions called frames by using control flow assertions. We measure the effectiveness of our construction technique when applied to static code versus applying it dynamically using branch correlation. We provide metrics for evaluating the effectiveness of frame construction. We propose and evaluate a hardware mechanism for constructing frames.

2 Basic concepts : assertions and frames

There are two basic concepts proposed in this paper: assertions and frames. An assertions is a type of branch instruction that has no explicit control flow associated with it [6]. An assertion verifies that certain conditions are true during execution, and initiates a recovery action if they are not. Frames are logically atomic blocks of instructions where all internal control flow has been replaced by assertions. In this section, we elaborate further on these concepts.

2.1 Assertions

A conditional branch instruction and an assertion instruction are similar in that they both test a condition. They are different, however, in the actions taken after the condition is tested. A conditional branch instruction will either divert the instruction stream to the taken target of the branch instruction if the condition is true, or allow the program to progress sequentially if the condition is false. An assertion does nothing if the condition is true. If the condition is false, however, the assertion triggers a recovery action and diverts control back to a recovery point. The recovery action involves reverting the architectural state to that of the beginning of the block that contains it. Essentially, an assertion that fires causes its entire block to be undone. We discuss the specifics of the recovery action later in this section.

This undoing of state creates an important distinction between a conditional branch and an assertion: subsequent instructions in the same block are not control dependent upon the assertion. An assertion therefore requires no prediction when fetched. An implicit prediction is made that the assertion will follow the direction the original branch instruction was biased towards.

We demonstrate the concept with an example. Figure 1 shows the difference between original code and code with assertions. The original code contains three basic blocks: BlockA, BlockFallThroughA, and BlockZ. BlockA contains a conditional branch that is taken to BlockZ. BlockA and BlockZ can be coalesced using an assertion. In Frame1, the instructions in BlockZ are not control dependent on the assertion, and can be safely moved ahead of the assertion. If the condition checked by the assertion is true, nothing happens. If it is not true, the entire block is flushed (i.e., architectural state is recovered back to the beginning of Frame1), and control is transfered to BlockA. We say that in this case, the assertion has fired.

```
BlockA:
        :
        BRz r3, BlockZ ; BR 1
BlockFallThroughA:
        :
BlockZ:
        :
        BRz r4, BlockK ; BR 2
        :
Frame1:
        :
        <insts from Block A and Z>
        :
        ASSERTz r3, BlockA
        :
        BRz     r4, BlockK ; BR 2
```

Figure 1. Example of a frame. BlockA, BlockFallThroughA, and BlockZ constitute the original control flow. Frame1 contains copies of Blocks A and Z joined by an assertion. If the assertion fires, control is diverted to BlockA.

We will demonstrate that using assertions in place of highly biased branches allows for the creation of large atomic regions (like Frame1). The objective is to promote conditional branches into assertions in situations where they are unlikely to fire.

The three general forms of a conditional assertion are shown below.

```
ASSERT   Rx, Ry, <cond>, assert_tgt
ASSERTi  Rn, <imm>, <cond>, assert_tgt
ASSERTil Rn, <long_imm>, <cond>, assert_tgt
```

All three versions compare a register with either a register, a short immediate value, or a long immediate value. A PC-relative assertion target (assert_tgt) specifies where control is to be redirected in the case the condition is not true. The conditional field can be any standard relational comparison (i.e., less than, less than or equal to, etc). Most ISAs only support conditional branches that compare a register with the value zero (i.e., the relational comparison is less than zero, less than or equal to zero, etc.). This is to allow high-speed implementation of branch execution logic; performing a register-to-register comparison and initiating a possible misprediction recovery in a single cycle at high frequencies can be problematic. Since the case of an assertion firing is by design the uncommon case, we allow two register values to be compared within assertions. As a result, the recovery due to a fired assertion might start a cycle after the comparison is done. There is no direct performance advantage in doing this, and this can be done with branches also. It does, however, allow the removal of an extra instruction in certain situations when converting from basic blocks into frames.

As we will show, our technique for converting branches into assertions also allows indirect branches to be converted. The third form shown above, `ASSERTil`, compares a register with a 32-bit (or 64-bit) immediate value, and therefore an `ASSERTil` takes the space of 2 (or 3) regular 32-bit instructions. Highly biased indirect branches or returns can be converted into assertions and their target blocks encapsulated with a frame. The address of the expected target is the immediate value field of the `ASSERTil` instruction.

2.2 Frames

A section of code in which all internal branches have been promoted into assertions is called a frame. A frame is an atomic region. If any instruction within the frame commits, then they all commit. Figure 2 shows how a likely path through a section of a program can be converted from original basic blocks into a frame.

The frame in Figure 2 has four assertions. These assertions test that the original branching conditions that would have taken program control from block A to block B to block C to block D to block E still hold. If they hold, then the frame completes. If any one of them do not hold, then an assertion will fire, the frame will be undone, and program control will transfer to the original block A and proceed from there.

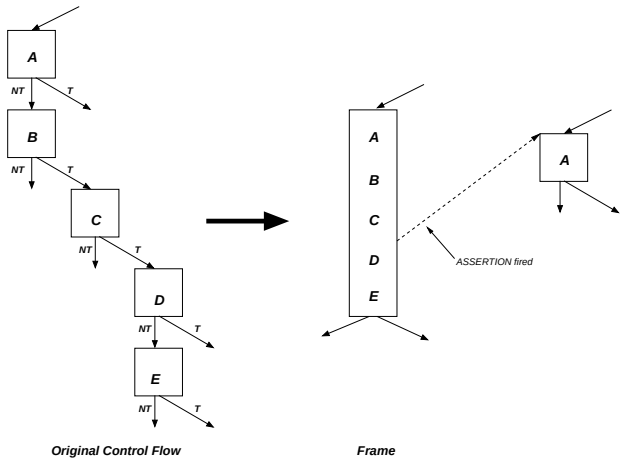

Figure 2. A frame is a region where all internal branches are promoted to assertions.

An optimization can be done in the mapping between branches and assertions. Assertions need only check for the most restrictive condition that must be true in order for a frame to execute. For example, if the branch at the end of block A tested for `(x < 10)` and the branch at the end of block B tested for `(x < 4)` then only an assertion to verify the condition `(x < 4)` is required (provided the value of `x` does not change in the interim).

Because a firing assertion can have a higher execution penalty than a mispredicted branch, frames should not be constructed unless the paths that they encapsulate are determined to have high likelihood of execution. The penalty of a firing assertion depends on two factors: (1) the dataflow depth of that assertion and (2) the efficiency of the processor in meeting that depth during execution.

Frames have a similarity to other types of regions identified by optimizing compilers, but are nonetheless different. Hyperblocks, superblocks, and traces from a trace scheduling compiler are not strictly atomic regions—all can have side exits or divergent paths. The use of the control flow assertion in frame construction alleviates an obvious limitation to region size imposed by atomicity. We will also demonstrate that frame construction can be carried out dynamically.

Recovery involves two things: (1) reverting architectural state back to what it was before the frame started execution, and (2) directing control back to the original (non-frame) version of the code. Reverting state is done using a state re-

covery mechanism similar to what is required for a deeply-pipelined dynamically-scheduled processor, such as checkpointing or a reorder buffer. A large store buffer is required to hold values stored to memory by instructions within a frame. Once the frame is determined to execute completely, the stores are committed to memory and the register values produced by the frame are committed to the architectural register set.

2.3 The rePLay Framework

The techniques presented in this paper can be applied directly to a hardware/software framework for dynamic optimization called rePLay [10]. In rePLay, frames are constructed by hardware using some of the techniques described in this paper. A software-driven optimization engine optimizes each frame before storing it within the frame cache. The atomic property of frames enables the optimization engine to perform aggressive optimization with lower overhead than if frames were non-atomic. A sequencer speculates through the control flow, initiating fetches of both frames and regular basic blocks. Figure 3 shows a high-level diagram of the rePLay framework.

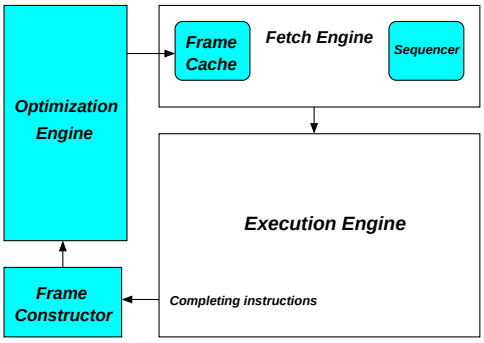

Figure 3. The rePLay Framework.

Since we are investigating frame construction for use with dynamic optimization, we are faced with two competing objectives: we want frames to be long in order to boost the potential of optimization, and we want frames to completely execute. In this paper, we examine frame construction techniques that achieve both.

3 Related Work

The fundamental elements of this work are derived from work done by Melvin and Patt on the Block-Structured ISA [6]. They proposed the concept of developing an ISA centered around atomic regions. In a similar vein to frame construction, trace scheduling [3] exploits infrequent branch paths by removing them from a trace and branching to compensation code if an infrequent path should have been executed. Assertions (and dynamic branch correlation) improve upon trace scheduling.

Much of this work builds upon previous trace cache research [11, 12, 9], in particular that of Branch Promotion [8]. Recently, Merten et al [7] have investigated identifying hot traces to focus the benefits of a trace cache-like mechanism. The one key difference between most previous trace cache work and this work is that here frames are considered atomic entities; traces in previous trace cache work could have side exits. Furthermore, we consider frames for dynamic optimization, and thus frames are required to be long. Trace caches were primarily investigated to boost instruction fetch bandwidth.

The concept of dynamic compilation and optimization is an emerging area. The desire to boost performance and efficiency by exploiting run-time behavior has spawned several alternative proposals [1, 5, 4, 10]. All of these systems rely on identifying good candidate regions for optimization. In this paper, we provide a region-identification mechanism that can be used by most of these run-time systems.

4 Experimental Model

4.1 Benchmarks

For this study, we used all but one of the SPEC2000 integer benchmarks. We omitted the benchmark *perlbmk* because of problems in running it within our simulation environment. All benchmarks were simulated to completion except the benchmark *vpr*, which was simulated for 1B instructions*. Table 2 shows the number of simulated instructions for each benchmark. For most benchmarks, we used modified versions of the input sets provided by SPEC in order to get benchmark instances that simulated completely in a reasonable amount of time.

All benchmarks were compiled using the Compaq Alpha C compiler DEC C V5.9 with optimization level 4. At this level of optimization, the compiler performs in-lining, loop unrolling, and code replication to eliminate branches.

4.2 Simulation Environment

Our simulation framework is built upon the Alpha instruction-level simulator provided as the core of the SimpleScalar 3.0 tool set. For the studies done in this paper, we use an instruction trace analyzer that emulates a frame constructor and models a frame cache and branch bias table.

*The benchmark vpr undergoes to two phases of execution (placement and routing). We cover all of the placement phase and part of the routing phase in our simulations.

Benchmark	Instructions	Input Set
bzip2	289M	modified SPEC test input
crafty	620M	modified SPEC test input
eon	609M	SPEC test input (cook)
gap	490M	modified SPEC test input
gcc	283M	jump.i -o jump.o
gzip	870M	modified SPEC test input
mcf	413M	modified SPEC train input
parser	508M	modified SPEC test input
twolf	574M	modified SPEC train input
vortex	265M	modified SPEC train input
vpr	1000M	SPEC test input

Table 2. Benchmarks used in simulations.

5 Evaluation

In this section we evaluate two techniques for frame construction. The first technique is based on a simple static analysis of branch behavior. Branches that are highly biased above a particular threshold are promoted into assertions. The second technique uses branch correlation to identify instances of branches for promotion.

Since we are proposing a frame construction technique for use with dynamic optimization, we have only considered frames above a minimum size. Small frames are unlikely to provide substantial benefit over basic blocks in terms of optimization opportunity, and instead can incur performance overhead that cannot be recovered. We therefore discard frames consisting of fewer than 3 basic blocks or fewer than 16 instructions from consideration. We also set an upper limit on frame size to accommodate restrictions imposed by real hardware (for instance, line size in the frame cache or number of outstanding stores in a store queue). Frames are truncated at the 256th instruction.

We use three primary metrics to evaluate our frame construction techniques: average dynamic frame size, frame completion rate, and coverage of the instruction stream. Average frame size is the average size in number of instructions of a frame measured over all committed frames. The frame completion ratio measures how likely a frame is to commit once issued. A frame does not commit if any of its assertions fires. The completion rate therefore is a measure of how often all assertions within a frame are correct. Frame coverage measures the fraction of the dynamic instructions that is derived from committed frames. For example, 80% coverage indicates that 80% of the i-stream came from instructions encapsulated within a frame.

5.1 Static frame construction

Static frame construction is performed by using a profiling compiler to first identify branches to promote into assertions. The compiler then promotes candidate branches and arranges their blocks into sequential frames, keeping the original copies to handle a firing assertion. An example of static frame construction is demonstrated in the example in Figure 1.

We evaluated a scheme for static frame construction by emulating an idealized compiler technique within our experimental framework. We first profiled each benchmark on a training input set to identify branches that are 97%[†] likely to go to a particular target. These candidate branches are treated as assertions in subsequent simulations of each benchmark on the measurement input sets listed in Table 2. In effect, we are modeling a compiler that is ideally able to promote every highly biased branch (conditional, indirect, and return) into an assertion and construct frames out of all paths containing sequences of 2 or more assertions.

Table 3 shows the average frame size, completion rate and coverage for each of the benchmarks. Also included is the number of unique frames generated by this static technique. With static frame construction, frames average 66 instructions in length, have a 97% probability of complete execution, and cover 50% of the instruction stream.

	Ave Frame Size	Completion Rate	Coverage	Assertions per Frame	Unique Frames
bzip2	137	91%	61%	15.9	1412
crafty	64	98%	42%	3.2	3954
eon	78	99%	57%	3.1	7210
gap	48	95%	53%	3.6	3844
gcc	37	99%	40%	3.6	21720
gzip	98	95%	59%	5.9	1423
mcf	93	96%	33%	6.1	1092
parser	33	99%	50%	4.1	3835
twolf	39	99%	54%	3.4	4497
vortex	58	99%	82%	5.2	8178
vpr	42	99%	18%	2.7	3428
Ave	66	97%	50%	5.2	5508

Table 3. Effectiveness of Static Frame Construction.

Figure 4 shows the distribution of frame sizes observed during execution, averaged over all benchmarks. Each bar represents a span of four sizes. For example the bar labeled 16 represents the dynamic frequency of frames of size 16, 17, 18, and 19 instructions. It indicates that frames of this

[†]We chose the 97% after investigating several thresholds. We selected one that maximizes size while not compromising completion rates.

size account for slightly over 9% of all frames. The distribution is wide, however the bulk of frames are between 16 and 48 instructions long.

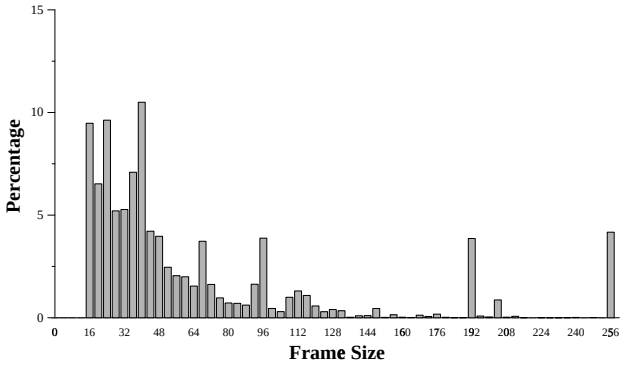

Figure 4. Distribution of statically-generated frame sizes at run-time.

5.2 Dynamic frame construction

A variety of basic research in branch prediction [14, 2] has provided substantial insights into the relationships among dynamic branch instructions. These studies have identified two types of basic correlation: *local correlation*, where a branch's current direction is highly correlated to its previous directions, and *global correlation*, where a branch's current direction is highly correlated to the direction of any previous branch or branches.

The dynamic techniques we explore in this section rely upon global correlation between branches to guide promotion from branches to assertions. Figure 5 provides a high-level view of the construction technique.

The frame constructor hashes (using XOR) the fetch address of each incoming block of committed instructions with the committed branch history to index into the branch bias table [8]. The bias table keeps track of whether the branch ending the block has gone in the same direction for a particular number of successive occurrences. If it has, the bias table indicates that the branch should be promoted. In our experiments, the bias table is configured to promote if the branch repeats its direction 32 consecutive times. Figure 6 shows the structure of the bias table. Once the 5-bit counter has saturated, the branch is promoted and the entire block is added to the frame construction buffer and the pending frame continues to grow. Once a branch is encountered that is not promoted, the block is added and the pending frame is considered complete. A separate bias table is maintained for indirect branches and returns. For such branches, a single bit for last direction does not suffice. A target address must be kept in each entry.

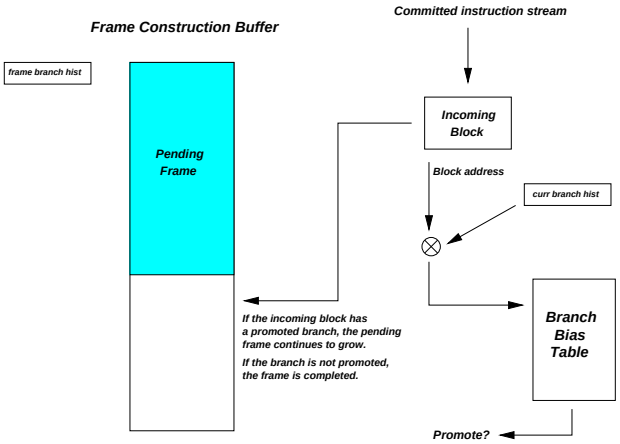

Figure 5. A hardware-based constructor that utilizes branch correlation.

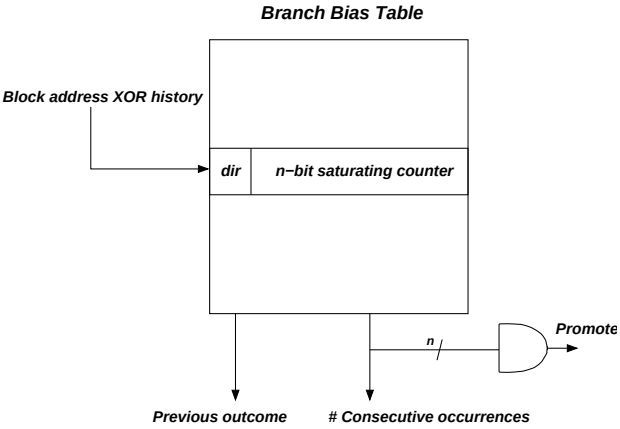

Figure 6. Branch bias table for conditional branches.

We also *demote* assertions back into branches when we detect that their behavior has changed. Using the branch bias table, we also track firing assertions to determine if they should be demoted back into branches. An assertion is allowed to fire once before it is demoted. A demoted assertion causes the frame containing it to be discarded.

The starting branch history of each frame (i.e., the committed history at the first branch 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 history of frame ABCDE is XYZ, then 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 *signature* for the frame and specifies when it 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 correlated to the outcomes of branches, or path, before it. The history used in the promotion decision helps separate branches into these biased instances.

We gathered branch information in two ways: global history and path history. Global history is a recording of the *n* most recent conditional branch outcomes. Path history is a recording of the *n* most recent branch target addresses. Global history can more compactly represent branch history because only a single bit is required to encode a branch direction. Path history is less compact. It requires more bits per target in order to uniquely identify the target from all others. In this way, the information stored in the path history can completely identify paths in cases where global history would be ambiguous. Also, path history can capture targets of indirect branches whereas global history cannot.

First we measure the fraction of all dynamic branches that are promoted into assertions as a function of path history length. Figure 7 demonstrates that as path history is increased beyond 6 targets, fewer than 20% of all dynamic branches actually remain as branches. The rest are promoted into assertions. Of these assertions, less than 0.5% ever fire. This data was collected using a bias table that promoted after 32[‡] consecutive similar occurrences.

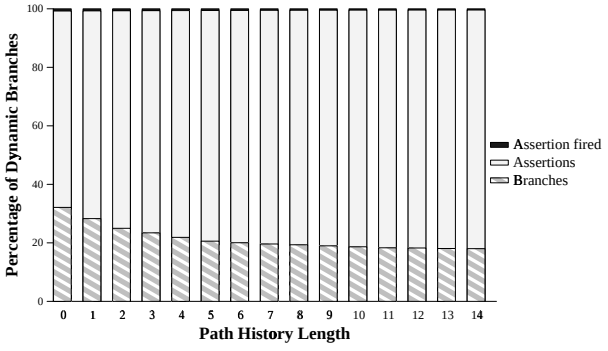

Figure 7. Fraction of dynamic branches converted into assertions. Bias threshold = 32.

Using an ideal version of this frame construction technique (i.e., a bias table that suffers no interference and an ideal hardware frame cache), we measured the effects of branch correlation on frame construction.

[‡]We use a threshold of 32 throughout this paper. After extensive studies on promotion thresholds, we determined that a threshold of 32 produces large frames with low assertion rates.

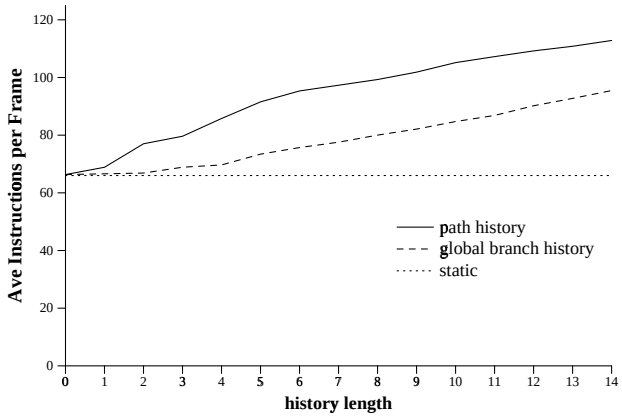

Figure 8. Average dynamic frame size as a function of history used in frame construction. Bias threshold = 32.

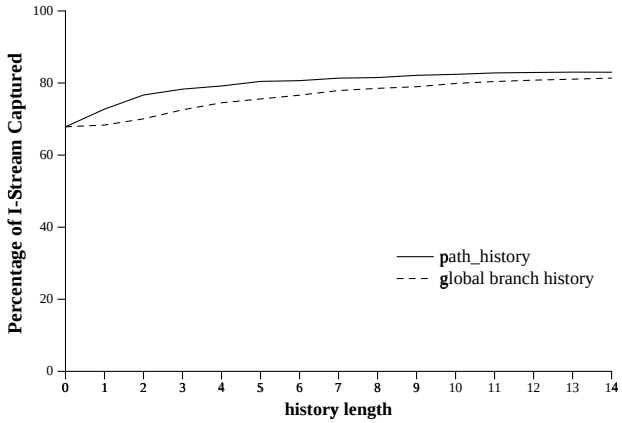

Figure 9. Frame coverage of the i-stream as a function of history. Bias threshold = 32.

Figure 8 demonstrates the average size of frames as measured on the benchmark set using both global branch history and path history. Figure 9 shows the coverage of the instruction stream. In these experiments, the bias table was configured to promote branches into assertions after 32 consecutive similar outcomes. The important trend is that even adding a small amount of branch correlation to the promotion decisions causes the size and coverage of the instruction stream increase. The completion rate of the frames remains nearly constant at 97% (this indicates that the per-assertion fire rate actually decreases because the average number of assertions per frame increases).

The data in Figures 7 and 8 indicate that decreasing the total dynamic branch count by even a small percent-

age causes a significant increase in frame size. This is because, after a certain critical number of branches have been promoted into assertions, promoting more branches causes adjoining frames to be coalesced into larger frames.

Average frame size serves as a gross summary of behavior. Firstly, this is because frame size has a wide distribution as demonstrated in Figure 10. There are small frames and very large frames (almost 12% of all frames are the maximum 256 instructions long). Also, each benchmark has its own characteristic distribution. Due to space constraints we have omitted the per benchmark distribution data here.

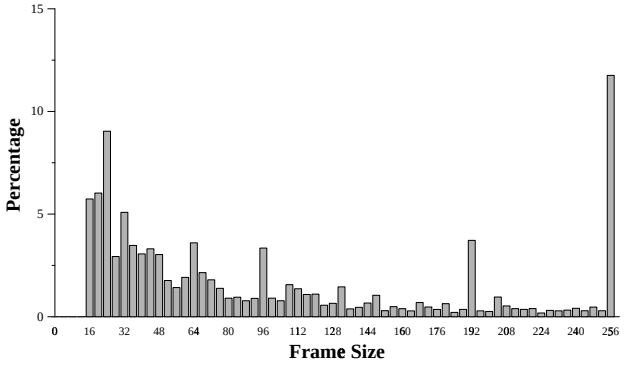

Figure 10. Distribution of dynamic frame sizes constructed using a 9 element path history.

We do however include a per benchmark average of the 9 element path history scheme on the three important metrics, plus the average number of assertions per frame, and the overall number of dynamically generated frames. Included for reference is the overall average of the static scheme and the dynamic scheme utilizing a 9-bit global history.

The frame properties resulting from path-history based frame construction are superior. One particular item of note is the relatively high number of unique frames generated via global history. We suspect this has to do with the ambiguity surrounding global history.

Overall, the results are promising. With a dynamic frame construction utilizing a 9 element path history, we are able to construct frames that span an average of 102 instructions, encapsulate over 9 branches, and have a 97% chance of complete execution. These frame characteristics make atomic frame construction useful for optimization. The reduction in dynamic branches opens opportunity for less complex fetch hardware. In the next section, we demonstrate that even with the simulated effects of finite hardware, our frame constructor is able to sustain good results.

While we have been calling this frame construction technique a dynamic frame construction technique because of its use of run-time branch information, Young et al [15] proposed a mechanism that can be adapted to exploit such dynamic information statically by creating duplicate versions of branches specific to an execution path.

	Ave Frame Size	Completion Rate	Coverage	Assertions per Frame	Unique Frames
bzip2	180	89%	79%	18.9	1108
crafty	88	96%	85%	6.7	15432
eon	179	98%	89%	8.6	1515
gap	155	98%	96%	15.4	5662
gcc	70	96%	77%	8.2	24687
gzip	89	95%	79%	5.6	1505
mcf	52	96%	71%	6.7	2097
parser	46	98%	78%	6.2	7629
twolf	66	99%	82%	6.8	2533
vortex	135	99%	94%	13.4	3273
vpr	61	99%	74%	4.8	2656
path	102	97%	82%	9.2	6191
global	82	97%	79%	7.4	13324
static	66	97%	50%	5.2	5508

Table 4. Per benchmark statistics for a constructor using a 9 element path history.

5.3 Hardware for frame construction

In this section, we examine the effects of using a finite sized branch bias table and a finite sized frame cache on the frame constructor.

In the first experiment, we examine the effects of bias table size. The data plotted in Figure 11 demonstrate the effects on frame size of using 16KB, 32KB, and 64KB bias tables. Also, each configuration uses a 4KB indirect branch bias table. The threshold for promotion was set to 32.

The bias table uses a 9 element path history maintained as suggested by Stark et al [13]. They proposed maintaining path history by XORing new targets into the path history and XORing old targets out. Along the way, each target is rotated to encode each target's position within the history. The number of bits selected from each target address depends on the size of the bias table. For example, a 32KB bias table uses 15 bits from each target address in forming the path history.

The frames generated by using finite sized bias tables peak at slightly over 80 instructions. The drop in frame length between a 64KB bias table and a 16KB bias table is significant but not severe.

Two things of note: First, the hardware frame constructor mechanism uses committed branch information and therefore requires no recovery mechanism for misspeculations as would a branch predictor in the frontend of a processor. Second, our bias table suffers from negative interference (as

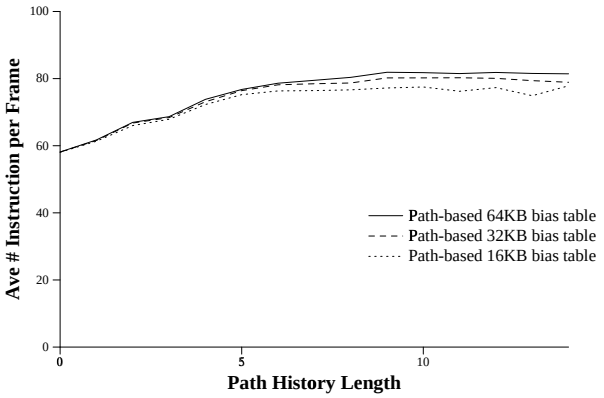

Figure 11. Dynamic frame size for various sized hardware bias tables.

demonstrated by the degradation from ideal to finite-sized). Many of the proposed interference reduction techniques explored for branch predictors such as filtering and agree prediction can be applied here to improve performance of the bias tables.

Next, we evaluate the effects of a finite sized frame cache. The data presented in Table 5 lists the results of using a 256 element frame cache with a 32KB branch bias table and a 4KB indirect branch bias table. Frame construction uses a 9 element path history. Promotion threshold is again set to 32 consecutive occurrences.

The data indicate that the constructor is able to coalesce almost 8 basic blocks together to form atomic regions of over 80 instructions, 7 of which are assertions. Almost 70% of the dynamic instruction stream is covered by these frames. These characteristics of frames not only present useful opportunity for dynamic optimization, but the increase in the span of branchless regions makes the job of a processor's fetch engine much simpler. A single fetch can produce 80 instructions with only a single branch prediction.

To give further context to the frame characteristics presented in Table 5, we also include the branch prediction accuracy of a small 4KB gshare using 14-bits of global branch history. Frame completion rates are high even though branch prediction accuracy is not. However, most benchmarks that suffer from low branch prediction accuracies also have smaller average frame sizes.

We measure frame cache size by the number of elements cached rather than number of bytes of storage because frame size is dependent on optimizations performed by an optimizer. Hand optimizations of high frequency frames in the SPECint95 benchmarks suite resulted in considerably smaller frames than we started with. When smaller frame cache sizes are evaluated, the primary change in metrics is a reduction in coverage. A 64 element frame cache gets a coverage of 58%, a 128 element frame cache gets 64%, and a 256 element frame cache gets 67%. Frame size remains nearly constant.

Degradation from ideal hardware is significant, but there is opportunity for increasing the effectiveness of a frame cache using the hot spot identification techniques of Merten et al [7]. They have determined that the behavior of frame-like regions of the control flow exhibit hot-cold behavior. At certain regions of execution, some frames are likely to be more frequently utilized than others. Using their technique it is possible to only cache frames that are detected to be hot, and to drop cold frames, and thereby use the limited capacity of the frame cache more effectively.

6 Analysis

In this section, we provide some insight into the frame construction techniques. We provide some ancillary data to shed light on the types of program behavior that are being exploited by the frame constructor.

6.1 Rationale behind what is happening

Frame construction, both dynamic and static, exploit biased branches. Based on our experimentation, we found that via profiling approximately 55% of all branches are categorized as promotable biased branches using the criterion we mention in Section 5.1. This number increases to 67% when the classification is done dynamically using a bias table. The number increases to over 80% when branch correlation is added to the classification.

The phenomenon being captured by the dynamic frame constructor is very similar to the phenomenon captured by a 2-level branch predictor. Take for example a string of correct predictions made by a 2-level predictor such as gshare. The initial correct prediction is made by indexing the predictor with the starting global branch history and a fetch address. At the end of the cycle the fetch engine provides a fetch block of instructions, a new global history, and a new fetch address. In the next cycle, the new history and fetch address index the fetch mechanism to produce another fetch block, history, and fetch address. This cycle continues until an event such as a branch misprediction, or BTB miss, or cache miss causes a disruption. The process begins with an initial history and an initial fetch address. The frame constructor *unfurls* this process by prepackaging the fetch blocks with the predictions; both can be known *a priori* by traversing history information stored in the bias table. Since the cost of an assertion can be higher than that of a branch misprediction, we use larger counters in the bias table than in the standard pattern history table (5-bit as opposed to 2-bit) to gain more confidence about branch behavior. The

	Average Frame Size	Completion Rate	Coverage	Assertions per Frame	Unique Frames	4KB gshare
bzip2	179	89%	78%	18.9	1151	97.4
crafty	75	96%	61%	5.3	13643	92.4
eon	87	98%	46%	4.1	1613	97.1
gap	114	96%	88%	11.1	7569	98.0
gcc	51	97%	36%	5.7	21033	89.4
gzip	89	95%	77%	5.6	1579	90.7
mcf	53	96%	68%	6.6	2051	90.1
parser	43	99%	69%	5.9	6946	93.1
twolf	56	99%	67%	5.6	3569	90.4
vortex	89	98%	76%	8.4	5769	97.6
vpr	52	99%	75%	4.1	3085	85.0
Average	81	97%	67%	6.9	6182	
Ave - Ideal Dynamic	102	97%	82%	9.2	6191	
Ave - Ideal Static	66	97%	50%	5.2	5508	

Table 5. Frame stats with 256 entry frame cache, 32KB+4KB bias tables, and 9 element path history.

starting address and starting branch history serve as a fetch signature for a frame created with this technique.

6.2 Is it simply loop unrolling?

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

We explored the effects of compiler loop unrolling on the frame constructor by running an experiment with binaries generated with loop unrolling disabled. Table 6 presents the results. The table contains the average across all benchmarks for a frame constructor utilizing a 9 element path history. The first two data rows of the table present the results with compiler unrolling enabled and with it disabled. The data in the third row was measured using a frame constructor that was inhibited from adding duplicate blocks to a pending frame (i.e., if a frame already contained block X, then the frame would be considered complete if another copy of X were attempted to be added). This is a very severe way of restricting the effects of loop unrolling because it factors out loops that would be otherwise difficult for a compiler to unroll, such as loops with complex control paths or function calls. This test was run on binaries generated with compiler unrolling enabled.

Based on the data collected with loop unrolling disabled, the effect of compiler unrolling on frame construction is minimal. The frame constructor exploits loops, as demonstrated by the sharp drop in frame size and coverage when duplicate blocks are inhibited. Nonetheless, frame size and coverage is still substantial.

	Ave Frame Size	Completion Rate	Coverage	Unique Frames
W/ unrolling	102	97%	82%	6191
W/o unrolling	105	96%	90%	6774
No duplicates	74	98%	71%	4581

Table 6. The effects of loop unrolling on frame construction using a 9-element path history.

6.3 Phased behavior

Static construction relies on profiling and is therefore brittle to the difference in behavior between the execution profiles and actual execution. Dynamic construction, with extra hardware costs, can adapt to actual execution behavior.

Another benefit of dynamic frame construction over static construction is the ability for frames to be generated and destroyed depending on the dynamic behavior. A branch may be biased during a section of a program, and not biased during another. Such phased behavior is more easily exploited by a dynamic mechanism than a static one.

We measured the dynamic variation in branch promotions throughout the execution of each benchmark. For every branch instance (branch preceded by specific path) that executed at least 3200 times (this number was chosen because our promotion threshold is 32, thus the warmup cost is a smaller factor), we counted how often the branch in-

stance was observed as an assertion and how often it was not. We found that a majority of such branches were encountered as promoted only between 90% and 95% of the time, indicating that there are periods of execution where these branches have irregular behavior.

7 Conclusion

Frame construction using assertions creates large atomic regions of instructions that have a very high probability of complete execution. We demonstrate that incorporating branch correlation into the branch promotion decision results in larger frames with a larger degree of coverage of the instruction stream, even when finite sized hardware is used for frame construction and frame caching.

We submit that the dynamic frame constructor is pre-packaging the instructions associated with easy-to-predict branches into a frame, leaving the harder-to-predict branches as the connective branches between one frame and the next. From a hardware standpoint, this is good because with a single fetch, several cycles worth of instructions can be streamed out of the frame cache allowing multiple cycles for the prediction of these connective branches.

Our analysis demonstrates that the frame constructor is able to unroll loops and in-line function calls in situations difficult for a compiler to exploit. In addition to unrolling, the frame constructor is able to exploit run-time control stability in paths that contain no loops as well.

We view these results as preliminary; they are the first step for rePLay, which is a hardware/software framework for dynamic optimization. Frames serve as the regions of optimization within rePLay in the same way that a trace is the basic unit within a trace scheduling compiler. Frames are different from superblocks and hyperblocks in that they contain only a single path of execution and no side entrances or side exits. This gives a dynamic optimizer with greater leeway in performing low-overhead optimizations.

8 Acknowledgments

We thank both the other members of the Advanced Computing Systems group and Prof. Steve Lumetta for their valuable insights in the development of these ideas. We also thank Intel and Hewlett-Packard for their generosity in providing equipment.

References

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

[2] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52 – 61, 1998.

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

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

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

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

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

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

[9] S. J. Patel, D. H. Friendly, and Y. N. Patt. Evaluation of design options for the trace cache fetch mechanism. *IEEE Transactions on Computers*, 48(2):435–446, Feb. 1999.

[10] S. J. Patel and S. S. Lumetta. rePLay : a hardware framework for dynamic program optimization. Technical Report CRHC-99-16, University of Illinois Technical Report, Dec. 1999.

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

[12] E. Rotenberg, S. Bennett, and J. 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.

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

[14] T.-Y. Yeh and Y. N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, 1991.

[15] C. Young and M. 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.