Improving Region Selection
in Dynamic Optimization Systems

A thesis presented
by
David J. Hiniker
to
Computer Science
in partial fulfillment of the honors requirements
for the degree of
Bachelor of Arts
Harvard College
Cambridge, Massachusetts

April 5, 2005
Abstract

The performance of a dynamic optimization system depends heavily on the code it selects to optimize. Ideally, nearly all execution takes place in the selected code, so its optimization and layout determine the performance of the system. Yet despite its importance, relatively little work has been done to improve how code is selected.

We define a region to be the unit of code that a dynamic optimization system selects for optimization. We consider the most widely-used region, a trace, and show that current trace-selection algorithms are not designed to handle the continuing growth in application size. To address this problem, we develop two new region-selection algorithms.

We first develop a trace-selection algorithm that identifies cyclic paths of execution. This allows it to select many more traces that correspond to loops, and we find that this improves locality of execution while reducing the amount of code selected. Its overhead is comparable to current algorithms, and we show that it is likely to result in significantly better performance.

We then develop an algorithm that selects a larger region by combining multiple traces. It can be applied to any trace-selection algorithm, and we measure its performance when applied both to a widely-used trace-selection algorithm and to ours. We find that it significantly improves both, and that the combined performance of our trace-selection and trace-combination algorithms is especially good. Together, they select less code and substantially improve locality of execution while averaging a 44% improvement in our best measure of performance.
Acknowledgments

First, I want to thank my advisor, Mike Smith, for all of his help and guidance. He introduced me to this topic and always pointed me in the right direction. But more than that, he was always willing to talk, and his encouragement made all the difference.

Along with Mike, thanks to Brian, Glenn, Kim, and Professor Ramsey and his writing group for providing helpful advice on the content and organization of this thesis. It is certainly better for it. I owe a special thanks to Glenn for his feedback, for setting up the systems I used, and for helping out whenever I ran into trouble.

I also want to thank my parents for always being interested in my work. It means a lot to me.

Finally, I want to thank Alexis. Not only for being my editor and my collaborator, but for being my motivation and my happiness.
# Contents

Abstract ................................................................. i  
Acknowledgments ..................................................... ii  
List of Figures ........................................................ iii  

1 Introduction .......................................................... 1  
  1.1 Regions .......................................................... 2  
  1.2 Problems with Existing Trace Selection ....................... 3  
  1.3 Selecting Larger Regions ...................................... 4  

2 Background and Related Work ..................................... 6  
  2.1 Terminology ...................................................... 6  
  2.2 Dynamic Optimization Setting ................................. 8  
  2.3 Next-Executing Tail Trace Selection .......................... 10  
  2.4 Related Trace-Selection Algorithms ............................ 12  

3 Properties of Effective Regions ................................ 14  
  3.1 Spanning a Cycle ............................................... 15  
  3.2 Combining Multiple Paths .................................... 18  
  3.3 Measuring Overall Performance ............................... 22  
  3.4 Methodology ................................................... 23  

4 Last-Executed Iteration Trace Selection ......................... 25  
  4.1 Analysis of Next-Executing Tail ............................... 25  
  4.2 Last-Executed Iteration Algorithm ........................... 27  
  4.3 Experimental Results ......................................... 31  
  4.4 Summary ....................................................... 35  

5 Trace Combination .................................................. 36  
  5.1 Analysis of Trace-Selection Algorithms ....................... 37  
  5.2 Trace Combination Algorithm .................................. 38  
  5.3 Experimental Results ......................................... 46  
  5.4 Summary ....................................................... 52  

6 Conclusion ........................................................... 53  
  6.1 Future Work .................................................... 54  

A Additional Data .................................................... 55  

Bibliography ........................................................... 57
# List of Figures

1.1 Overview of a dynamic optimization system ........................................... 2
2.1 Simple pseudocode and the corresponding control-flow graph .................. 7
2.2 Overview of execution under a dynamic optimization system .................. 9
2.3 Example traces formed by Next-Executing Tail ...................................... 12
3.1 Traces selected for simple nested loops ............................................. 16
3.2 Traces selected by NET for an unbiased branch .................................... 18
3.3 Region with multiple paths selected for an unbiased branch .................... 19
4.1 Traces selected by NET for a loop with a function call ........................... 26
4.2 Importance of spanning interprocedural cycles .................................... 27
4.3 Algorithm for Last-Executed Iteration trace selection ............................ 29
4.4 Algorithm for forming an LEI trace from a history buffer ....................... 29
4.5 Example history buffer for LEI .................................................... 30
4.6 Effect of LEI on spanning cycles .................................................. 32
4.7 Effect of LEI on 90% cover sets .................................................... 33
4.8 Effect of LEI on code expansion and region transitions .......................... 34
4.9 Memory overhead of LEI ............................................................ 35
5.1 Exit-dominated traces in NET and LEI ............................................. 37
5.2 Exit-dominated duplication in NET and LEI ..................................... 38
5.3 Algorithm for trace combination ..................................................... 39
5.4 Control-flow graph and memory layout of the extended example ............... 40
5.5 Algorithm for representing a trace compactly ..................................... 41
5.6 The control-flow graph built for trace combination ............................... 43
5.7 Algorithm for marking rejoining paths ............................................ 44
5.8 The selected region and a possible memory layout ............................... 45
5.9 Effect of trace combination on exit-dominated duplication ...................... 47
5.10 Effect of trace combination on 90% cover sets .................................. 48
5.11 Effect of trace combination on region transitions ................................ 49
5.12 Memory overhead of trace combination .......................................... 51
5.13 Effect of trace combination on exit stubs ........................................ 52
A.1 Effect of trace combination on exit-dominated regions .......................... 56
A.2 Effect of trace combination on code expansion .................................... 56
Chapter 1

Introduction

Traditionally, a program is compiled and optimized before it is executed. Optimizing a program while it executes, however, can improve its performance by using information not available until run-time. Specifically, a static compiler can only estimate the behavior of a program on an input, must make conservative assumptions about the target processor, and cannot analyze libraries loaded at run-time. In contrast, a dynamic optimization system observes execution patterns given the current input on the current processor and can optimize any code that executes. As processor designs evolve and more software is written as a collection of run-time libraries, dynamic optimization is becoming increasingly important.

A dynamic optimization system fits into one of three general categories: transparent optimization, just-in-time compilation, or binary translation. A transparent optimizer takes a binary executable for the 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 the target processor. In each case, optimization is central to the translation and is performed while the program executes.

The main disadvantage of dynamic optimization is the additional run-time overhead it requires, both the time for optimization and the space for optimized code. In order to reduce this cost, optimizations are applied selectively to portions of the code that are expected
to execute frequently. Figure 1.1 provides an overview of how a dynamic optimization system accomplishes this. Code from the original binary is initially emulated, and profiling information about its execution patterns is gathered. When a section of code begins to execute frequently, part of it is selected and optimized. The optimized code is placed in a code cache and all subsequent executions occur directly on the target processor. As the cost of optimization is high but execution from the code cache is fast, deciding what code to optimize is central to the performance of a dynamic optimization system.

1.1 Regions

We define a region to be the unit of code that a dynamic optimization system selects for optimization. The two regions commonly selected by existing systems are a whole method and a trace. Like traditional static compilers, just-in-time compilers are often method-based. However, since a main goal of dynamic optimization is to improve code across static program boundaries—such as procedure and library boundaries—an increasing number of dynamic optimization systems are trace-based.

A trace is a single interprocedural path of execution.\(^1\) A trace is formed from a sequence of executed instructions, so it is not limited by any static program boundary. Despite this advantage, selecting a trace in isolation from other paths of code leads to several inefficiencies. This thesis focuses on identifying problems with existing trace-selection algorithms and developing region-selection algorithms that address them.

\(^1\)In the terminology of static instruction scheduling, a trace is an interprocedural superblock [Chang et al., 1991]. Like a superblock, a dynamic optimization trace has a single entry-point.
1.2 Problems with Existing Trace Selection

We focus on two main problems caused by existing trace-selection algorithms: the problem of separation and the problem of duplication.

The problem of separation is that related paths of code in the original program are selected as separate traces and often placed far apart in the code cache. This occurs because once a trace has been selected it takes many executions for a related trace to be selected. During this time, traces from other parts of the code are likely to be selected and promoted to the code cache. Once the 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. For a widely-used trace-selection algorithm, the source and destination of an average jump within the code cache are separated by 147 other traces.

The problem of duplication is 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 a static compiler uses duplication in method inlining to improve performance. However, we identify types of trace duplication that yield little benefit and argue that the costs of duplication are very high in a dynamic system. Duplication degrades performance because it increases the amount of code optimized at run-time and the impact of the code cache on the memory system. For the same widely-used trace-selection algorithm, on average 31% of the instructions selected are duplication.

These problems are becoming especially significant because of a trend in software toward executables with more frequently-executing paths. As shown by Ball and Larus (1990), the number of paths that comprise 90% of execution in modern commercial software is often one to two orders of magnitude higher than in the standard benchmark programs used

---

2 Next-Executing Tail trace selection [Duesterwald and Bala, 2000], which we later use to evaluate the performance of our algorithms.

3 This and the next statistic were obtained using the simulation framework and benchmarks described in Section 3.4.
1.3 Selecting Larger Regions

when trace selection was developed. As the number of related paths grows, the extent of separation and the amount of duplication grow with it.

### 1.3 Selecting Larger Regions

This thesis develops two region-selection algorithms to reduce these problems. Each of our algorithms selects regions that include more code than standard traces and result in less duplication. By doing so, they select a smaller number of larger regions and reduce the overall size of the code cache. By combining more related code, these algorithms reduce the problem of separation and improve locality of execution. Chapter 3 develops the properties of effective regions that motivate our algorithms, and it describes the performance metrics we use to evaluate them.

In Chapter 4, we develop the *Last-Executed Iteration* (LEI) algorithm that selects traces based on cyclic paths.\(^4\) As we will discuss, current trace-selection algorithms do not allow a trace to cross certain types of branches. Our traces can include arbitrary branches, which reduces the chance that frequently-executed paths are separated into multiple traces.

In Chapter 5, we develop the *trace combination* algorithm to select a region that includes multiple paths. Like a trace, such a region is interprocedural and does not include all of a method. Trace combination observes multiple traces and combines those that execute frequently, expanding the resulting region to include any executed path that exits and rejoins. By including multiple paths in a region, it ensures that none are separated and no code is duplicated among them.

Trace combination can observe and combine any type of trace, which allows us to evaluate the performance of our algorithms separately. In Chapter 4, we compare performance metrics for LEI traces against those for a standard trace-selection algorithm. In Chapter 5, we measure the effect of trace combination on each of these algorithms. We find that LEI

\(^4\) A *cyclic path* is simply a path that ends with a branch to its beginning.
traces outperform standard traces, and that trace combination improves both algorithms considerably.

In developing and evaluating these region-selection algorithms, this thesis contributes the following novel ideas:

- Traces should be selected from cyclic paths, and this can be done with low overhead.
- Regions other than traces and methods should be considered, specifically those that include many interprocedural paths.
- Selecting larger regions can result in a smaller code cache.

In order to evaluate the benefits and costs of our algorithms, we perform simulations and compute metrics that correlate with overall performance. Given their success, the next step is to implement our algorithms in the region selection module of a real dynamic optimization system and measure the performance impact.
Chapter 2

Background and Related Work

This chapter defines the generic dynamic optimization system we simulate and the specific trace-selection algorithm we use for comparison. To do so, we begin by defining a trace more precisely, which requires us to introduce the concept of a control-flow graph. We then describe the dynamic optimization system and how it executes a program. Finally, we describe the Next-Executing Tail trace-selection algorithm, the systems that use it, and how it compares to other existing trace-selection algorithms.

2.1 Terminology

Control-Flow Graphs

In order to analyze the possible paths of execution within a function, compilers often construct a control-flow graph (CFG). Formally, a CFG is a directed graph in which each node represents a basic block and each edge represents a potential control transfer from the source node to the destination node. A basic block, which we often refer to simply as a block, is a sequence of instructions with a single entry point and no internal branches or jumps—if control reaches the block, then each instruction in that block is executed exactly once in order.

Figure 2.1 illustrates the control-flow graph for a simple loop containing a conditional statement. As there is an edge from block $A$ to block $B$, we say that $B$ is a successor
2.1 Terminology

A do {
  if ( B )
    C
  else
    D
} while ( E );

Figure 2.1: Simple pseudocode and the corresponding control-flow graph.

of A and A is a predecessor of B. A block with multiple predecessors begins with a join point and one with multiple successors ends at a split point; in our example, block B is both. Intuitively, these points correspond to blocks at which multiple paths of execution join together or split apart.

One main focus of our work is to dynamically identify cycles, which correspond to loops in the original program. Block B is an example of a loop header, which begins all iterations of a loop. Formally, we say that for every block X in the body of the loop, B dominates X, which means that all paths from the entry point of the CFG to X include B. The edge from E to B is called a back edge of the loop, which returns control to the loop header. A cycle is a path that begins and ends with the same block, representing a path through a loop in the program. As there can be many paths through a loop, many distinct cycles can be executed, such as BCEB and BDEB from Figure 2.1.

Traces

Consider a run of a program during which block D executes significantly more frequently than block C. We call the branch at the end of block B a biased branch. For example, if B checks for an unlikely error condition such as failed memory allocation, C will rarely execute. By optimizing a path containing BDE without considering the constraints imposed by its relationship with C, the overall performance of the program can be improved.
[Hwu et al., 1993]. In many cases, this optimization will increase the execution time for 
$BCE$, but for a sufficiently-biased branch in $B$ this effect will be outweighed.

A *trace* is a single path of execution that is optimized without considering the effects of split 
points. The path’s first block is called the *trace entrance* and its last block the *trace exit*; 
each split point on the path creates a *side exit* for the branch target that is not included. 
Control may only enter at the trace entrance, so there are no corresponding side entrances. 
As a trace is optimized assuming that control will flow from the trace entrance to the trace 
exit—treating it as an extended basic block—*compensation code* may be required at side 
exits to maintain the semantics of the original program. Specifically, any instruction moved 
across a split point requires compensation code at the corresponding side exit. Traces were 

### 2.2 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 2.2 highlights the architecture we consider, which has been used by several dynamic 
optimization systems including Dynamo [Bala et al., 2000, Bala et al., 1999], DynamoRIO 
[Bruening et al. 2003], Mojo [Chen et al., 2000], and the Binary-translated Optimization 
Architecture (BOA) [Gschwind et al., 2000].

A dynamic optimization system begins executing a binary by emulating it in an *interpreter*. 
By initially emulating it, the system can gather 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 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. Specifically, the system performs a
Figure 2.2: Overview of execution under a dynamic optimization system. Shaded boxes correspond to parts of the system that emulate or execute original program instructions.

hash-table lookup to determine if the branch target begins a region that has been optimized and inserted into the code cache. If so, control jumps to that location and executes the optimized code natively, without any profiling overhead.

If an optimized region does not begin at the branch target, the system uses two criteria to decide whether one 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 reduces the number of regions selected for a section of code. 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.

From a specific starting address, a region is selected based on two types of information. First, it may consider profile information maintained by the interpreter about what has executed so far. Second, it may observe the sequence of blocks executed immediately after the counter reaches the threshold. Once all blocks have been selected, inexpensive optimizations—such
2.3 Next-Executing Tail Trace Selection

as strength reduction and redundancy and dead code elimination [Bala et al., 1999]—are run on the region and it is inserted into the code cache.

Once a region is selected, an exit stub is created for each exit in order to execute any necessary compensation code and transfer control back to the interpreter. To avoid the overhead of switching to and from the interpreter, region chaining is performed so that trace exits can directly target other regions in the code cache. Hazelwood and Smith (2004) show that region chaining is crucial to the overall performance of the system.

In certain situations, a region must be evicted from the code cache. If the size of the code cache is bounded, inserting a new region might require an existing region to be evicted. Even if the size of the code cache is unbounded, self-modifying code and unloaded libraries force evictions, because the optimized region no longer corresponds to the original code [Bruening and Amarasinghe, 2005].

If the input to the dynamic optimization system consists of instructions native to the target machine, the underlying hardware can be used to reduce the runtime overhead of the interpreter. This is not the case in binary translation or just-in-time compilation, but it is for transparent optimization such as in Dynamo, DynamoRIO, and Mojo. For example, DynamoRIO copies basic blocks into a separate cache and executes them natively. As any branch instruction ends a basic block, taken branches can still be profiled. Another alternative to using a software interpreter is to instrument the binary to include instructions that count taken branches and transfer control to the dynamic optimization system when a counter reaches the threshold. These choices may reduce the overhead of interpretation, but we do not focus on them because they do not directly affect region selection.

2.3 Next-Executing Tail Trace Selection

All of the dynamic optimization systems mentioned above select and optimize traces, and all except BOA do so using the _Next-Executing Tail_ (NET) trace-selection algorithm. As
shown by Duesterwald and Bala (2000), NET is very effective in selecting traces such that nearly all instructions execute from the cache, and it does so with low overhead. In later chapters, we measure the extent of separation and duplication in NET traces and we use NET as a baseline to evaluate the performance of our algorithms.

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. NET profiles each of these interpreted branch targets with a separate execution count.

When an execution count reaches a predefined threshold, NET selects a trace by interpreting and copying the path that is executed next. Although the path is not based on any accumulated information, it is statistically likely to execute frequently [Duesterwald and Bala, 2000]. 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. This definition implies that NET traces are often interprocedural as they can extend across function calls or returns. Once the 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.\footnote{We assume that exit stubs are placed immediately following the selected blocks, as is the case in DynamoRIO. One recent dynamic optimization system places exit stubs in a separate area of memory [Luk et al., 2005].}

Figure 2.3 shows the example CFG from Section 2.1 and the resulting NET traces, assuming that execution initially traverses the $BDEB$ cycle. When the execution count for $B$—the target of the backward branch from $E$—reaches the threshold, the first trace begins. It extends along the next-executed path until a backward branch is taken, forming $BDE$. These blocks are optimized as an extended basic block, and an exit stub is created for the
2.4 Related Trace-Selection Algorithms

Several other trace-selection algorithms are used in existing 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, along with the properties of effective regions that we identify, apply as much to these trace-selection algorithms as to NET. Furthermore, our algorithm

---

\footnote{The presence of these exit stubs implies that B and E end with conditional branches in the trace, but for clarity we do not show their arrows.}
for trace combination can be applied to any of these traces.

BOA is a binary translation system developed at IBM that translates PowerPC instructions to run on a high-frequency processor [Sathaye et al., 1999]. 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. When no count is above a certain threshold, the trace ends.

Wiggins/Redstone is a transparent optimization system developed at Compaq that uses a combination of hardware sampling and software instrumentation [Deaver et al., 1999]. In order 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. In this way, both BOA and Wiggins/Redstone profile multiple executions of each branch instruction and form a trace from the sequence of most frequent targets.

ADORE is a transparent optimization system developed at the University of Minnesota that uses performance counters built into the target processor [Lu et al., 2004]. 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.
Chapter 3

Properties of Effective Regions

In this chapter, we identify two properties of regions that reduce the problems of separation and duplication. In particular, we find that a region should be formed from a cycle, but it should avoid duplicating any nested cycles. Also, we find that a region should be formed from multiple paths, especially when those paths rejoin to overlap. For each property, we introduce metrics to measure how well an algorithm selects regions that satisfy it. We then discuss metrics to evaluate the overall performance of a region-selection algorithm. In the next chapters, these properties form the basis of our region-selection algorithms, and these metrics form the basis of how we evaluate them.

Each property we identify reduces both separation and duplication. There is often a trade-off between the two, as changing the length of a trace reduces one problem while increasing the other. If traces are very long, execution jumps between them less often, but they are more likely to overlap and cause duplication. At the other extreme, single-block traces cause no duplication but execution jumps between them at every branch instruction. Rather than reducing one problem at the expense of the other, we are interested in properties of regions that reduce both.

Specifically, each property reduces a type of duplication that does not improve locality of execution significantly. When considering cycles, we avoid duplicating an inner cycle in an outer cycle. Instead, all execution of the inner cycle occurs within one region. When
considering multiple paths, we reduce what we will later define to be *exit-dominated* duplication. Instead, we combine such regions into a single, larger region that has only one copy of each original block.

### 3.1 Spanning a Cycle

When a loop in a program is executed frequently, there are frequent jumps between the regions along its dominant path. Therefore, locality of execution is much better if the entire path is contained in a single region rather than split among separated regions. Bala et al. (1999) also makes this intuitive observation by considering a contrived example in which a program consists of one path through a large loop. They agree that an algorithm that selects the entire path as a single region will outperform one that splits the path into multiple regions.

As a cycle of execution usually corresponds to a loop in the original program, the first property we identify is that effective regions span cycles. We say that a region spans a cycle when repeated execution of the cycle does not exit the region. For regions that are traces, this means that the trace contains all blocks in the cycle, including one that branches to the top of the trace. For regions that include multiple paths, the cycle can be contained anywhere in the region, as long as all edges in the cycle are included.

It is useful for a cycle-spanning region to begin with a loop header or an exit from another region, as traces do in NET. A loop header is an ideal block to begin a region, as a region has a single entrance and all paths through the loop begin at the header. As we will see next, beginning a region at an exit from another region allows traces to be selected from nested loops with less separation and duplication.

Regions that span cycles reduce not only separation but also duplication if they avoid extending into nested loops. If regions span cycles, this can easily be approximated by ending a region when the next block on the path is itself the start of a region. Figure 3.1
shows how this improves trace selection on simple nested loops. For comparison, it shows the traces that NET selects: $B$, $C$, and $AB$. $B$ is selected first as the target of its own backward branch. After $B$ exits to $C$ repeatedly, $C$ 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 $B$.

This is an example of a common tendency of NET to duplicate one iteration of an inner loop in 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. As in our example, this causes each trace to contain a copy of a path through the inner loop. For a frequently-executing inner loop, this duplication does not improve locality of execution significantly. In Figure 3.1, we see that control must jump from the third trace to the first after one execution of $B$.

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. Overall, fewer blocks are selected and
3.1 Spanning a Cycle

divided among fewer traces, and it is only possible because the trace can extend across a backward branch. This same example is discussed by the implementers of Mojo, who agree that selecting $B$ and $CA$ is ideal [Chen et al., 2000].

Measuring Cycles Spanned

We measure how well an algorithm selects traces that span frequently-executing cycles by using three metrics: a potential cycle count, a spanned cycle ratio, and an executed cycle ratio.

The potential cycle count is an upper bound on the number of frequently-executed cycles that a trace-selection algorithm can span for a given program run. As any frequently-executed cycle contains a frequently-executed backward branch, we consider each backward branch that executes at least a threshold number of times. For a backward branch from block $B$ to block $A$, we determine whether the algorithm could select a trace that includes a path in the opposite direction: from $A$ to $B$. This metric allows us to compare algorithms that limit the types of branches a selected path can contain.

The spanned cycle ratio is the fraction of traces selected for a given program run that are cycle-spanning. For each trace, we determine whether any of its exit branches target its entrance. For direct branches, we can simply check the branch target, but for indirect branches we must consider all targets taken during the run. This metric allows us to say how frequently traces span cycles, but it does not indicate the effect of these cycles on execution.

The executed cycle ratio is the fraction of trace exits executed during a given program run that jump to the top of the same trace. Since all other trace exits jump to another trace or context switch to the interpreter, it measures how well the selected cycles reduce separation and improve locality of execution.
3.2 Combining Multiple Paths

An unbiased branch is an inherent problem for trace selection, because a trace can contain only one of its targets. A trace-selection algorithm splits the targets of each frequently-executed unbiased branch into separate traces. In the simplest case, each side of an unbiased if-else statement is selected in a different trace, potentially far apart in the code cache. If these traces both reach the block following the if-else statement, this code will be duplicated.

Figure 3.2 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 traces that NET would select from it, illustrating separation and duplication caused by an unbiased branch. It first selects a trace for one branch of the first conditional and later selects a second trace as the other branch is taken. Notice that the second conditional—consisting of blocks D and F and an exit stub to E—is selected in each trace.

The second property we identify solves this problem: effective regions combine multiple equally-likely paths. Rather than selecting each target of an unbiased branch in a different region, selecting them in the same region reduces separation and duplication. In the previ-
3.2 Combining Multiple Paths

Figure 3.3: Region selected for the same CFG that includes multiple paths. No separation or duplication is required.

ous example, this corresponds to selecting both sides of the first conditional, as shown in Figure 3.3. The resulting region contains no duplication and allows control to remain in the region regardless of which unbiased branch target is taken. The exit stub to block $B$ has been replaced by the block itself, and there is no need to duplicate the exit stub to block $E$.\textsuperscript{1}

This example shows that it is especially important to combine any frequently-executing paths that rejoin and overlap. Doing so avoids the duplication that would occur if they were selected separately. Moreover, rejoining paths that include a backward branch often complete cycles.

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.

\textsuperscript{1}This reduction in exit stubs is significant. Hazelwood (2004) 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.
3.2 Combining Multiple Paths

Effect on Optimization

Incorporating multiple paths into a region is likely to affect how it is optimized. First, 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, and these larger regions improve on traces in this respect by reducing the problem of separation. In the Dynamo system, roughly two-thirds of the average performance speedup on benchmarks is caused by trace selection rather than other optimizations [Bala et al., 2000]. That is, removing unconditional jumps and forming traces that cross procedure boundaries are far more important than Dynamo’s other optimizations. Regions with multiple paths have the same 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. This is the case for the region-selection algorithm we develop in Chap-
3.2 Combining Multiple Paths

ter 5. With each region, the optimizer is also given an execution count for each of its blocks. 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. Furthermore, our algorithm can also provide edge and path counts to the optimizer. Young and Smith (1998) show that path counts can guide optimization more effectively.

Measuring Missed Paths

We measure how well a region-selection algorithm combines multiple paths by considering the number of exit-dominated regions it produces. As we will see, an exit-dominated region indicates a situation in which two regions should have been combined but were not. Combining the two regions would have reduced separation. Moreover, if they contain any of the same blocks—a situation we call exit-dominated duplication—combining the two regions would have reduced 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\(^2\) 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 $S$ and separate the regions. This is the case in Figure 3.2 as the first trace exit-dominates the second.

We define exit-dominated duplication to be the total number of instructions that exist in both a region and a region that exit-dominates it. This measures the amount that exit-dominated regions overlap, and suggests the amount of duplication that can be avoided by combining additional paths. In Figure 3.2, the amount of exit-dominated duplication is the number of instructions in blocks $D$ and $F$.

\(^2\)The traditional definition of domination (from Section 2.1) 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.
3.3 Measuring Overall Performance

We evaluate the performance of our region-selection algorithms relative to Next-Executing Tail trace selection, and we do so with two questions in mind. First, how do we expect each algorithm to affect the overall performance of the dynamic optimization system? Second, how well does each algorithm address the problems of separation and duplication?

To estimate the relative overall performance of a dynamic optimization system, we adopt the “trace quality metric” used by the implementers of Dynamo, who designed the NET algorithm [Bala et al., 1999]. 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 algorithms, the authors compare the 90% cover sets of each against the overall performance of running Dynamo with each. For all algorithms on each of six benchmarks, the two correspond perfectly: if one algorithm results in a smaller 90% cover set than another, then it also results in a larger performance speedup.

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. For a given hit rate, we measure the problem of duplication by comparing the amount of code expansion produced by two region-selection algorithms.

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. Locality of execution within a region will be very good, as its blocks are stored together in memory. Therefore, for a given amount of
execution in the cache, we measure the problem of separation by comparing the number of region transitions two algorithms require.

### 3.4 Methodology

To determine the effect of our algorithms on these performance metrics, we created a framework for simulating a dynamic optimization system and implemented each within it. It relies on the Pin dynamic instrumentation system to report the sequence of basic blocks executed by a program [Luk et al., 2005]. The framework looks for each taken branch and simulates the generic dynamic optimization system described in Section 2.2. All details of region selection have been abstracted out of the framework, allowing it to gather data for each region-selection algorithm without modification.

We remove the question of cache management from our study of region selection by using an unbounded code cache. Intuitively, however, our algorithms should improve a system built with a bounded code cache because they reduce the problems of separation and duplication. When regions are evicted, separation and duplication have especially large effects. Separation increases the chance that some related regions are evicted but not all, while duplication increases the frequency of eviction. Existing work shows that evicting a smaller number of larger regions reduces the overhead of cache management [Hazelwood, 2004]. However, studying these effects in detail is outside the scope of this thesis.

We present data for running each of the twelve SPECint2000 benchmarks to completion on its test input [SPEC, 2000]. When there are multiple test inputs for a benchmark, the input that requires the program to execute the most instructions is used. Appendix A lists the command-line arguments given to each benchmark.

For each algorithm, we focus on the metrics that give the most interesting results. For example, we only show the number of exit stubs an algorithm produces when it has changed disproportionately relative to the change in code expansion. We find that all algorithms we
consider achieve a consistently high hit rate, so rather than examine each benchmark we
discuss general trends across benchmarks and highlight the few that changed in a significant
way.
Chapter 4

Last-Executed Iteration

Trace Selection

In this chapter, we focus on applying the principle that effective traces span cycles. First, we measure how frequently the Next-Executing Tail (NET) trace-selection algorithm spans cycles, and we find that it has a fundamental weakness in this respect. To address this, we develop a trace-selection algorithm that explicitly selects cycles but does so with overhead comparable to NET. We evaluate our algorithm relative to NET and find that it reduces both separation and duplication while requiring significantly less memory for profiling.

4.1 Analysis of Next-Executing Tail

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 [Bala et al., 2000]. Moreover, traces are selected across procedure boundaries and can include any forward branch, so an indirect jump will not necessarily end a trace before it reaches a loop back edge.

However, two properties of NET suggest that a single trace will often fail to span the entire path from loop header to backward branch. First, NET selects an interprocedural forward path, as defined by Duesterwald and Bala (2000), that ends with a call to a function
at a lower address or return from a function at a higher address: “if a path includes a (for-
ward) procedure call it will terminate at the corresponding return branch.” Therefore, NET
cannot extend a trace across both a function call and its corresponding return. As shown
in Figure 4.1, if a loop contains a function call on its dominant path then NET will select
two separate traces. 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. Hank et al. (1995) demonstrates that a high percentage of execution
in common benchmarks is within such cycles.

Second, NET selects a trace from the target of a backward branch by observing only one path
of execution. Although the path is in some sense a random sample of execution from the top
of the loop—and thus most likely to be the dominant path [Duesterwald and Bala, 2000]—
at most one cycle can be spanned. If the path of execution does not end with a branch to
the initial target, it spans no cycles. Instead, important cycles will be split over multiple
traces. This problem is fundamental to trace selection, as it suggests a region including
multiple cycles should be selected after multiple paths of execution are considered. We
analyze region-selection algorithms based on these ideas in the next chapter.

To analyze the effect of the first property, we use the potential cycle count discussed in
Section 3.1. We compare the number of potential cycles for NET traces to the number of
Figure 4.2: Number of potential cycles spanned by NET relative to the number it spans if traces can extend across backward function calls and returns. Only backward branches taken at least 100 times that are not calls or returns are considered.

potential cycles for these traces if backward calls and returns are allowed. By doing so, we can measure the proportion of cycles missed by NET because it does not extend across both a call and its corresponding return.

Figure 4.2 presents the results when considering backward branches taken at least 100 times that do not themselves correspond to function calls or returns. As NET extends across any taken branch, we base this analysis on a profile of each benchmark that includes indirect jump targets. As NET extends only across taken branches, we only include edges in the CFG traversed at least once. The results present a convincing case that stopping at backward calls and returns hinders NET’s ability to span cycles: on average, 14% fewer backward branches are reachable from their respective targets. This suggests that trace selection can be improved by incorporating backward function calls and returns, provided that the algorithm does not introduce the same code expansion that NET would with this modification.

4.2 Last-Executed Iteration Algorithm

We propose a trace-selection algorithm that extends traces across backward branches but avoids code expansion from duplicating nested cycles. Last-Executed Iteration (LEI) selects
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 indicates 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 source and destination 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 4.3 shows how LEI responds 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 branch 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 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 4.4 shows how LEI uses the history buffer to form a trace. The loop beginning
4.2 Last-Executed Iteration Algorithm

INTERPRETED-Branch-Taken( history buffer Branches , address src , address dest )
1 address cached ← Hash-Lookup( code cache , dest )
2 if cached
3 then jump cached
4
5 Circular-Buffer-Insert( Branches, src , dest )
6 if Hash-Lookup(Branches.hash, dest )
7 then history buffer location old = location of dest in Branches
8 Hash-Update(Branches.hash, dest , last element in Branches )
9 if dest ≤ src or old follows exit from code cache
10 then increment counter c associated with dest
11 if c = T \_cyc
12 then newTrace ← Form-Trace(dest)
13 remove all elements of Branches after old
14 recycle counter associated with dest
15 jump newTrace
16
17 else Hash-Insert(Branches.hash, dest , last element in Branches )

Figure 4.3: Algorithm for Last-Executed Iteration trace selection

FORM-Trace( history buffer Branches , address start )
1 trace newTrace ← \emptyset
2 address prev ← start
3 for each branch in Branches after start
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.dest ∈ newTrace
13 then break
14
15 prev ← branch.dest
16 return newTrace

Figure 4.4: Algorithm for forming an LEI trace given a history buffer and a starting address.
4.2 Last-Executed Iteration Algorithm

Figure 4.5: A loop with a function call on its dominant path, and the contents of the branch history buffer after three iterations.

on line 3 iterates over each subsequent taken branch in the buffer to determine the executed path. For a taken branch, all instructions from its destination 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 is only invoked when an interpreted branch is taken, and in all benchmarks over 98% of execution occurs in 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 one buffer insertion and one hash table lookup for the history buffer, as the hash table update or insertion can be combined with the lookup. LEI compensates for these additional operations by requiring fewer counters, as we will see in the next section.

Figure 4.5 illustrates the case described in Section 4.1: a loop that contains a function call on its dominant path. Whereas NET selects the two traces in Figure 4.1, LEI selects the 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.

Figure 4.5 also shows the contents of the history buffer for three iterations of this loop, assuming the cycle through $B$ is executed. It highlights several important properties of LEI. The branch history buffer only contains taken branches, so the fall-through path from $A$ to $B$ is not included. Relatively few branches 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, we see that the branch history buffer represents the full interprocedural cycle.

### 4.3 Experimental Results

Using the simulation framework discussed in Section 3.4, 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 [Bala et al., 2000, Bala et al., 1999, Duesterwald and Bala, 2000]. As LEI only counts 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 to be 500. Intuitively, this seems small enough to require little memory but large enough to capture very large cycles and those with frequently-executing nested cycles.

Although the hit rate remains above 99% for every benchmark but two, it is noteworthy that LEI achieves a slightly lower hit rate than NET in most cases. 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
4.3 Experimental Results

Figure 4.6: 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).

Chen et al. (2000) could compensate for this difference, but we cannot evaluate this trade-off without runtime measurements from a dynamic optimization system that implements LEI.

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 (as defined in Section 3.1). Together, they measure how many additional cycles LEI spans and how much of an effect these cycles have on locality of execution.

Figure 4.6 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 cycles. 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
4.3 Experimental Results

The size of the 90% cover set for each benchmark is given in Figure 4.7. 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 [Bala et al., 1999]. 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.

**Code Expansion and Locality of Execution**

In order to examine the performance of LEI in detail, we focus on two potential problems with the approach. As LEI selects traces that extend across arbitrary backward branches, we must consider whether it results in more code expansion. As LEI limits code duplication by ending a trace when another is reached, we must also consider whether locality of execution is sacrificed. In fact, we find that LEI balances code expansion and locality of
4.3 Experimental Results

Figure 4.8 shows that LEI produces less code expansion than NET while improving the locality of execution in the code cache. For all benchmarks but crafty, the elimination of cycle duplication outweighs the effect of selecting traces across forward and backward branches, so on average LEI results in 92% of the code expansion of NET. Despite promoting fewer instructions to 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 cyclic trace exits, this causes a significant improvement in locality of execution: on average the number of region transitions 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 4.8 stand out: code expansion for crafty and the number of region transitions for parser. In both cases, LEI performs no better than NET. Considering Figure 4.6, 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, where the more additional cycles LEI spans the more it outperforms NET. This is intuitive, as the primary difference between LEI and NET is its emphasis on spanning cycles.
Memory Overhead

Finally, we consider the memory overhead that LEI requires for counter variables. Memory overhead is an important issue for profiling techniques, and one of NET’s strong points is that it only requires a counter for some branch targets [Duesterwald and Bala, 2000]. 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 what targets to associate with a counter. 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 4.9 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.

4.4 Summary

In this chapter, we have developed an algorithm to select a trace from a cycle that can include any type of branch. By doing so, it can select a trace that spans an interprocedural cycle, which NET cannot. We find that this results in many more cycle-spanning traces and reduces the problem of separation significantly. Also, our trace-selection algorithm avoids duplicating nested cycles, and we find that this results in less code expansion. Overall, selecting a trace from a cycle improves performance metrics and requires very little overhead.
Chapter 5

Trace Combination

In this chapter, we address the problems inherent to trace selection by developing an algorithm that selects regions containing multiple interprocedural paths. The difficulty lies in efficiently deciding which paths to include and which to exclude. More profiling provides more information, but it incurs more overhead. A region-selection algorithm must balance these factors. Ours selects regions by combining multiple traces, and it 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 first confirm that NET and LEI frequently produce exit-dominated traces and that these traces include exit-dominated duplication, as defined in Section 3.2. We then introduce our algorithm for trace combination, which incorporates the observation from Chapter 3 that it is especially important to select rejoining paths together. Storing traces that will be combined requires additional memory, but we use a compact representation that requires little memory relative to the size of the code cache. By evaluating our algorithm using NET and LEI, we find that it reduces exit domination and improves all of our performance metrics. This includes significantly reducing the number of exit stubs, which we show offsets the additional memory required by trace combination.
5.1 Analysis of Trace-Selection Algorithms

Figure 5.1: The proportion of traces selected by NET and LEI that are exit-dominated.

5.1 Analysis of Trace-Selection Algorithms

Figure 5.1 shows that both NET and LEI select many exit-dominated traces. This suggests unnecessary separation, as there is no way to enter the dominated trace except by exiting the dominating trace. On average, 15% of NET traces and 22% of LEI traces are exit-dominated. If dominated traces were selected in the dominating region, locality of execution would likely 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 exit-dominate large numbers 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 such traces are not considered, eon’s proportion of exit-dominated traces decreases to just above average.

Figure 5.2 suggests that not only would combining exit-dominated traces improve locality of execution, but it would also cause less duplication. It shows the proportion of all instructions selected that are exit-dominated duplication, which ranges from 1 to 7% over
all benchmarks. This implies that it is common for an exit-dominated trace to rejoin the dominating trace. As expected, eon does not have a large amount of duplication among its exit-dominated traces because many of them are in different functions.

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; Chapter 4 demonstrated the opposite. Rather, it shows that both are likely to be improved by trace combination. As LEI has more exit domination, trace combination should be especially beneficial for it. As we will see in Section 5.3, this is very much the case.

5.2 Trace Combination Algorithm

Trace combination is based on a simple extension to trace selection. A trace-selection algorithm profiles certain branch targets until one executes a specific 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 observed traces.
5.2 Trace Combination Algorithm

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 select a trace beginning at dest
9 store a representation of the trace in memory
10 if c = T_start + T_prof
11 then recycle counter c
12 CFG G ← combine observed traces beginning at dest, counting block frequencies
13 mark all blocks that appear in at least T_min traces
14 MARK-REJOINING-PATHS(R)
15 remove from G all unmarked blocks
16 remove any region exit that targets a block in G
17 address newRegion ← OPTIMIZE-AND-INSERT-REGION(G)
18 jump newRegion
19

Figure 5.3: Algorithm for trace combination.

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 it. This is because all observed traces contain the same blocks. As we develop our algorithm further, we see that it selects multiple paths only when each executes frequently.

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 all blocks that occur in frequently-executed traces and then it includes rejoining paths to those blocks.

Figure 5.3 provides an overview of the algorithm and shows how it selects blocks that occur in many 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 in at least $T_{\text{min}}$. 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. To explore these questions, we consider the example in Figure 5.4, which consists of two conditionals with a loop in one branch of the first. The figure shows its control-flow graph and how the blocks might be arranged in memory. We assume for this example that $T_{\text{prof}} = 15$ and $T_{\text{min}} = 5$, which implies that 15 traces are observed and blocks that appear in at least 5 are initially marked.

**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 where an observed trace overlaps with others while storing it, because the counter associated with their entrance may never reach the threshold. Although this avoids unnecessary analysis, it might require significantly more
5.2 Trace Combination Algorithm

**COMPACT-TRACE**( trace \( t \) )

1. bitstring \( b \)
2. for each branch in \( t \) 
3. do if branch is taken to an address not contained in the instruction 
4. then append “01” to \( b \) followed by the target address 
5. else if branch is conditional 
6. then if branch is not taken 
7. then append “10” to \( b \) 
8. else if branch is taken 
9. then append “11” to \( b \) 
10. append “00” to \( b \) 
11. append the address of the last instruction in \( t \) to \( b \) 
12. return \( b \)

<table>
<thead>
<tr>
<th>Path</th>
<th>Conditional Branches</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-&gt;C</td>
<td>taken, end C</td>
<td>1100&lt;C_end&gt;</td>
</tr>
<tr>
<td>A-&gt;C-&gt;D-&gt;F</td>
<td>taken, not taken, taken, end F</td>
<td>11101100&lt;F_end&gt;</td>
</tr>
<tr>
<td>A-&gt;B-&gt;D-&gt;F</td>
<td>not taken, taken, end F</td>
<td>101100&lt;F_end&gt;</td>
</tr>
</tbody>
</table>

Figure 5.5: Algorithm for representing a trace compactly, and representations for traces from the example.

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.\(^1\) Normally, this representation of a path introduces overhead as it requires the path be traversed. However, the optimizer must already decode each instruction in the region and identify all branch targets. This representation 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 unknown target, the representation must explicitly include the target address, requiring an additional 32 or 64

---

\(^1\)In fact, we ignore any unconditional call or jump when the target address is known, because all paths that reach such an instruction continue with its target.
5.2 Trace Combination Algorithm

bits. As a trace can end at any instruction, the address of the end of the trace is appended to its representation. Figure 5.5 gives the algorithm and representations for three possible traces from our example.

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 target not yet in the CFG, a block is created for it. The end of this block is determined by examining each subsequent instruction until a branch is found or the start of an existing block is reached. In this way, individual instructions are only decoded the first time they are encountered in a trace.

The downside of making only one pass over the observed traces is that branch targets are not known until the corresponding branch is encountered. This allows a situation in which a taken branch target is in the middle of an existing block. In the event that this happens, the existing block is split into two blocks: one represents the instructions before the branch target and the other begins at the target. Thus, when a branch target is already in the CFG, the algorithm ensures that it begins a block and adds an edge from the current block to it.

\[^2\text{This is true even of NET, which not only ends a trace at a backward taken branch but also when it contains a maximum number of instructions.}\]
This process constructs a valid CFG that represents the observed traces, because each trace instruction is in exactly one block and each taken branch corresponds to an edge between blocks. Each trace is considered once, and each instruction is decoded once. It is simple to include block frequencies as the CFG is built by incrementing a block counter each time it is traversed. This allows all blocks that appear at least \( T_{\text{min}} \) times to be marked as the CFG is constructed.

Figure 5.6 shows a CFG as it is built from the traces given in Figure 5.5. The first trace builds the CFG in Figure 5.6(a) where an edge with no destination block represents a branch target that exits the region. The next seven duplicate traces simply traverse the blocks and increment each counter. When a counter reaches \( T_{\text{min}} \), the block is marked, which we illustrate by shading the block. The next set of traces expands the CFG to Figure 5.6(b) as the path is extended to include blocks \( D \) and \( F \). Finally, Figure 5.6(c) shows the CFG after the last new trace is added. A block is created for \( B \) that ends with a branch to \( D \). As \( D \) already begins a block in the CFG, an edge is added and the existing path through \( D \) and \( F \) is traversed. The resulting CFG consists of five blocks, four of which are marked. It contains two side exits, but later in the algorithm the exit from \( C \) will be recognized as a self-loop and removed.
Mark-Rejoining-Paths( marked CFG $G$ )

1 repeat
2 $changed \leftarrow false$
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 $changed \leftarrow true$
8 9 until $changed = false$

Figure 5.7: Algorithm for marking rejoining paths.

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 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.

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 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 5.7. 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 is ever unmarked. Therefore, for a CFG with $n$ blocks and $e$ edges, there are at
As each iteration considers each 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.

Running this algorithm on the CFG produced in Figure 5.6 marks block $B$. Conceptually, this is because $B$ is on the path that exits the marked region at $A$ and rejoins it at $D$. Algorithmically, it is because $B$ has a marked successor, $D$.

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, for example $C$ in Figure 5.4. This happens also with a trace, but as this algorithm selects regions with split and join points, the exit can be replaced with an edge. Doing so keeps control in the region for longer and reduces the number of exit stubs required. Figure 5.8 shows the region that is optimized and inserted in the code cache, and it also shows one possible layout of the

Figure 5.8: The final region selected, and one possible layout of it in memory.
region in the cache. As the optimizer has been given block counts for the selected region, it identifies $ACDF$ as frequently-executing and positions it contiguously.

## 5.3 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 draw 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 3.3 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 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 continue to use $T_{prof} = 15$ and $T_{min} = 5$, this means that combined NET begins profiling after 35 executions 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.\(^3\)

\(^3\)For example, setting $T_{prof} = 5$ and $T_{min} = 2$ results in smaller but similar improvements.
5.3 Experimental Results

As with LEI, our trace combination algorithm has a very small effect on hit rate so we do not discuss it in detail. 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.

Reducing Exit Domination

As expected, trace combination reduces the amount of exit-domination for all benchmarks. Figure 5.9 shows the exit-dominated duplication that remains in each benchmark. Compared with Figure 5.2, it shows that on average trace combination reduces roughly 65% of exit-dominated duplication caused by the trace-selection algorithms. The number of exit-dominated regions shows a reduction of roughly 40%, and full details are given in Appendix A. While the amount of exit-dominated duplication is higher for LEI, trace combination reduces it by a greater amount.

These results highlight two important properties of our trace combination algorithm. First, it does not avoid all exit-domination, as regions are selected from paths that occur frequently in a sample of $T_{prof}$ 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 [Sherwood et al., 2002]. 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.

**90% Cover Sets**

Figure 5.10 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. 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%,
5.3 Experimental Results

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 for a trace.

Code Expansion and Locality of Execution

Figure 5.11 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.

For vortex with 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 $T_{\text{min}}$ 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 ob-
servation that the average region size increases far less in vortex than in other benchmarks. However, Figure 5.11 shows that the overall effect of trace combination on locality of execution is strongly positive.

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. A breakdown of the change in code expansion by benchmark is given in Appendix A.

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 $T_{\text{min}}$ 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.

**Memory Overhead**

Figure 5.12 shows the memory overhead of trace combination, computed as the maximum amount of memory needed at any point in the program 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 bytes in all instructions inserted in the code cache and add 10 bytes for each exit stub. In the DynamoRIO system, each exit stub requires a minimum of three instructions [Hazelwood, 2004] 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
5.3 Experimental Results

Figure 5.12: Maximum amount of memory overhead required for storing observed traces, computed as a percentage of the estimated size of the cache.

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 a 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 5.13, trace combination reduces the number of exit stubs significantly: 18% fewer are required with NET and 26% fewer are required with 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.

---

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

In this chapter, we have developed an algorithm to select a region that can include many interprocedural paths. It is based on observing and combining multiple traces, which allows it to be applied to any trace-selection algorithm. We have applied it to NET and LEI traces and found that it reduces the problems of separation and duplication for both. Trace combination improves locality of execution significantly, and it does so while reducing the size of the code cache. Moreover, each improvement is larger for LEI than for NET, which suggests that our algorithms are especially effective when combined.

Figure 5.13: Effect of trace combination on the number of exit stubs produced by NET and LEI.
Chapter 6

Conclusion

Applications are continuing to grow in size, and existing trace-selection algorithms are not designed to handle the resulting increase in the number of paths. Instead, the problems of separation and duplication lead to degraded performance as locality of execution decreases and memory overhead increases.

Motivated by these problems, we have developed two region-selection algorithms that select a smaller number of larger regions. By doing so, they are able to select more related code together and improve locality of execution. Each algorithm is designed to decrease a type of duplication that yields little benefit, which enables each to simultaneously reduce memory overhead.

The success of our algorithms is best summarized by comparing their combined performance to the standard trace-selection algorithm, Next-Executing Tail. By performing slightly more analysis, they reduce code expansion by 9% and the number of exit stubs by 32% while 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.

Given this success, our main conclusion is that region-selection for dynamic optimization is an important and promising area of research. To date, only two regions have been seriously
considered: an entire method or a single trace. Moreover, very few different trace-selection algorithms are used in practice. This thesis shows that not only can more effective trace-selection algorithms be developed, but different regions should be considered.

6.1 Future Work

We show that our algorithms improve metrics that are important to performance, so the next step is to implement them in a dynamic optimization system and measure the actual performance improvement. With this in place, we can measure the relative importance of hit rate, code expansion, and region transitions and tune our threshold values accordingly. For trace combination, it is important that the system implement aggressive optimizations, as we believe our larger regions will make them more beneficial.

Implementation will also allow us to measure the run-time overhead of observing and storing multiple traces. If this overhead is high, the trace combination algorithm could be modified to observe fewer traces and expand the resulting region with static program-based heuristics. As shown by Wu and Larus (1994), conditional branch bias can be approximated simply by examining the code for the condition.

Furthermore, trace combination is not the only way to select a region that includes multiple interprocedural paths. Another approach would begin with a whole method and remove blocks that are not expected to execute frequently. With sufficient inlining, this could produce a region that spans procedure boundaries and includes both sides of unbiased branches.

More generally, the future work suggested by this thesis is that new region-selection algorithms should be developed and implemented in dynamic optimization systems. Our work only begins to reduce the problems of separation and duplication, but it shows that substantial improvements can be made to the few algorithms currently in use.
## Appendix A

### Additional Data

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Command Line</th>
<th>Dynamic Inst. Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>input.compressed 2</td>
<td>2,411,758,185</td>
</tr>
<tr>
<td>175.vpr</td>
<td>net.in arch.in place.out dum.out -nodisp -place_only -init 5 -exit 0.005</td>
<td>1,634,126,912</td>
</tr>
<tr>
<td></td>
<td>-alpha 0.9412 -inner_num 2</td>
<td></td>
</tr>
<tr>
<td>176.gcc</td>
<td>cccp.i</td>
<td>1,372,035,086</td>
</tr>
<tr>
<td>181.mcf</td>
<td>inp.in</td>
<td>136,456,123</td>
</tr>
<tr>
<td>186.crafty</td>
<td>&lt; crafty.in</td>
<td>5,037,135,902</td>
</tr>
<tr>
<td>197.parser</td>
<td>2.1.dict -batch &lt; test.in</td>
<td>3,284,153,644</td>
</tr>
<tr>
<td>252.eon</td>
<td>chair.control.kajiya chair.camera chair.surfaces</td>
<td>2,209,764,915</td>
</tr>
<tr>
<td></td>
<td>chair.kajiya ppm ppm pixels ppm.kajiya</td>
<td></td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>-I ./lib diffmail.pl 2 24 15 24 23 24</td>
<td>1,485,520,816</td>
</tr>
<tr>
<td>254.gap</td>
<td>-l ./ -q -m 64M &lt; test.in</td>
<td>830,070,095</td>
</tr>
<tr>
<td>255.vortex</td>
<td>lendian.raw</td>
<td>7,903,295,624</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>input.random 2</td>
<td>7,559,134,179</td>
</tr>
<tr>
<td>300.twolf</td>
<td>test</td>
<td>217,438,649</td>
</tr>
</tbody>
</table>

Table A.1: Benchmarks used and the command-line arguments given to each, along with the resulting dynamic instruction count as reported by Pin.
Figure A.1: The proportion of regions selected by combined NET and combined LEI that are exit-dominated.

Figure A.2: Effect of trace combination on code expansion for NET and LEI. mcf with LEI appears significantly higher than the rest partially for two reasons: it is the smallest benchmark, and Figure 4.8 shows that its code expansion decreases the most from NET to LEI.
Bibliography


