Improving Region Selection in Dynamic Optimization Systems

David Hiniker
Microsoft Corporation

Kim Hazelwood
University of Virginia

Michael D. Smith
Harvard University

Abstract

The performance of a dynamic optimization system depends heavily on the code it selects to optimize. Many current systems follow the design of HP Dynamo and select a single interprocedural path, or trace, as the unit of code optimization and code caching. Though this approach to region selection has worked well in practice, we show that it is possible to adapt this basic approach to produce regions with greater locality, less needless code duplication, and fewer profiling counters. In particular, we propose two new region-selection algorithms and evaluate them against Dynamo's selection mechanism, Next-Executing Tail (NET). Our first algorithm, Last-Executed Iteration (LEI), identifies cyclic paths of execution better than NET, improving locality of execution while reducing the size of the code cache. Our second algorithm allows overlapping traces of similar execution frequency to be combined into a single large region. This second technique can be applied to both NET and LEI, and we find that it significantly improves metrics of locality and memory overhead for each.

1 Introduction

A *dynamic optimization system* observes execution patterns in a running program and optimizes the frequently executing code to improve program performance. Dynamic optimization systems fall into three general categories based on their primary function: transparent optimization, just-in-time compilation, and binary translation. A transparent optimizer takes a binary executable for a target processor and re-optimizes it based on run-time information. A just-in-time compiler takes a machine-independent program and compiles it for a target processor. A binary translator takes an executable compiled for an incompatible processor and translates it to an equivalent program for a target processor. In each case, effective selection and optimization of the portions of the code that are expected to execute frequently is critical for limiting system overhead and achieving good performance.

In general, we use the term *region* to describe the portion of code that a dynamic optimization system selects for optimization. The two types of regions commonly selected by existing systems are a *whole method* and a *trace*. Just-in-time compilers, such as the Jikes™ Research Virtual Machine (Jikes RVM) [1, 2], are often organized around the optimization of whole methods. The design of these systems reflects the traditional focus of compilers on procedures and other programmer-defined units of code. In contrast, systems such as Dynamo [3, 4], DynamoRIO [6], Mojo [7], Adore [15], Transmeta's CMS [9] and the Binary-translated Optimization Architecture (BOA) [12] are designed around the optimization of traces. These systems focus on the interprocedural program paths (i.e., traces) that are executed during a program run. Ideally, region selection and optimization in such systems are not limited by static code organization that is meant to aid the programmer. In this paper, we propose and evaluate two improvements to region selection for trace-based dynamic optimization systems.

To understand how we might improve upon trace-based region selection, it helps to understand why traces have been such a popular choice. The traces constructed by most existing dynamic optimization systems are interprocedural superblocks [11], regions with a single entry point and multiple exit points. Such regions have been shown to be very simple to construct and optimize, thus incurring little run-time overhead. Furthermore, by placing frequently executed code together in consecutive memory locations, traces provide a range of instruction fetch benefits. Finally, prior research [5] has shown that programs often follow a small number of "hot" paths. Identifying and optimizing these hot paths has been crucial to the performance of dynamic optimization systems. In summary, a good algorithm for region selection must accurately identify the frequently executed application code, select the regions of frequently executed code with little run-time overhead, and produce optimized code in the dynamic optimization system's code cache that exhibits excellent locality and fetch characteristics.

Duesterwald and Bala [10] proposed one such trace-based region selection technique called *Next-Executing Tail* (NET). It exhibits all of the desirable features described above and thus has been particularly popular—it is used as the trace-selection mechanism in Dynamo, DynamoRIO, and Mojo. Because of its popularity and good performance, we use NET throughout this paper as the baseline of our investigations and experimental results. However, the shortcomings that we identify are not unique to NET; these problems appear in all other trace-selection mechanisms known to us.

When studied closely, all existing trace-selection algorithms suffer from two orthogonal problems: the problem of *trace separation* and the problem of *excessive code duplication*.

The problem of trace separation is that related code paths in the original program are selected as separate traces and placed far apart in the code cache. Once a trace has been selected, there is a delay during which a related trace is identified as executing frequently. During this time, traces from other parts of the code are likely to be selected and promoted to the code cache. Once a related trace is selected, it is inserted far from the original trace, potentially on a separate virtual memory page. Separation degrades performance because it reduces locality of execution—and therefore instruction cache performance—as control jumps between distant traces.

The problem of excessive code duplication stems from the fact that related paths often have code in common, so selecting isolated traces results in duplication of this code. In some cases, duplication is beneficial because it improves locality of execution, just as method inlining uses duplication to improve performance. However, we identify and quantify one type of trace duplication—tail duplication associated with early trace exits of unbiased branches—where the code duplication costs greatly outweigh any benefits of the resulting path separation. Excessive code duplication stresses the memory system, and it can also degrade the performance of a dynamic optimization system because it increases the overhead of code translation and optimization.

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

The importance of these two problems will grow as the software industry continues to produce executables with an increasing number of frequently executing paths. As shown by Ball and Larus [5], the number of paths that comprise 90% of execution in modern commercial software is often one to two orders of magnitude greater than in the standard benchmark programs used to develop NET. As the number of related paths grows, the extent of trace separation and the amount of code duplication grow with it.

To address these problems, we describe and evaluate two new, low-overhead region-selection algorithms. Our first algorithm, called *Last-Executed Iteration (LEI)*, selects traces that correspond to cyclic application paths.[1] Current trace-selection algorithms do not directly search for cyclic paths, so they often split frequently executed paths across multiple traces, and they duplicate code from inner loops in the trace for an outer loop. Our second algorithm, called *trace combination*, produces regions that may include multiple related paths. Trace combination observes multiple paths during trace selection and combines those that share code and execute frequently. By combining such paths, it is able to eliminate excessive code duplication resulting from paths that split at an unbiased conditional branch and later rejoin. We argue that combining traces that share an unbiased branch improves memory performance and enhances opportunities for loop-based and profile-driven optimization.

This paper has four main contributions:

- We identify and quantify problems of trace separation and excessive code duplication in the popular NET trace-selection algorithm.
- We describe and evaluate a low-overhead algorithm—LEI—for selecting traces that correspond directly to cycles in the executing application. Building more cycle-spanning traces significantly reduces the problem of trace separation. This algorithm also avoids duplicating nested cycles, which results in less code duplication than under NET.
- We describe and evaluate an efficient algorithm—trace combination—for building a region with one or more interprocedural paths. This algorithm significantly improves locality of execution, and it does so while reducing the amount of code cached. Since trace combination is independent of the method of trace selection, we demonstrate that it reduces the problems of separation and duplication for both NET and LEI. Moreover, each improvement is larger for LEI than for NET, which suggests that our algorithms are especially effective when combined.
- Through simulation, we show that LEI (and LEI with trace combination) requires fewer traces and less code cache space than NET (and NET with trace combination) to cover 90% of the instructions executed in our benchmarks. This "90% cover set" metric has been found to correlate well with the performance of real dynamic optimization systems [4]. Along with our arguments about the low overhead of our algorithms, these results provide strong evidence that our algorithms will improve the performance of real systems.

We begin in Section 2 with a brief description of the NET trace-selection algorithm, how it exemplifies the problems of trace separation and excessive code duplication, and the metrics and methodology used to evaluate all region-selection techniques. Sections 3 and 4 develop and evaluate our LEI and trace-combination algorithms. Section 5 discusses other existing trace-selection algorithms, and Section 6 summarizes.

[1] A *cyclic path* is simply a path that ends with a branch to its beginning.

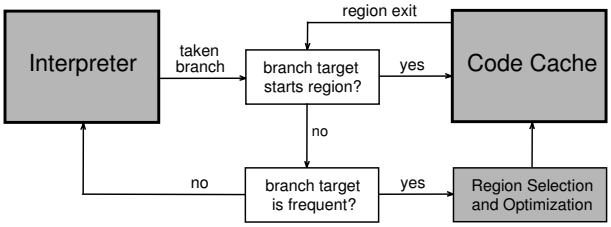

Figure 1. *Overview of execution under a dynamic optimization system. Shaded boxes correspond to parts of the system that emulate or execute original program instructions.*

2 Background and Methodology

Section 2.1 describes the dynamic optimization system we simulate and the NET trace-selection algorithm we use as a baseline for comparison. Section 2.2 then presents three illustrative shortcomings of NET with respect to the problems of trace separation and excessive code duplication. We end in Section 2.3 with a brief discussion of the experimental methodology used in the later sections.

2.1 Dynamic Optimization Setting

Although we identify principles of region selection that are not limited to a single setting, we develop algorithms with a software-based dynamic optimization system in mind. Figure 1 highlights the architecture we consider, which has been used by several dynamic optimization systems including Dynamo [3, 4], DynamoRIO [6], Mojo [7], and the Binary-translated Optimization Architecture (BOA) [12].

A dynamic optimization system begins executing a binary by emulating it in an *interpreter*. As the system emulates the binary, it gathers profiling information about which portions of code execute frequently. This allows it to decide when and where to start selecting a region. In some cases, this information is also used to select the basic blocks that comprise the region.

At every interpreted taken branch, the system decides whether to switch from emulating the code to executing a region from the code cache. If the branch target corresponds to an optimized region in the code cache, the system starts executing that code. If not, the system uses two criteria to decide whether a new region should be selected and added to the cache. First, the branch target must be allowed to begin a region. Restricting which branch targets are considered reduces profiling overhead and code duplication. Second, the branch target must have executed many times, as this suggests that it will continue to execute many times. This is implemented by associating a counter with the target of any taken branch that is allowed to begin a region. When the counter exceeds a predefined threshold, a region is selected beginning at the branch target. If the branch cannot begin a region or the count does not meet the threshold, control returns to the interpreter.

NET selects a trace that begins at the target of one of two types of branches: a backward branch or an exit from an existing trace. A *backward branch* is simply an instruction that transfers control to a lower address, and it often corresponds to the back edge of a loop. In this way, NET attempts to select traces that begin at loop headers. If a loop has more than one frequently executed path—or if NET does not identify the dominant path initially—an exit from the first trace will execute frequently and NET will select a second trace that begins with its target.

The execution count threshold for trace selection in NET is 50. When this threshold is reached, NET selects a trace by interpret-

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

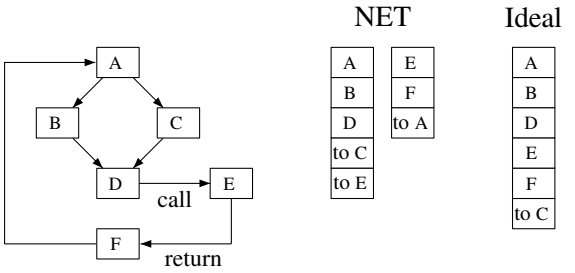

Figure 2. *The control flow graph on the left contains a loop with a function call on its dominant path (ABDEF). NET requires two traces (ABD and EF) to span the cycle. In this diagram, we assume that the function beginning with E is at a lower address, so the call is a backward branch rather than the return. Ideally, only one trace would be selected, and it would require two fewer exit stubs (which are represented as trace blocks labeled "to X").*

ing and copying the path that is executed next. Although the path selected is not based on any accumulated information, it is statistically likely to execute frequently [10]. The trace continues to extend along the interpreted path until a backward branch is taken, a branch is taken that targets the start of another trace, or a size limit is reached. These rules allow a NET trace to extend across procedure calls or returns.

Once a trace has been selected, an exit stub is created for each side exit and placed at the end of the trace, leaving the selected blocks contiguous in memory. To avoid the overhead of switching to and from the interpreter, the branch to any exit stub that leads directly to another cached region is rewritten to jump directly to that other region.

2.2 Three Shortcomings

Loops. When a loop in a program executes frequently, locality of execution is much better if the entire path is contained in a single region rather than split among several regions.[2] We say that a region *spans a cycle* when repeated execution of the cycle remains in the region. For regions that are traces, this means that the trace contains all blocks in the cycle, including one that ends with a branch to the top of the trace.

On the surface, NET seems well-suited to selecting traces that span cycles. Its profiling focuses on the targets of taken backward branches because they often correspond to loop headers [3]. Moreover, traces can cross procedure boundaries and include any forward branch, so an indirect jump or call will not necessarily end a trace before it reaches a loop back edge.

However, the definition of NET implies that a single trace cannot span an interprocedural cycle. NET selects an *interprocedural forward path*, as defined by Duesterwald and Bala [10], that ends with a call to a function at a lower address or return from a function at a higher address. NET, therefore, cannot extend a trace across both a function call and its corresponding return. As shown in Figure 2, if a loop contains a function call on its dominant path, then NET will form two separate traces that include two unnecessary exit stubs. Our experiments indicate that stopping at a backward function call or return enables NET to limit code expansion, but it prevents any interprocedural cycle from being spanned.

Nested loops. Another common but generally undesirable tendency of NET is to duplicate the beginning of an inner loop in a trace for an outer loop. Although NET ends a trace when a taken branch jumps to an existing trace, with nested loops control often falls into the inner loop. Figure 3 illustrates trace selection performed by NET on a simple fragment of code consisting of a pair of nested loops. Block B is selected first as the target of its own backward branch. After B exits to C 50 times, C is selected, but the trace does not extend to A because CA is a backward branch. Only after C exits to A does A start a trace, which extends until the next backward branch to include another copy of B.

[2] Bala et al. [4] make a similar observation by considering an example in which a program consists of one path through a large loop: an algorithm that selects the entire path as a single region will outperform one that splits the path into multiple regions.

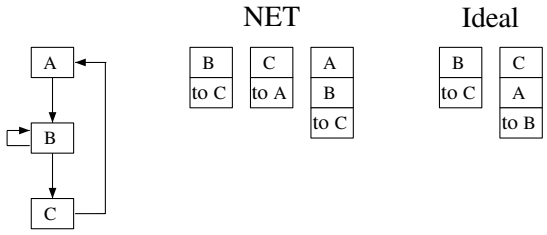

Figure 3. *Traces selected for simple nested loops. NET selects three traces and duplicates the inner loop. An ideal trace-selection algorithm would avoid duplication of the inner loop and separation of the outer-loop blocks.*

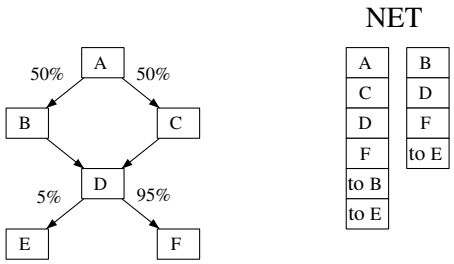

Figure 4. *Traces selected by NET for an unbiased branch (ending A) followed by a biased branch (ending D), assuming that the path through block C is selected first. The unbiased branch targets are separated, and two blocks and an exit stub are duplicated.*

In contrast, separation and duplication are both reduced by selecting traces based on cycles B and CA. B is selected first as a single-block cycle, and the second trace begins at its exit to C. This trace continues with A but ends because the next block on the path, B, is the start of a region and therefore a nested loop. Here, if a trace is allowed to extend across a backward branch, fewer blocks are selected and divided among fewer traces. Chen et al. [7] discuss a similar example and come to a similar conclusion, but do not appear to have a solution that achieves the ideal.

Unbiased branches. An unbiased branch is an inherent problem for a trace selection technique like NET, because a trace can contain only one of the branch targets. The result is that frequently executed paths from an unbiased branch are split into separate traces. In addition, if these traces reach a common join point, all blocks following this join point are duplicated.

Figure 4 shows a control-flow graph for an unbiased branch that is followed by a biased branch, where the label on each edge from a split point indicates its probability of being taken. The right side of the figure shows the traces that NET would select from it, illustrating separation and duplication caused by an unbiased branch. It begins by selecting a trace for one direction of the first conditional and later selects a second trace as the other direction is taken. Notice that the second conditional—consisting of blocks D and F and an exit stub to E—appears in each trace.

Section 4 describes an algorithm that combines equally likely paths into a single region. This region contains no duplication and allows control to remain in the region regardless of which unbiased

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

branch target is taken. The exit stub to block B is replaced by the block itself, and there is no need to duplicate the exit stub to block E.[3]

Clearly, not all regions should contain multiple paths. If there is a single dominant path beginning at a profiled block, it should be selected as a trace and no additional paths should be added. Adding rarely executed blocks would increase the overhead of optimization and the strain on the memory system without providing an offsetting improvement in execution time for the region. Therefore, an algorithm that allows a region to contain multiple paths should still be able to detect a dominant path and form a single trace for it.

2.3 Methodology

To determine the effect of our changes to region selection, we created a framework for simulating a dynamic optimization system and implemented each of our region-selection algorithms within it.[4] The framework relies on the Pin dynamic instrumentation system [16] to report the sequence of basic blocks executed by a program. For each executed branch taken, it simulates the actions of the dynamic optimization system described in Section 2.1.

Our framework assumes an unbounded code cache. As we report in the following sections, our region-selection algorithms should help improve the performance of dynamic optimization systems with bounded code caches, because our algorithms reduce code duplication and produce fewer cached regions. This improves memory performance, reduces the overhead of cache management, and regenerates fewer evicted regions. Detailed investigation of these effects, however, is outside the scope of this paper.

We present data for running each of the twelve SPECint2000 benchmarks to completion on its `test` input [19]. When there are multiple `test` inputs for a benchmark, the input that requires the program to execute the most instructions is used.

For each algorithm, we begin with metrics that illustrate improvements over NET in trace separation and excessive code duplication. We also quantify the additional costs of each approach.

To estimate the effect of a trace-selection algorithm on the overall performance of a dynamic optimization system, we adopt the "trace quality metric" used by the implementers of Dynamo, who designed the NET algorithm [4]. They define the *X% cover set* of a region-selection algorithm to be the smallest set of regions that comprise at least X% of program execution. In evaluating four region-selection algorithms, the authors found that the 90% cover sets were a perfect predictor of performance: a smaller 90% cover set implied a smaller execution time.

To measure how well a region-selection algorithm predicts the code that will execute frequently, we compute the *hit rate* and amount of *code expansion*. The hit rate for a program is the percentage of executed program instructions that execute from the code cache. The amount of code expansion is the number of program instructions that are copied into the code cache. We focus on this metric rather than the overall size of the cache because it measures the amount of work done by the optimizer, but at times we also consider the number of exit stubs the regions require.

To measure how well a region-selection algorithm selects code that executes together, we count the number of *region transitions*. A region transition is a jump between regions in the code cache, which are often far apart. Fewer region transitions implies better locality of execution.

[3] This reduction in exit stubs is significant. Hazelwood [14] shows that, in DynamoRIO, exit stubs occur roughly every six instructions and each requires at least three instructions. This implies that exit stubs usually comprise over one-third of the instructions in the code cache.

[4] All details of region selection have been abstracted out of the framework, allowing us to gather data for each region-selection algorithm without modification.

```
INTERPRETED-BRANCH-TAKEN( history buffer Buf , addr src , addr tgt )
 1  address cached ← HASH-LOOKUP( code cache , tgt)
 2  if cached
 3    then jump cached
 4
 5  CIRCULAR-BUFFER-INSERT(Buf, src, tgt)
 6  if HASH-LOOKUP(Buf.hash, tgt)
 7    then history buffer loc old = location of tgt in Buf
 8         HASH-UPDATE(Buf.hash, tgt, last element in Buf )
 9         if tgt ≤ src or old follows exit from code cache
10           then increment counter c associated with tgt
11                if c = T_cyc
12                  then newT ← FORM-TRACE(Buf, tgt, old)
13                       remove all elements of Buf after old
14                       recycle counter associated with tgt
15                       jump newT
16
17    else HASH-INSERT(Buf.hash, tgt, last element in Buf )
```

Figure 5. *Last-Executed Iteration Algorithm.*

3 Last-Executed Iteration Trace Selection

In this section, we focus on applying the principle that effective traces span cycles but avoid duplicating nested cycles. We showed in the previous section that NET cannot span an interprocedural cycle, and prior work [13] has shown that such cycles account for a large percentage of execution in some benchmarks. To address this, we develop a trace-selection algorithm that explicitly selects cycles and does not restrict the type of branch a trace can include. We evaluate our algorithm relative to NET and find that it reduces both separation and duplication while requiring significantly less memory for profiling.

3.1 Last-Executed Iteration Algorithm

Last-Executed Iteration (LEI) selects cyclic traces based on a history buffer containing the most recently interpreted taken branches. If the target of a branch is in the buffer, a cycle has executed and the buffer contains its path. This just-executed cycle is considered for optimization and promotion to the code cache.

A trace is only formed from the cycle if it meets criteria similar to those of NET. As we want a trace to begin at a loop header or grow from an existing trace, only a cycle that is completed with a backward branch or follows an exit from the code cache is considered. As we want a trace that executes frequently, only after the branch executes a predefined number of times, T_{cyc}, is a trace selected. Like NET, this requires additional memory for counter variables, but only for a small subset of taken branch targets.

When a counter reaches the execution threshold, T_{cyc}, a trace is selected from the cyclic path specified by the history buffer. Given the address and target of each taken branch, the full path is reconstructed by repeatedly appending the instructions between the target address of the current branch and the source address of the next branch. The trace ends when a cycle is completed or the next instruction begins an existing trace. This second condition allows LEI to avoid duplicating the first iteration of an inner cycle, even on a fall-through path.

Figure 5 gives the LEI algorithm, which is invoked when an interpreted branch is taken. Like NET, it first checks to see if the target is in the code cache, and if so transfers control to it. If not, line 5 inserts it into the branch history buffer. In order to make searching for a target in the history buffer efficient, a hash table of all targets currently in the buffer is maintained. Line 6 uses this hash table to check for a cycle, and lines 8 and 17 update the hash table to refer to this new occurrence of the target. If the target completes a cycle, line 9 determines whether it can begin a trace. If so, its counter is incremented and a trace is selected when the

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

```
FORM-TRACE( history buffer Buf , addr start , history buffer loc old )
 1  trace newTrace ← ∅
 2  address prev ← start
 3  for each branch in Buf after old
 4  do for each inst in fall-through path from prev to branch.src
 5     do
 6        // Stop if next instruction begins a trace
 7        if HASH-LOOKUP( code cache , inst)
 8          then break
 9        newTrace ← newTrace ∪ {inst}
10
11     // Stop if branch forms a cycle
12     if branch.tgt ∈ newTrace
13       then break
14
15     prev ← branch.tgt
16  return newTrace
```

Figure 6. *Algorithm for forming an LEI trace given a history buffer and a starting address.*

counter reaches T_{cyc}. Once the trace is selected, the corresponding branches in the history buffer are removed and its counter is made available for reuse.

Figure 6 shows how LEI uses the history buffer to form a trace. The loop beginning on line 3 iterates over each taken branch in the current cycle to determine the executed path. For a taken branch, all instructions from its target to the address of the next taken branch must have been on the path of execution. The loop beginning on line 5 copies each instruction from the path into the trace, and stops when one begins an existing region. If one is not found, the trace grows until it completes a cycle on line 12.

Although LEI maintains enough information to select cycles, its runtime overhead remains comparable to that of NET. Each algorithm is only invoked when an interpreted branch is taken, and in all benchmarks over 98% of execution occurs natively from the code cache. On each taken branch, both algorithms do a constant amount of work: each performs a hash table lookup to determine if the target is in the code cache, and each potentially updates a counter. LEI adds only one buffer insertion and one hash table lookup for the history buffer, as all other modifications of the hash table can be combined with this lookup.

Consider the example described in Section 2.2: a loop that contains a function call on its dominant path. Whereas NET selects the two traces in Figure 2, LEI selects the ideal trace that spans the cycle. This reduces the problem of separation, as future iterations remain in the same trace. It also reduces the size of the code cache, as fewer exit stubs are required.

This example highlights several important properties of LEI. The branch history buffer contains only taken branches, so the fall-through path from A to B is not included. Relatively few branch targets require counters; here A does but D does not. Once A is entered into the buffer a second time, the hash table is updated to refer to its most recent entry. Between its entries, the branch history buffer represents the full interprocedural cycle.

3.2 Experimental Results

Using the simulation framework described in Section 2.3, we compare LEI to NET with two main questions: how well does LEI select traces that span cycles, and what effect does this have on separation and duplication. For all of our performance metrics, we find that LEI produces more effective traces than NET.

For these comparisons, we use an execution threshold of 50 for NET, which is the published standard [3, 4, 10]. As LEI counts only certain executions of a backward branch—those where the branch target exists in the history buffer—a smaller value should be used, and we choose 35. We set the size of the history buffer

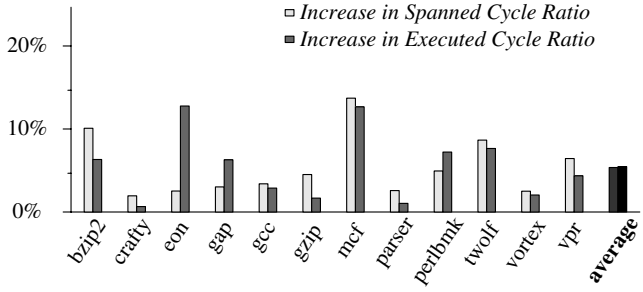

Figure 7. *The improvement of LEI over NET in selecting traces that span cycles. The lighter bars show the increase in the spanned cycle ratio (a measure based on what traces are selected). The darker bars show the resulting increase in the executed cycle ratio (a measure based on how traces execute).*

to be 500. Intuitively, this seems small enough to require little memory but large enough to capture very long cycles and those with frequently executing nested cycles.

The hit rate is slightly lower for LEI than for NET in most of the SPECint2000 benchmarks, but it remains above 99% for all but two. Only for these two benchmarks is the difference in hit rate more than 0.2%, with `mcf`'s falling from 99.80% to 98.31% and `gcc`'s from 99.37% to 98.98%. Lowering the execution threshold, as is done by Chen et al. [7], could compensate for this difference, but we do not attempt to tune this parameter without run-time measurements.

3.2.1 Spanning Cycles

We first evaluate how well LEI achieves its primary goal of improving on NET's ability to span cycles. To measure this, we compute both the *spanned cycle ratio* and the *executed cycle ratio* of LEI relative to NET. The spanned cycle ratio is the percentage of selected traces that include a branch to the top of the trace. The executed cycle ratio is the percentage of trace executions that end by taking a branch to the top of the trace, thereby executing the entire spanned cycle. Together, they measure how many additional cycles LEI spans and how much of an effect these cycles have on locality of execution.

Figure 7 summarizes the effect of LEI on these two metrics. For all benchmarks, LEI spans more cycles than NET, raising the overall proportion of cycle-spanning traces by nearly 5%. This meets our expectation, as allowing interprocedural forward paths to include backward function calls and returns increases the number of potential cyclic traces. The additional cycles increase the proportion of executed cycles in all cases, and in general the two metrics are highly correlated. This confirms that spanning more cycles improves the behavior of execution in the code cache, as fewer exits are taken to the interpreter or to a separated region.

3.2.2 Code Expansion and Locality of Execution

Figure 8 shows that LEI succeeds in reducing both separation and duplication: it produces less code expansion than NET while improving locality of execution in the code cache. For all benchmarks but `crafty`, the elimination of cycle duplication outweighs the effect of selecting longer traces across forward and backward branches, so on average LEI results in 92% of the code expansion of NET. Despite copying fewer instructions into the code cache, the average size of a trace is larger (increasing from an average of 14.8 to 18.3 instructions over all benchmarks). Together with the increase in executed cycles, this causes a significant improvement in locality of execution: on average the number of region transi-

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

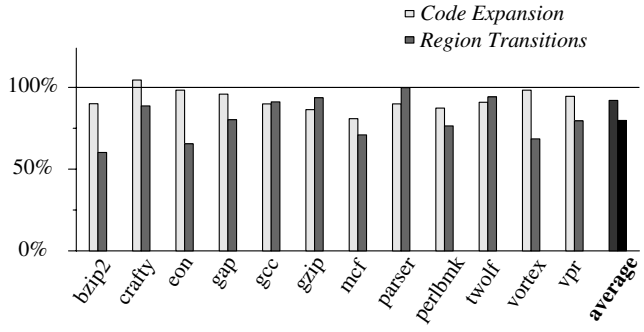

Figure 8. *Code expansion and region transitions of LEI relative to NET.*

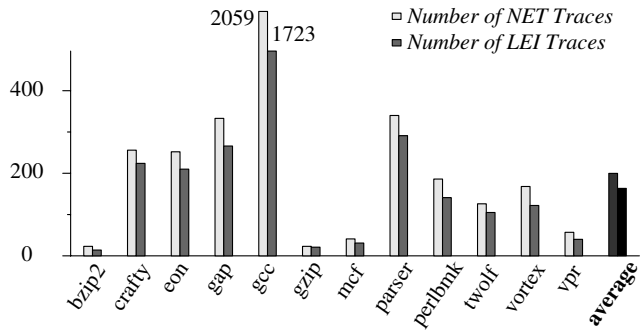

Figure 9. *Minimum number of traces required to cover 90% of the instructions executed by each benchmark.*

tions is only 80% of that of NET. In this way, including backward branches but avoiding cycle duplication enables LEI to simultaneously reduce the problems of separation and duplication.

Two results from Figure 8 stand out: code expansion for `crafty` and the number of region transitions for `parser`. In both cases, LEI performs no better than NET. By considering Figure 7, the reason for this becomes clear: these are the two benchmarks for which LEI spans the fewest additional cycles. This correlation holds for other benchmarks as well, as the more additional cycles LEI spans the more it outperforms NET.

3.2.3 90% Cover Set Size

The size of the 90% cover set for each benchmark is given in Figure 9. In all cases, LEI requires a significantly smaller set of traces, with an average reduction of 18%. As execution is concentrated in a smaller number of larger regions, locality of execution improves and the opportunity for optimization within each trace increases [4]. Given the empirical association between the size of a 90% cover set and the runtime performance of trace-based dynamic optimization, this provides strong evidence that LEI would be more effective in practice than NET.

3.2.4 Profiling Overhead

Finally, we consider the memory overhead that LEI requires for profiling. The history buffer itself requires additional memory, but its size is fixed and very small relative to overall memory overhead. Counter overhead is an important issue for profiling techniques, and one of NET's strong points is that it only requires a counter for a subset of branch targets [10]. Moreover, once a counter reaches the threshold value it can be reused for another branch target. LEI maintains both of these properties and requires less counter memory because it is more restrictive about whether to

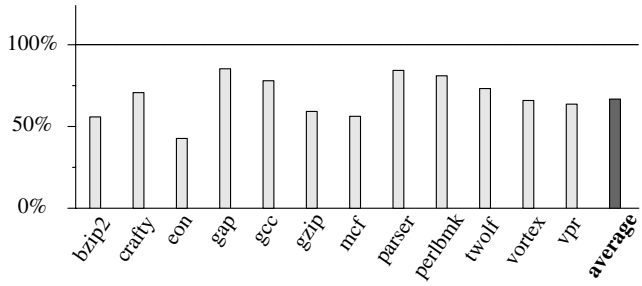

Figure 10. *Number of counters required by LEI relative to NET.*

associate a counter with a branch target. Not only must an address be the target of a backward branch or follow an exit from the code cache, it must also be in the history buffer of recently interpreted branch targets. Figure 10 gives the relative values of the maximum number of counters in use at any point in the benchmark, indicating that LEI requires only two-thirds the profiling memory of NET. Therefore, by doing slightly more analysis, LEI is able to select more effective traces while requiring less memory for profiling.

4 Trace Combination

In this section, we address the problems inherent in trace selection by developing an algorithm that selects regions containing multiple interprocedural paths. The difficulty lies in deciding which paths to include and which to exclude while not incurring significant profiling overhead. We observe that a trace-selection algorithm efficiently selects a single path, so we develop an algorithm that combines multiple traces and attempts to minimize the overhead of combination. It does not depend on how traces are selected, so we evaluate its performance using both Next-Executing Tail (NET) traces and our Last-Executed Iteration (LEI) traces. We find that it reduces the problem of trace separation significantly while resulting in a much smaller code cache.

4.1 Exit Domination

To show that NET and LEI suffer from separation and excessive code duplication because they select individual traces, we measure the amount of *exit domination* they produce. As we will see, when one region exit-dominates another, there is no benefit to selecting each independently. Instead, combining the two regions reduces separation. Moreover, if they contain any of the same blocks—a situation we call *exit-dominated duplication*—combining the two regions also reduces code duplication.

We say that region R exit-dominates region S if three conditions hold. First, S begins at an exit from R. Second, the exit block is the only predecessor to the entrance block of S that executes[5] and is not contained in S. Third, R was selected before S. Together, these imply that there was no benefit to stopping R at the exit to S: it served only to delay the selection of S, separate the regions in the code cache and during optimization, and potentially introduce additional duplication.

Figures 11 and 12 show the amount of exit domination between traces selected. In Figure 11, we see that exit-dominated traces often contain duplicate instructions, ranging from 1 to 7% of all instructions selected. More significantly, Figure 12 shows that a

[5]The traditional definition of domination considers every edge regardless of whether it executes. However, we use exit-domination to detect when separating regions is not useful, and an incoming edge that is never executed does not make separation useful.

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

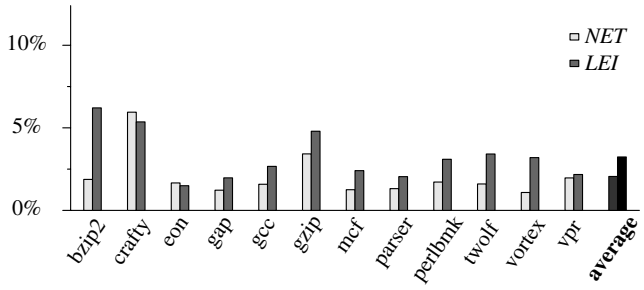

Figure 11. *The proportion of instructions selected by NET and LEI that are exit-dominated duplication.*

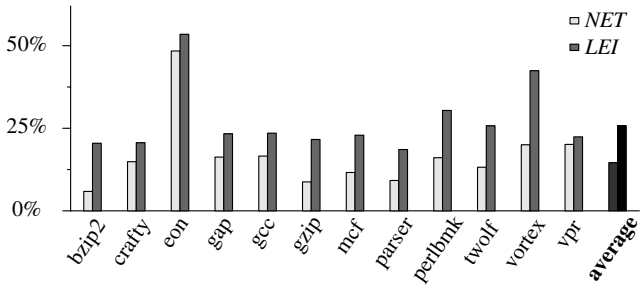

Figure 12. *The proportion of traces selected by NET and LEI that are exit-dominated.*

high percentage of traces are exit-dominated: on average, 15% of NET traces and 22% of LEI traces. This unnecessary separation—where the dominated trace can only be entered by exiting the dominating trace—suggests that if a larger region were selected that contained both traces, locality of execution would improve.

For nearly all benchmarks, exit-dominated traces comprise between 10 and 25% of selected traces. However, `eon` is a clear outlier. The difference is that `eon` produces several traces that each exit-dominates a large number of other traces. For example, three of these exit-dominating traces correspond to constructors of the widely used `ggPoint3` class. Once a trace is selected for such a constructor, an exit-dominated trace will be selected for each frequently executed function that calls it. If these traces are not considered, `eon`'s proportion of exit-dominated traces decreases to just above average.

In almost all cases, LEI produces more exit-dominated traces than NET, and this causes more exit-dominated duplication. This does not imply that it is a less effective trace-selection algorithm than NET; Section 3 demonstrated the opposite. Rather, it shows that the same opportunities exist under LEI and NET, despite LEI emitting fewer traces and less code. As LEI has proportionally more exit domination, trace combination should be especially beneficial for it. As we will see in Section 4.3, this is very much the case.

4.2 Trace-Combination Algorithm

Trace combination is a simple extension of trace selection. A trace-selection algorithm profiles certain branch targets until one executes a especific number of times, and then it selects a single trace. Trace combination lowers this threshold and observes the traces generated for the next several executions of the target. It then selects a region that begins at the branch target and combines blocks from these several *observed traces*.

By doing so, trace combination allows a region to contain multiple paths without requiring it to. If there is a single dominant path from a branch target, trace combination selects only that path, because all observed traces contain the same blocks. Only when multiple paths execute frequently does it combine them.

```
INTERPRETED-BRANCH-TAKEN( address src , address dest )
 1   address cached ← HASH-LOOKUP( code cache , dest)
 2   if cached
 3     then jump cached
 4
 5   if dest is a potential trace entrance
 6     then increment counter c associated with dest
 7          if c > T_start
 8            then form a trace t beginning at dest
 9                 store COMPACT-TRACE(t) in memory
10          if c = T_start + T_prof
11            then recycle counter c
12                 CFG G ← combine observed traces at dest
13                 mark all blocks that appear in at least T_min traces
14                 MARK-REJOINING-PATHS(G)
15                 remove from G all unmarked blocks
16                 replace any region exit that targets a block in G
17                 address newR ← INSERT-OPTIMIZED-REGION(G)
18                 jump newR
```

Figure 13. *Algorithm for trace combination.*

The simplest form of trace combination would select all blocks from each observed trace. This increases the chance that control remains within the optimized region, but it also increases the number of infrequently executed blocks that are selected. As we argued when discussing code expansion, this has a high cost in a dynamic system. To prevent this, our algorithm constructs a region with a two-step process: first it includes only those blocks that occur in frequently executed traces and then it includes executed paths that rejoin those blocks.[6]

Figure 13 provides an overview of the algorithm and shows how it selects blocks that frequently occur in observed traces. For each of T_{prof} executions from a branch target, line 8 selects a trace and line 9 stores it to memory. On the last execution, line 12 combines all of these traces to form a CFG representation of the paths, which will be used not only to select the region but also to optimize it. As it builds the CFG, the algorithm labels each block with the number of observed traces in which it occurs, marking all blocks that occur at least T_{min} times. Line 14 then runs an algorithm to mark all blocks in the CFG on a path to one of these frequently occurring blocks. All unmarked blocks are removed on line 15, and all unnecessary exit stubs are removed on line 16. Line 17 optimizes the resulting region and inserts it into the code cache.

Three aspects of this algorithm require more detail: how observed traces are stored, how they are combined into a CFG, and how rejoining paths are marked.

4.2.1 Storing a Trace

In order to delay all analysis until a region is selected, we store each observed trace independently. That is, we do not look for blocks in common between observed traces when storing them, because the counter associated with their entrance may never reach the threshold. Although this avoids unnecessary analysis, it could require significantly more memory, as many copies of the same trace may be stored.

To reduce memory overhead, we store a compact representation of each trace. Specifically, we record the outcome of each branch encountered in the trace so that it can be reconstructed if a region is selected. As the optimizer must already decode each instruction

[6]Executed paths that rejoin frequently executed blocks are included because, if selected separately, they are the cause of exit dominated duplication.

```
COMPACT-TRACE( trace t )
 1  bitstring b
 2  for each branch in t
 3  do if branch is an indirect branch
 4        then append "01" to b followed by the target address
 5
 6        else if branch is conditional
 7                then if branch is not taken
 8                        then append "10" to b
 9
10                else if branch is taken
11                        then append "11" to b
12   append "00" to b
13   append the address of the last instruction in t to b
14   return b
```

Figure 14. *Algorithm for representing a trace compactly.*

and identify all branch targets, this representation adds little overhead to region selection, and it leads to a simple CFG construction algorithm that decodes each instruction at most once.

Each branch in a trace falls into one of three categories: it is not taken; it is taken and the target is known from the instruction; or it is taken and the target is not known from the instruction. With only three possibilities, representing a trace as a sequence of branches requires only two bits for most branches. For each taken branch with an indirect target, the representation must explicitly include the target address, requiring an additional 32 or 64 bits. As a trace can end at any instruction, the address of the end of the trace is appended to its representation.[7] Figure 14 gives the algorithm for constructing this compact representation.

4.2.2 Constructing the CFG

Constructing a control-flow graph for the observed traces is simpler than constructing a control-flow graph in general. Rather than representing all possible branches, the CFG for a region represents only those branches taken in an observed trace. This simplifies construction, as all branch targets taken in an observed trace are known. Although the resulting CFG only represents the observed traces, it is sufficient because control exits the region if any other target is taken.

Our algorithm constructs a CFG by incrementally adding each observed trace. As all traces start at the same address, adding a trace is done by starting at the entrance of the current CFG and traversing its path of taken and not-taken branches. If the trace contains a control transfer that does not correspond to an edge in the current CFG, we ensure that the target address begins a block in the trace and add the edge. As the CFG is constructed, each block is annotated with the number of observed traces in which it appears, and all that appear in T_{min} traces are marked.

4.2.3 Marking Paths that Rejoin

Given a CFG with marked blocks, we mark all paths that exit and rejoin these blocks. As marked blocks correspond to those that will be selected, this expands the original region to include any observed block on a rejoining path.

All observed traces begin at the first block in the CFG, so we are guaranteed that its frequency is T_{prof} and therefore that it is marked. As each block in our CFG is reachable from the first, each block is on a path that exits a marked block. The problem, then, reduces to deciding whether each block is on a path that rejoins a marked block—that is, whether a marked block is reachable from it.

[7] This is true even of NET, which not only ends a trace at a backward taken branch but also when it contains a maximum number of instructions.

```
MARK-REJOINING-PATHS( marked CFG G )
1  repeat
2        changed ← false
3        for each unmarked block b in G in post-order
4        do for each successor s of b
5           do if s is marked
6                 then mark b
7                      changed ← true
8
9    until changed = false
```

Figure 15. *Algorithm for marking rejoining paths.*

Deciding this for each block can be done with a simple version of an iterative data flow algorithm. Blocks are initially marked or unmarked based on how many observed traces contain them. Marks are propagated backward along any path: if any successor of a block is marked, the block is marked. The algorithm terminates when all blocks are considered without marking any.

The full algorithm is given in Figure 15. It is correct because it only terminates when no unmarked block has a marked successor, and therefore no path exists from an unmarked block to a marked block. It terminates because each iteration marks at least one block, and no block's mark is is ever erased. Therefore, for a CFG with n blocks and e edges, there are at most $O(n)$ iterations.

As each iteration considers every edge, the algorithm has a worst-case complexity of $O(ne)$. However, in practice it is almost always linear in the number of edges. Since blocks are considered in a post-order traversal of the CFG, successors are visited before predecessors and marks can propagate through multiple blocks in the same iteration. Successors cannot always be visited before predecessors because of the presence of back edges, but this rarely causes an additional iteration. For our benchmarks, roughly 0.1% of regions that mark blocks in the first iteration proceed to mark additional blocks in the second.

Once rejoining paths have been marked, all unmarked blocks are removed from the CFG. The resulting region may contain exits that target blocks in the region. As our algorithm selects regions with split and join points, each such exit can be replaced with an edge. Doing so keeps control in the region longer and reduces the number of exit stubs required.

4.3 Experimental Results

To evaluate the performance of trace combination, we use our simulation framework to measure its effect on NET and LEI. To isolate its effect, we consider each trace-selection algorithm separately, presenting the performance of *combined NET* relative to NET and the performance of *combined LEI* relative to LEI. By doing so, we come to two conclusions: that trace combination improves each trace-selection algorithm significantly, and that LEI traces are especially well-suited to combination.

We first measure the effect of combining traces on reducing exit domination. We find that in all cases both the number of exit-dominated regions and the amount of exit-dominated duplication decreases significantly. We then present results for the metrics developed in Section 2.3 and determine that trace combination reduces both separation and duplication and is likely to improve overall performance. A main cost of trace combination is the memory it requires to store observed traces, but we find that on average this increase is offset by the reduction in the size of the code cache, so the system's total memory overhead is similar.

For these experiments, we select thresholds that allow direct comparison with the underlying trace-selection algorithm. Specifically, we ensure that regions are selected after the same number of interpreted executions. As we use $T_{prof} = 15$ and $T_{min} = 5$, this means that combined NET begins profiling after 35 execu-

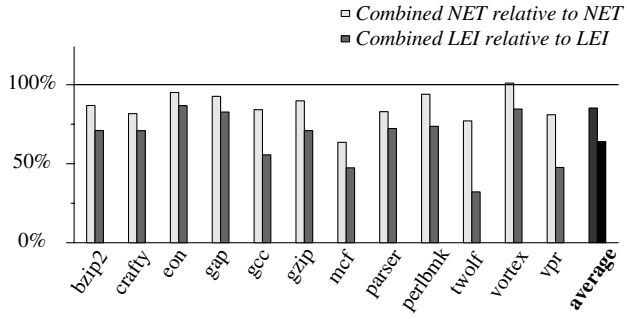

Figure 16. *Reduction in the number of region transitions under trace combination.*

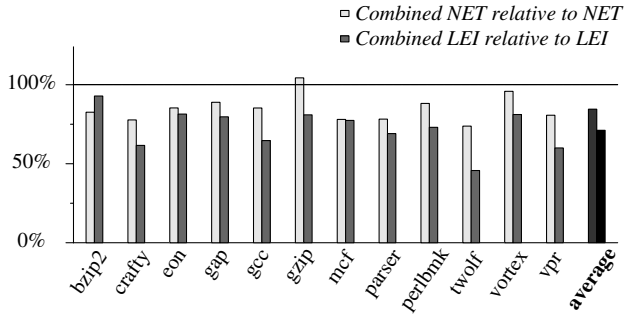

Figure 17. *Reduction in the 90% cover set size under trace combination.*

tions rather than 50, and combined LEI begins after 20 rather than 35. Profiling 15 executions strikes a balance between gathering more information for selection and optimization and incurring more overhead to combine traces. Trace combination remains effective with many fewer executions, so a different balance could be struck if this overhead is significant in practice.[8]

As with LEI, our trace-combination algorithm has a very small effect on hit rate so we discuss the results only briefly. For combined NET, hit rate increases very slightly for all benchmarks. For combined LEI, hit rate decreases more significantly, but only by an average of 0.1%. Moreover, it remains above 98% for all benchmarks and above 99% for all except `gcc`.

4.3.1 Reducing Exit Domination

As expected, trace combination reduces the amount of exit domination for all benchmarks. On average, combining traces avoids roughly 65% of exit-dominated duplication. In addition, the number of exit-dominated regions, which in Section 4.1 we found to be very common, decreases by 40%.

These results highlight two important properties of our trace-combination algorithm. First, it does not avoid all exit-domination, as regions are selected from blocks that occur in a sample of only T_{prof} traces. Not only does this limit the number of paths that can be incorporated into a region, but it also relies on current execution being representative of future execution. This is often not the case, as programs have been shown to execute different paths in different *phases* of execution [18]. Second, it reduces exit-dominated duplication more than it reduces the number of exit-dominated regions. As exit-dominated duplication is caused by an exit-dominated path that rejoins the original region, this indicates the success of incorporating all observed rejoining paths.

4.3.2 Code Expansion and Locality of Execution

Figure 16 shows the effectiveness of trace combination in reducing the problem of separation. Trace combination replaces many region transitions with local branches where the branch instruction and target are optimized together. When NET traces are combined, there are on average 85% as many region transitions. When LEI traces are combined, there are only 64% as many. Both are able to incorporate paths that would have been exit-dominated, and as expected the reduction in region transitions is correlated with the reduction in exit-dominated regions. Combining LEI traces improves locality of execution further by incorporating more cycles into a region, reducing the number of region transitions as control remains in the same region longer.

[8]For example, setting $T_{prof} = 5$ and $T_{min} = 2$ results in smaller but similar improvements.

For `vortex` under NET, the number of region transitions rose roughly 1%. It is possible for this to occur because trace combination requires that each block be observed in T_{min} traces. This can cause selected paths to include only some of the blocks from each trace, and the resulting shorter paths will require more region transitions. This matches our observation that the average region size increases far less in `vortex` than in other benchmarks. However, Figure 16 shows the overall positive effect of trace combination on locality of execution.

In addition, trace combination achieves this improvement without increasing code expansion. Specifically, combined NET selects 98% as many instructions as NET and combined LEI selects 99% as many as LEI. Besides reducing exit-dominated duplication, trace combination has two effects on the amount of code expansion. First, it may select infrequently executed blocks that would not have been selected otherwise. This is because the threshold for including a block in a region is much lower than the threshold for beginning a region with a block. Second, it may avoid selecting infrequently executed blocks that would have been selected otherwise. This is because it verifies each selected block by requiring that it appear in at least T_{min} observed traces or be on a rejoining path. In our experiments, the second effect slightly outweighs the first, as trace combination reduces code expansion by more than it reduces exit-dominated duplication.

4.3.3 90% Cover Set Size

Figure 17 shows that the size of the 90% cover set for each benchmark decreases substantially when traces are combined. Trace combination reduces the average size of NET cover sets by 15% and LEI cover sets by 28%. There is only one case in which the size of the cover set increases: for `gzip` with NET traces, it rises trivially from 23 to 24 traces. Similarly, `bzip2` is the only case in which trace combination improves LEI less than NET. This is because `bzip2` has a much smaller cover set with LEI than with NET, so additional reductions are more difficult to achieve. Overall, we see a consistent reduction in cover set size, and we find that, just as LEI produces more exit domination, it can benefit more from trace combination.

Not only does trace combination reduce the size of the 90% cover set for each benchmark, it also reduces the total number of regions selected. For NET, the average reduction is 9%, while for LEI it is 30%. This allows more time to be spent optimizing each region, which helps to compensate for the fact that optimization will take longer than on traces.

4.3.4 Profiling Overhead

Figure 18 shows the memory overhead of trace combination, computed as the maximum amount of memory needed at any program

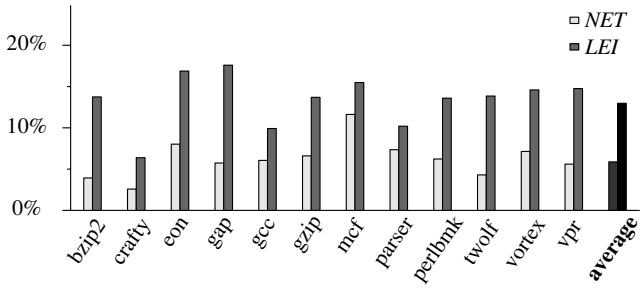

Figure 18. *Maximum amount of memory overhead required for storing observed traces, computed as a percentage of the estimated size of the cache.*

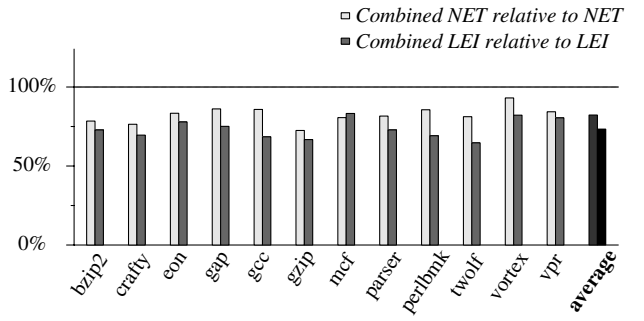

Figure 19. *Effect of trace combination on the number of exit stubs produced by NET and LEI.*

point to store observed traces. To allow comparison across benchmarks, we report each as a percentage of the estimated size of the code cache. To estimate its size, we compute the total number of instruction bytes inserted in the code cache and conservatively add 10 bytes for each exit stub. In the DynamoRIO system, each exit stub requires a minimum of three instructions [14] and for all benchmarks the average size of a selected instruction is between three and four bytes. We do not attempt to estimate the effect of optimization on region size, and we ignore the memory required for links between regions in the cache.[9]

With these conservative assumptions, the average memory overhead for trace combination is 6% for NET and 13% for LEI. Trace combination never requires more than a 12% overhead with NET or more than an 18% overhead with LEI. More memory is consistently required for LEI because it produces longer traces, and the requirement that a branch target be in the history buffer causes more delay in identifying each subsequent trace. This delay means that traces are observed longer for each branch target, which increases the number of branch targets that are observed concurrently.

This memory overhead is significant, but trace combination compensates for it by reducing the size of the code cache. As shown in Figure 19, trace combination reduces the number of exit stubs significantly: 18% fewer are required for NET and 26% fewer are required for LEI. Together with selecting fewer instructions, on average this effect reduces the size of the cache by 7% for NET and 9% for LEI. In terms of performance, reducing the size of the code cache is more important than reducing profiling memory, as exit stubs and duplication in the code cache reduce locality of execution.

[9] Our algorithms are very likely to reduce the number of such links, as fewer regions are selected and each contains more related code.

4.4 Effect on Optimization

Incorporating multiple paths into a region is likely to affect how it is optimized. In particular, the optimizer must perform more analysis to determine how a block interacts with its predecessors and successors. Therefore, aggressively optimizing a region with multiple paths will in general take longer than performing the same optimizations on a trace. However, three factors suggest that optimizations will be more effective on regions that contain multiple paths.

First, the most beneficial dynamic optimization is code layout. In Dynamo, roughly two-thirds of the average performance speedup results from trace selection rather than other optimizations [3]. That is, removing unconditional jumps and forming traces that cross procedure boundaries are far more important than Dynamo's other optimizations. Regions containing multiple paths also exhibit these benefits, and they improve locality of execution further by replacing distant inter-region jumps with local intra-region jumps.

Second, regions with multiple paths give the optimizer more information about the context of the selected paths. When a region contains both sides of an `if-else` statement, redundancy elimination does not need to produce compensation code. When a region contains a cycle, loop optimizations can be performed that are not possible when the cycle is separated over multiple traces. Loop-invariant code motion is an especially important example, which moves code from within a loop to above the loop when possible. Opportunities for such code motion often increase in dynamically selected regions, because an instruction may be invariant in a selected cycle but not in the entire original loop. However, even a trace that spans a cycle cannot perform this optimization, because it has nowhere outside the cycle to move an instruction.

Third, an algorithm for selecting a larger region can gather information about execution patterns within it (e.g., edge and path counts). This allows the optimizer to perform code layout and other optimizations that make a frequently executed path more efficient at the expense of an infrequently executed path. For example, Young and Smith [20] use path counts to guide a global instruction scheduler.

5 Related Work

In addition to NET, several other trace-selection algorithms have been used in dynamic optimization systems. They differ from NET in two main ways. Those that are software-based use more profiling to identify the behavior of each potential branch in a trace. Those that are hardware-based observe native execution and often use random sampling to select a trace. The problems of separation and duplication apply as much to these trace-selection algorithms as to NET.

Mojo is a transparent optimization system for Windows that is very similar to Dynamo. One main difference is that it uses one threshold for backward-branch targets and a lower threshold for trace exits. The authors claim that this lower threshold reduces the impact of the rare case where the next-executing trace is a cold path. In terms of our analysis, having a lower threshold for exit targets also reduces the separation between related hot traces. However, this approach still does not allow the related traces to be optimized together.

BOA is a binary translation system developed at IBM that translates PowerPC instructions into native instructions for a proprietary, high-frequency processor [17]. In its emulation phase, BOA maintains counts for each conditional branch that indicate how many times each target is taken. After the entry point to an instruction sequence is emulated 15 times, a trace is selected by following the target of each conditional branch with the highest count.

Wiggins/Redstone is a transparent optimization system developed at Compaq that uses a combination of hardware sampling and software instrumentation [8]. To identify the beginning of a trace, the program counter is periodically sampled. From a starting instruction, a trace is selected by adding instrumentation code that determines the most frequent target of each selected branch.

ADORE is a transparent optimization system developed at the University of Minnesota that uses performance counters built into the target processor [15]. Specifically, it samples registers from the *performance monitoring unit* of the Intel Itanium 2 in order to detect the four most recently taken branches. When a set of four branches occurs frequently, the corresponding path is selected and linked with other frequent paths to form a trace. Besides being hardware-based and processor-specific, the main difference between this algorithm and others discussed is that frequent branch targets are identified by random sampling.

All three techniques profile more branches in the hope of better identifying a hot trace. Unfortunately, careful selection of traces does not address the problems of separation and duplication. The detailed profile information gathered by these techniques can, however, reduce the cost of trace combination, because this information can help select which blocks to combine.

6 Conclusion

We have identified and quantified problems of trace separation and excessive code duplication in the popular Next-Executing Tail (NET) trace-selection algorithm. These problems negatively impact application performance, especially in large applications with many important procedures and a mix of biased and unbiased branches (e.g., 176.gcc). Motivated by these problems, we developed two new region-selection algorithms that often select a smaller number of larger regions than NET. By doing so, our algorithms better select regions of frequently executing code and directly improve locality of execution. In addition, each algorithm is designed to decrease a specific type of needless duplication.

The success of our algorithms is best summarized by comparing their combined performance to that of NET. By performing slightly more analysis, our algorithms reduce code expansion by 9% and the number of exit stubs by 32% while simultaneously cutting the number of region transitions in half. Our best measure of performance, the 90% cover set size, improves by more than 25% for every benchmark, averaging a 44% improvement. The small costs of our additional analysis are more than covered by the savings produced.

Given this success, we are currently working with Intel to modify Pin so that it can accept a user-specified trace-selection algorithm. The early results are positive—they demonstrate that our trace-selection algorithms improve overall performance. We are also beginning to investigate dynamic optimizations that benefit from the larger regions produced by our algorithms.

7 Acknowledgments

This work has been funded in part by NSF grants CCR-0310877 and CCR-0429782, and by gifts from Microsoft. The work was done while David Hiniker was an undergraduate at Harvard University and Kim Hazelwood was employed by Intel.

References

[1] B. Alpern, et al. "The Jikes Research Virtual Machine project: Buliding an open-source research community," IBM Systems Journal, vol. 44, no. 2, 2005.

[2] M. Arnold, et al. "Adaptive optimization in the Jalapeño JVM," Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000), October 2000.

[3] V. Bala, E. Duesterwald, and S. Banerjia. "Dynamo: A Transparent Runtime Optimization System," Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pp. 1–12, June 2000.

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

[5] T. Ball and J. Larus. "Programs Follow Paths," Microsoft Research Technical Report MSR-TR-99-01, January 1999.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. "An Infrastructure for Adaptive Dynamic Optimization," Proc. First Annual International Symposium on Code Generation and Optimization, pp. 265–275, March 2003.

[7] W. Chen et al. "Mojo: A Dynamic Optimization System," Proc. 4th ACM Workshop on Feedback-Directed and Dynamic Optimization, pp. 81–90, December 2000.

[8] D. Deaver, R. Gorton, and N. Rubin. "Wiggins/Redstone: An Online Program Specializer," Proc. IEEE Hot Chips XI, 1999.

[9] J. Dehnert et al. "The Transmeta Code Morphing(tm) Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges," Proc. 2003 International Symposium on Code Generation and Optimization, pp. 15–24, March 2003.

[10] E. Duesterwald and V. Bala. "Software Profiling for Hot Path Prediction: Less is More," Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 202–211, November 2000.

[11] P. Chang, S. Mahlke, and W. Hwu. "Using Profile Information to Assist Classic Code Optimizations," Software Practice and Experience, vol. 21, pp. 1301–1321, December 1991.

[12] M. Gschwind et al. "Dynamic and Transparent Binary Translation," IEEE Computer Magazine, vol. 33, pp. 54–59, March 2000.

[13] R. Hank, W. Hwu, and B. Rau. "Region-Based Compilation: An Introduction and Motivation," *Proc. 28th Annual International Symposium on Microarchitecture*, pp. 158–168, November 1995.

[14] K. Hazelwood. "Code Cache Management in Dynamic Optimization Systems," PhD thesis, Harvard University, 2004.

[15] J. Lu et al. "Design and Implementation of a Lightweight Dynamic Optimization System," Journal of Instruction-Level Parallelism, vol. 6, pp. 1–24, April 2004.

[16] C. Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," Proc. ACM Programming Language Design and Implementation, pp. 190–200, June 2005.

[17] S. Sathaye et al. "BOA: Targeting Multi-gigahertz with Binary Translation," Proc. of the 1999 Workshop on Binary Translation, pp. 2–11, December 1999.

[18] T. Sherwood et al. "Automatically Characterizing Large Scale Program Behavior," Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002.

[19] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation. http://www.spec.org/osg/cpu2000/.

[20] C. Young and M. Smith. "Better Global Scheduling Using Path Profiles," Proc. 31st Annual ACM/IEEE International Symposium on Microarchitecture, pp. 115–123, November 1998.