An API for Dynamic Partial Evaluation under DynamoRIO

by

Christopher Leger

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

© Massachusetts Institute of Technology 2004. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by .....
Gregory T. Sullivan
Research Scientist
Thesis Supervisor

Certified by .....
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

An API for Dynamic Partial Evaluation under DynamoRIO

by

Christopher Leger

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Dynamic optimization systems have available runtime (dynamic) data and can cross more boundaries than traditional static optimization systems in the pursuit of program transformations. However, dynamic optimization systems are limited by the fact that any time spent in analysis or transformation is included in the running time of the program; thus any transformations must not only improve the performance of the program, but also make up for the time spent in analysis.

Interpreters in general confound current heuristics for dynamic optimization. In previous work, the dynamic optimization system `DynamoRIO` was extended with `DynamoRIO-with-Log-PC` to address this issue by maintaining extra state; this thesis generalizes the additions made by `DynamoRIO-with-Log-PC` and develops a useful API for a wider range of programs running under the `DynamoRIO` system.

Thesis Supervisor: Gregory T. Sullivan
Title: Research Scientist

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

I would like to thank my thesis supervisor, Greg Sullivan, for his constant support, encouragement, and availability. Greg was always helping through discussions, pointing to related work that could help, focusing on the details when I was thinking abstractly, and looking at the big picture when I was mired in minutiae. Greg helped to hash out the goals of this work and issues that crept up. He also encouraged me to write down my thoughts and plan everything out step by step, a difficult thing for me.

I am also grateful to Professor Saman Amarasinghe, for making this thesis possible. His thoughtful comments and suggestions have improved both my work and this thesis.

I would also liked to thank the entire `DynamoRIO` group. Although it is such a low-level, complex system, `DynamoRIO` has been designed well with ample comments and clean division between modules. It was a pleasure working with `DynamoRIO`, and I can only hope that my modifications were as clear and well-explained as the original system.

The work in this thesis would not have been possible without the loving support of my fiancee Michelle Duvall. I would like to thank her profusely. She has helped me in every way and this work and my life are much better for her influence.

And finally, I would like to thank my family. My parents, Wayne and Loretta, have always done anything they can for me, and have always been supportive. They have taught me the value of hard work, persistence, and achieving goals. I would also like to thank my siblings, Art and Steph, Cisco, Aja, and Carlos. I have learned a lot from each of them in their own way, and I look forward to being able to spend more time with them in the future.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Static analysis at its best is limited to optimizations performed on static data; the content of dynamic data cannot be used in analysis. Furthermore, with the increased usage of run-time binding paraphernalia (e.g. dynamic libraries, etc.) static analysis of programs is becoming more limited in the transformations it can do to improve performance. One way to obtain better performance is through dynamic optimization. This allows optimizations to cross the boundary of dynamically linked code, as well as allowing optimizations to be performed concerning the dynamic (run-time) input to the program.

One particular class of programs relying heavily on runtime data is interpreters. Interpreters have two forms of runtime data: the program to be interpreted, and any input to the program. Interpreters also have more to gain from dynamic optimization; interpreters inherently have an interpretive overhead in producing the desired output. Thus, previous efforts in dynamic optimization through partial evaluation (`Dynamo-RIO-with-Log-PC`) have focused on improving the performance of interpreters. This thesis follows in that vein, but provides a more general API to guide dynamic optimization for a broader range of applications.

Our work builds on top of an existing dynamic optimization system for Intel's x86 [20] architecture, `DynamoRIO`. `DynamoRIO` has previously been modified to apply the techniques of partial evaluation for the declarative dynamic partial evaluator `DynamoRIO-with-Log-PC`. In this thesis, we present an alternative expansion of

DynamoRIO, aiming for a more generally applicable system than DynamoRIO-with--Log-PC.

This thesis presents an API for assisting a dynamic native translator in optimizing execution through native partial evaluation. In the first chapter we introduce the thesis and provide an outline. The second chapter presents some related work and alternative bases. The third chapter describes DynamoRIO, and the Logical PC extensions made to effect DynamoRIO-with-Log-PC. The fourth chapter analyzes a class of situations in which DynamoRIO can perform more optimizations, while the fifth chapter demonstrates the changes necessary to effect better optimization. The sixth chapter describes the internal details of the modifications necessary to improve optimization. The seventh chapter describes the current state of DynamoRIO-PE, describing current mechanisms and presenting some high-level results. Finally, the eighth chapter concludes and touches upon future work.

Chapter 2

Related Work

Our work inherits from three main, interrelated areas: partial evaluation, dynamic code generation, and binary translation. This section introduces concepts and related work from each category.

2.1 Partial Evaluation/Specialization

The authoritative source on partial evaluation is that of Jones, Gomard and Sestoff [21]. This work introduces the notion of partial evaluation and presents a partial evaluator, MIX.

The idea of program specialization is to provide a specialized version of a more general program by considering some input as fixed, or *static* (in functional languages this is related to currying, or binding of variables). For instance, consider a program P with two input types S and D, returning a result of type R (P : S x D $\rightarrow$ R). For each input s $\in$ S, there is also a derivable program P_s requiring one piece of input d, such that for all values $d \in D$: P_s(d) = P(s,d). Furthermore, any code conditioned on the value of s can be resolved in P_s. Figure 2-1 (adapted from [21]) presents a general function (`f`) for raising a number (`x`) to an integer exponent(`n`) and a specialized version ($\texttt{f}_5$) specialized to `n` = 5.

In the example, `n` is defined as *static*, since its value is known at specialization time, while `x` is *dynamic*, since its value is unknown. One note of interest is that

A two-input program p =

```
f(n, x) :  if n = 0 then 1
           else if even(n) then f(n/2, x) ↑2
           else x ∗ f(n-1,x)
```

Program `p`, specialized to static input `n` = 5:

$p_5 =$ $f_5(x) = x * ((x\uparrow 2)\uparrow 2)$

Figure 2-1: Specialization of a program to compute x^n. Notice how the conditional code has been resolved, recursive calls to `f` have been removed, and we are left with a single straightforward expression.

using `n` as static greatly simplifies the computation, wherease using `x` as static would not have resulted in any noticeable improvement. We will say that `f` is *specialized against* (or *specialized with respect to* `n`) when `n` is static.

Partial evaluation (PE) is the process that automates program specialization. A partial evaluator takes as input a program and a set of static data; the program is then partially evaluated by performing calculations depending only on the static data. The output of partial evaluation is another program that requires the dynamic data as input and produces the expected result.

There are, traditionally, two main classes of partial evaluation: *on-line* and *off-line*. An on-line partial evaluator takes as input the source of a program P and the static values of a subset of the program input, S. Through symbolic computation on the available data the evaluator produces the source for a specialized program P_S. An off-line partial evaluator divides the work into two steps. In the first step, *binding-time analysis* is provided with which input is to be static and propagates information about other constructions that will be static based on the provided classification. The second phase is subsequently provided with the actual value of the static data and produces specialized code. Note that the terms online and offline refer to the time at which specialization is done with respect to the static values; online systems can perform partial evaluation at compile-time (statically) or at run-time (dynamically), as can offline systems.

This section introduced the notion of program specialization through partial evaluation and provided an example. The next section describes the related field of dynamic code generation, providing details for some off-line partial evaluators that

generate code dynamically.

2.2 Dynamic Code Generation

We define the term *dynamic code generation* as the act of producing machine code at runtime. Most partial evaluators have the ability to produce specialized code at runtime; however, such systems generally require guidance on what and when to specialize, as well as which state to make static and which dynamic. Among dynamic code generating partial evaluators, there are three broad categories: imperative, declarative, and automatic. The imperative approach requires the user to write programs that explicitly generate and manipulate the dynamic constructs at run-time. The declarative approach, on the other hand, only requires the user to provide annotations to guide dynamic code generation. Finally, the automatic approach determines how to specialize without user intervention.

2.2.1 Imperative Code Generation

This subsection presents a few example systems using an imperative approach to dynamic code generation and describes some of the trade-offs. The imperative approach requires the user to explicity manipulate program fragments at runtime.

One example of a system taking the imperative approach is dcg [17]. The API presented by dcg allows the programmer to generate dynamic code through constructs representing the intermediate representation (IR) of the LCC compiler, one method at a time. Through the abstraction of the LCC IR, dcg is able to easily provide portability; client code does not require alteration when linked with dcg routines on a new target machine. Once the client is ready to compile its generated method, it calls the `dcg_gen` method and is provided a pointer to the generated code. The `dcg_gen` method translates the IR tree into machine instructions for the targetted machine; reportedly at the rate of one machine instruction produced for every 350 instructions.

‘C (Tick C) [16, 25] is a further project loosely based on DCG. ‘C itself is a language definition for dynamic code generation; the ‘C language is a superset of

```
void cspec mkscale(int **m, int n, int s) {                              1
  return `{                                                              2
    int i,j;                                                             3
    for(i=0; i < $n; i++) { /* Loop can be dynamically unrolled */       4
      int *v = ($m)[i];                                                  5
      for(j = 0; j < $n; j++)                                            6
        v[j] = v[j] * $s; /* Multiplication can be strength-reduced */    7
    } };                                                                 8
}                                                                        9
```

Figure 2-2: ‘C code to specialize multiplication of a matrix by an integer. The value returned is a dynamic code specification which will need to be `compile`d before being used. Notice that the $ indicates data that is considered static.

ANSI C. ‘C allows the programmer to generate *code specifications* in ANSI C, which are static descriptions of dynamically generated code. These code specifications can be composed at run time to build larger specifications. These specifications can then be dynamically compiled, at which time the program obtains a pointer to the generated code. See Figure 2-2 (taken from [16]) for an example showing the dynamic creation of code designed to multiply a matrix by a integer scalar.

Tcc is an implementation of a compiler for the ‘C language, based on lcc. It uses VCODE [15] as one of the possible runtime code generators. VCODE is a template-based system for dynamic code generation, providing a similar interface to that of an ideal RISC architecture.

The imperative approach to dynamic code generation allows the user to have almost complete control over the generated code. This is, of course, both an advantage and a disadvantage. Although it offers great flexibility and control, it requires significant programmer effort and is difficult to debug. The next subsection describes an alternative, the declarative approach.

2.2.2 Declarative Code Generation

The declarative approach to dynamic code generation requires the user to provide annotations guiding the partial evaluator in choosing how to perform its specializations, but the system then performs the actual specializations.

DyC [18] provides functionality for dynamic code generation with two calls: `make_static` and `make_dynamic`. At the most basic level, these calls simply denote regions around which to classify variables as either static or dynamic. When a variable is marked as static, DyC generates a generating extension (GE) to produce code specialized against the runtime value of said variable until a corresponding `make_dynamic` call (or the variable exits scope). The region of code between a `make_static` annotation for a variable and the corresponding `make_dynamic` is called a *dynamic specialization region*, or *dynamic region* for short.

One major difference between DyC and other dynamic code generating systems is its flexibility in dynamic regions allowed. Specifically, it allows both *polyvariant specialization* and *polyvariant division*. Polyvariant specialization allows multiple compiled versions to be produced, each specialized for different values of the static variables. Polyvariant division allows the same program point to be analyzed for different combinations of variables being treated as static. Additionally, dynamic regions for different variables can overlap, be contained in, or interact in any way with dynamic regions of other variables. See Figure 2-3 (expanded from [18]) for an example of its flexibility. As is common throughout off-line partial evaluators, DyC runs a binding-time analysis to generate the necessary GEs which are then run at runtime to produce code specialized against the values of static data.

In addition to the basic functionality presented above for DyC, at each `make_static` call site, each variable can be augmented with a list of policy choices to guide DyC in specialization. For instance, one policy choice represents how likely it is for a dynamic region to occur with the same static values; if only a single static value will be present, more time and effort can be spent in optimizing for that value (as it will be used frequently and cost can be amortized). At the other extreme, if a static value will only occur a single time, there's no point trying to optimize highly or test to see if the value has already generated code.

Tempo [10, 11] is another declarative partial evaluator for the C programming language, driven by the goal of improving the performance of actual systems level applications. Tempo defines a declarative language for the developer to describe

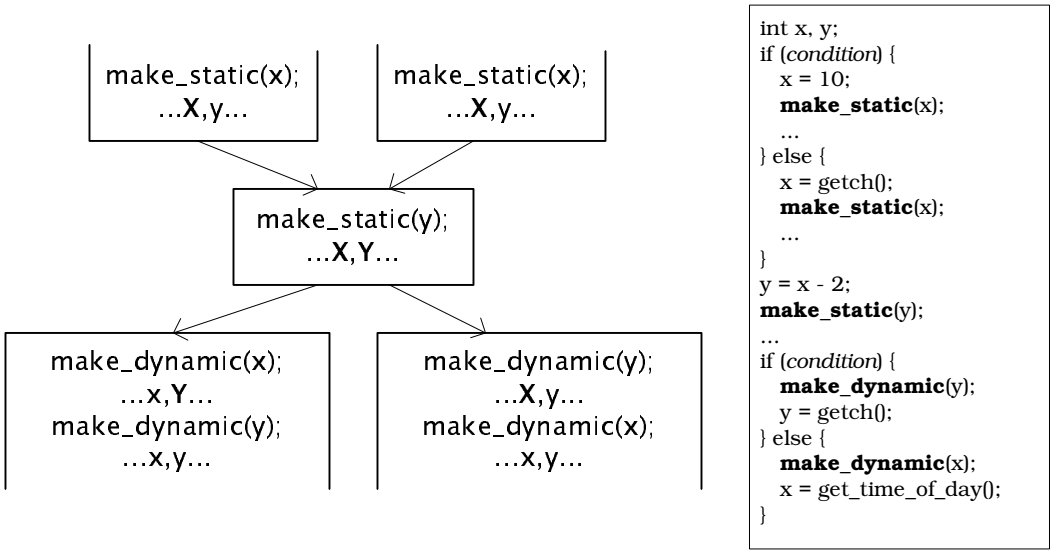

Figure 2-3: A graph fragment showing the flexibility of DyC in theflow of dynamic regions. The code on the right could have generated the different dynamic regions shown on the left.

specialization opportunities. The developer uses this language to construct specialization scenario modules that identify functions, global variables, or data structures that can be specialized, as well as classification of components as static or dynamic. These modules are compiled to produce declarations to guide Tempo's binding-time analysis. Tempo then performs a series of preprocessing transformations to produce *specializable code*. This specializable code can then be provided with the value of static data and processed to produce specialized code. Tempo can perform the processing either at compile-time or run-time. In addition to compile-time or run-time program specialization, Tempo provides the ability for data specialization.

Tempo and DyC are two examples of declarative partial evaluators that rely on user-provided annotations to guide partial evaluation. Compared with the imperative approach, declaration provides easier and safer partial evaluation, since annotations do not alter the behaviour of the program. The next subsection briefly talks about a third approach to dynamic code generation.

2.2.3 Automatic Code Generation

At the other extreme from imperative partial evaluation is automatic partial evaluation where the partial evaluator is not provided with any guidance from a user. These systems offer less control even than declarative systems, but also require less from a user.

Calpa [24] uses a combination of injected profiling and cost-benefit analysis to automatically insert DyC annotations. Calpa requires some profiled executions to run and be analysed before creating the annotations and is designed more to be run before execution than concurrently at runtime. Empirically, Calpa generated annotations performed as well as hand-crafted ones tuned over a number of weeks.

A recent project in automatic partial evaluation can be found in [28]. This system can use semantic constraints in Java code for use in static vs. dynamic classification; for instance, the `final` keyword specifies that values will be initialized once and never again modified. Additionaly, for data that cannot be guaranteed to be static, specialization can occur protected by run-time guards to invalidate specializations upon any changes. The system performs analysis on the byte-code to try and determine a key variable that can specialize many instructions. A specialization scope is then defined (similar to a dynamic region in DyC) and different versions are created optimized against different values of the key. This system performs analysis and specializes code automatically at the time of program execution.

This section presented three alternative approaches for dynamic code generation: imperative, declarative, and automatic. The next section introduces the notion of binary translation and discusses several different goals and systems.

2.3 Binary Translation

The term *binary translation* generically refers to translation from one binary format to another (possibly the same) binary format. There are many different uses for binary translation: one notable use is Just-In-Time (JIT) compilation of interpreted languages; another is architectural compatability; a third use is for optimization; and

a fourth is introspection (see [1] for a more thorough survey of binary translation goals and systems). The next subsections address each of these uses, with optimization and introspection combined into a single subsection.

2.3.1 Just-In-Time (JIT) compilation

A JIT compiler translates an intermediate byte-code representation to architecture dependent machine code at run-time. The purpose of a JIT is to improve performance by reducing interpretive overhead (see [26] for a comparison of techniques for increasing performance of executing Java programs).

JIT compilers have recently become extremely popular for Java Virtual Machines (JVMs). Since Java is widely used, any improvement in performance has drastic ramifications in the software industry. Furthermore, the conceptual design of a JVM has inherent performance penalties when run on any modern architecture; specifically, the JVM is a stack-based machine, but modern chips are register based. JITs such as Jalapeño [2] and LaTTe [32] can improve performance greatly by performing register allocation for sequences of Java bytecode, thus reducing the number of memory accesses. Also the translation of bytecodes can be cached, thus reducing the dynamic overhead for commonly executed sequences.

A similar approach is taken by Psyco [27] to partially evaluate Python code. Psyco introduces a notion of *representations*, and calls itself a just-in-time representation-based specializer for the Python language. Psyco includes a run-time dispatcher, and a specializer to iteratively generate machine code based on Python bytecodes as needed.

2.3.2 Architectural Compatibility

The goal of binary translation for architectural compatibility is to dynamically retarget machine code from one architecture to another, thus emulating the original architecture. Such emulation can be used to expand the base of usable applications for a different architecture or for taking advantage of hardware support not available when

the machine code was produced. IBM's DAISY [14] was created to support applications developed for the PowerPC RISC architecture on a VLIW machine. Similarly, Transmeta's Crusoe [22] was created to emulate Intel's x86 architecture on a VLIW. In a somewhat similary vein, DISCO [7], presents a single-processor, uniform-memory abstraction to commodity guest operating systems on a multi-processor NUMA machine.

2.3.3 Dynamic Optimization

Dynamic optimization systems are used to improve the performance of machine code, while introspection systems are used to examine and inspect running programs. Notice that these two uses are highly related and can be performed by a single system; this thesis will focus on the optimization aspect. The rest of this section describes some benefits of dynamic optimization and presents a situation that defies common heuristics.

Dynamic optimization systems have more data available to them (at run-time) than static (compile-time) optimization systems, and thus are better suited for certain optimizations. For instance, a static optimization system cannot cross any boundaries for dynamically loaded modules, while a dynamic system may examine (and optimize) any code executed. Also, a dynamic optimization system has knowledge of the particular processor on which the application is being executed and may apply architecture specific optimizations. For instance, the Intel Pentium 3 processor can execute `inc` and `dec` instructions faster than `add` or `sub` while the Pentium 4 can execute the `add` or `sub` faster. A static compiler would have to choose one, slowing the runtime on the other, while a dynamic system can translate at runtime from one to the other, allowing optimal running time on both.

The `DynamoRIO` system [6, 5] (explained in more detail in the next chapter) is a runtime optimization and introspection system. Its goal is to provide a view on running processes, allowing an external control to see (and possibly modify) every instruction before it is executed by the processor. Although `DynamoRIO` performs reasonably well on most classes of programs, interpreters in general defy the heuristics

it uses. `DynamoRIO-with-Log-PC` [30, 4] is an extension of `DynamoRIO` created to improve the performance of interpreters; towards this goal, `DynamoRIO-with-Log-PC` has developed an API for interpreter writers to transmit information for the purpose of optimization. Our work, `DynamoRIO-PE` provides an extension to `DynamoRIO` that subsumes the changes for `DynamoRIO-with-Log-PC`, again with a goal of performance improvements. The next chapter goes into more detail about `DynamoRIO` as the base for our work, and `DynamoRIO-with-Log-PC` as an alternative design.

Chapter 3

DynamoRIO and DynamoRIO-with-Log-PC

This chapter provides an overview of the operation of DynamoRIO and DynamoRIO--with-Log-PC and introduces some terminology and concepts that will be used throughout this thesis. We first begin with an overview of DynamoRIO's operations in emulating the execution of a client application, and some optimizations DynamoRIO performs to compensate for its overhead. We then discuss the modifications made by DynamoRIO-with-Log-PC.

3.1 DynamoRIO Implementation

The DynamoRIO system contains two main components: emulation and optimization. This section first presents the emulation component and then briefly describes the optimization component.

DynamoRIO takes a common approach to emulating a program: rather than simulating the processor running the program, DynamoRIO translates a basic block of machine code on demand and then lets that block run natively. A DynamoRIO basic block is defined as a sequential block of code with one entry point and ends with the first conditional or indirect jump instruction. Call instructions are walked into, replacing the call with pushing the return address and continuing translation at the

destintation. Any jump instruction that would normally exit a basic block is replaced with a jump to an *exit stub*. This exit stub is then responsible for returning control to DynamoRIO. To maintain program transparency, DynamoRIO monitors basic blocks to detect if code is self-modifying since such blocks must be handled specially.

Beyond the basic translation necessary for correctness, DynamoRIO can also perform optimizations on the translated blocks. Some optimizations performed by DynamoRIO include copy propagation, dead code elimination, and peephole optimizations. Since such optimizations are not guaranteed to improve performance (the time spent in optimization itself has to be compensated for), DynamoRIO refers to environment variables specifying which optimizations are to be performed, and how aggresively. The next section describes some DynamoRIO-specific techniques for reducing overhead and improving performance.

3.2 DynamoRIO Techniques

A major goal of DynamoRIO is to reduce the overhead added by emulation. Three main techniques are used to achieve this goal: caching, linking, and trace generation. These techniques are described further in the rest of this section.

To avoid translating the same basic blocks repeatedly, translations are *cached* in a code cache. When the emulated program is about to enter a new basic block, DynamoRIO checks to see if the target is in the cache before translating again. This reduces overhead greatly, but emulation across basic blocks must still return to the translator before continuing (essentially a context switch, to save and restore machine state).

To reduce the number of context switches, basic blocks are *linked* together in the code cache. That is, if a basic block ends with a direct jump to a basic block already in the code cache, DynamoRIO inserts a direct jump to the address of the target block in the code cache. Indirect jumps, however, cannot be changed as easily since the actual target is unknown. Instead, indirect jumps can be replaced with a test for the most common branch target; if the test succeeds, then a context switch can be

avoided by performing a direct jump to the expected block within the code cache. Otherwise, control must return to the translator.

Finally, to achieve better code layout DynamoRIO performs *trace generation*. DynamoRIO marks certain blocks as candidate *trace heads*; candidate trace heads are either 1) the target of an exit from a trace, or 2) the target of a direct backward jump. Trace heads are monitored to record how frequently they are targets. If a trace head is executed more times than a run-time evironment-defined threshold, then DynamoRIO starts recording a trace. Trace recording follows the current path of execution, stitching together basic blocks as it goes. If a conditional jump is taken, the condition is reversed and the non-taken jump target is changed to an exit stub. If an indirect jump is encountered, a comparison is embedded into the trace against the target at trace creation time. If during some later trace execution the indirect jump target does not match, execution will exit the trace and return to DynamoRIO's translator. Trace recording stops when any of the following conditions is met: 1) encountering another target that is marked as a trace head, 2) encountering a target that cannot be a trace head (such as self-modifying code), 3) if there is not room in the cache to expand the trace farther, or 4) the trace has reached a provided maximum number of basic blocks. Notice that trace generation is only helpful if the traces are executed frequently and for the length of the full trace; emulation of the trace must stop any time conditional checks or indirect targets differ from when the trace was recorded. We collectively refer to basic blocks and traces as *fragments*.

This section has described some techniques that DynamoRIO uses to improve performance of executed code and compensate for overhead in translations. Building on top of the structure described, DynamoRIO-with-Log-PC adds the ability for partial evaluation and functional memoization.

3.3 DynamoRIO-with-Log-PC Implementation

DynamoRIO-with-Log-PC was developed as a way to improve the performance of interpreters under DynamoRIO. DynamoRIO in essence tries to improve the performance

of an interpreter at the lowest level; if the focus instead is on improving the performance of an input program to an interperter then better performance can be achieved. Generally, interpreters follow a similar design scheme: instruction fetch and dispatch within a loop. In such a system, DynamoRIO will notice that the head of the loop is a frequent target and attempt to record a trace starting there. However, each different operation will be dispatched differently, and thus the trace will frequently not be fully executed. DynamoRIO-with-Log-PC was designed to address this shortcoming, focusing on two main features present in many interpreters. The first is an immutable input program, and the second a mutable index into the program acting as a logical program counter for interpretation. The interpreter writer provides annotations that make run-time calls into DynamoRIO-with-Log-PC to guide optimization.

To accomdate the first feature, an immutable input program, DynamoRIO-with--Log-PC provides the annotation dynamorio_set_region_immutable(*start, end*). This annotation marks the region of memory between start and end as unchanging for the life of the program. Any fixed reference of memory in that region can equivalently be replaced with a constant.

To accomodate the mutable logical program counter, DynamoRIO-with-Log-PC provides the notion of a *Logical PC* that can be changed with the provided annotation dynamorio_set_logical_pc(*log_pc*). Notice that the logical pc does not actually have to be represented in memory, or correspond to any concrete data in the program; instead it is used as an abstract key into the run-time state of the interpreter. Whenever a logical jump occurs (for instance, in interpreting a logical CALL or logical RETURN operation), a parallel call to either dynamorio_logical_direct_jump or dynamorio_logical_indirect_jump should take place. Internally, DynamoRIO--with-Log-PC uses the logical jump annotations to determine candidate trace heads. It does, however, use the same considerations for when to stop recording a trace as does DynamoRIO. DynamoRIO-with-Log-PC additionally tags each trace with a value for the logical pc. So for instance, the same program point can mark the state of multiple trace heads, each with a different value of the logical pc. If the logical pc corresponds to the execution path from that point then these different traces will be

executed fully, in general.

When execution reaches the point of a logical jump, `DynamoRIO-with-Log-PC` needs to perform a lookup to see if a trace already exists for the program point and the current value of the logical pc. If a trace does exist, then execution should be dispatched to the correct trace. If not, then the number of times the point has been reached with the current value of logical pc should be incremented and if it passes a threshold a trace should be recorded.

In addition to the annotations described earlier, there are annotations for marking variables as either trace constants (that is, the value is purely dependant on the value of the logical pc), or as unaliased. There is also an annotation for functions that have no side effects and can be safely memoized (see [23] for a more in-depth description of memoization).

This chapter described an overview of `DynamoRIO` and the modifications made for `DynamoRIO-with-Log-PC`. The next chapter presents some motivating examples for how partial evaluation under `DynamoRIO` can be better performed.

Chapter 4

Motivation for extending DynamoRIO

This chapter presents two motivating examples detailing places in which both DynamoRIO and DynamoRIO-with-Log-PC can be improved. The examples are two different implementations of the same minimal virtual machine, μ**VM**, one recursive and the other iterative. We begin this chapter with a definition and description of the μ-language, and an example program in the μ-language. We then analyze a recursive and an iterative implementation of a μ**VM**: For each implementation, we present example code for the `eval` function in the interpreter. We then walk through the traces generated by DynamoRIO on the example input program. We then present an annotated interpreter for use with DynamoRIO-with-Log-PC, and walk through the traces generated by DynamoRIO-with-Log-PC.

4.1 A Microscopic language and Virtual Machine (μVM)

The language targetted here is a primitive accumulator language; there is an accumulation register and a program counter register. A program consists of a sequence of instructions, where each instruction contains two bytes: an operation byte, and an argument byte. There are two different kinds of operations: `continue` and `end`. The `continue` instruction requests that the argument byte be added to the accumulator,

	0	1	2	3	4	5
instrs	c	2	c	3	e	4

Figure 4-1: An example program written in the micro language. Every even-numbered byte is an operation byte (either c(ontinue) or e(nd)), and every odd-numbered byte is an argument byte. The full program text is an array of bytes labelled `instrs`. Execution of the program shown here will result in the value `9` (2+3+4).

and execution continue with the next instruction. The `end` instruction requests that the argument byte be added to the accumulator and that execution cease. The argument byte is an integer value that is added to the accumulator. See Figure 4-1 for an example program in this language.

An interpreter for the μ-language described here has two intuitive implementations: recursive and iterative. The recursive implementation recursively evaluates the next instruction for a `continue` instruction, and then returns the sum of the current argument and the value returned. The iterative implementation contains a single sum variable, and loops through the input program adding each argument to the sum. Although these examples are conceptually simplistic, they illustrate how partial evaluation can help performance under `DynamoRIO`.

The next sections present concrete implementations of the different kinds of interpreters. Each interpreter's execution is simulated, and its performance analyzed under basic `DynamoRIO` and with annotations provided for `DynamoRIO-with-Log-PC`.

4.2 Recursive Implementation

This section presents a concrete recursive implementation of μ**VM** written in the C programming language. We then show how the interpreter could be annotated for use by `DynamoRIO-with-Log-PC`, and evaluate the performance.

For brevity in later examples, Figure 4-2 presents the structure for all μ**VM** interpreters we examine. The only difference will be in the implementation of the `eval` function. Notice the annotation declaring the `instrs` array as immutable; this annotation is only used when considering execution under `DynamoRIO-with-Log-PC`. We frequently refer to source code statements by corresponding line numbers.

When we do so, we use the notation *function*:*line_number*, where the line number is taken from the figure presenting the source code; when referring to a particular implementation of the `eval` function we disambiguate by appending a suffix of either i (iterative) or r (recursive) followed by either R (`DynamoRIO`) or L (`DynamoRIO-with--Log-PC`) (e.g., $\texttt{eval}_r R\texttt{:5}$ for line 5 of `eval` in the recursive implementation executed under `DynamoRIO`). In figures, we also reduce both `call` instructions and `return` instructions into their component instructions. Thus, a subroutine call is replaced with pushing a return address followed by a direct jumps (that can be inlined into a `DynamoRIO` block), whereas a return statement is replaced with popping the return address followed by an indirect jump to that location. (When presenting figures, we depict indirect jumps as jumping to the address `RIO_INDIRECT_DISPATCH` and show a dotted line to the run-time target.)

4.2.1 Recursive μVM under `DynamoRIO`

Figure 4-3 shows a straightforward recursive implementation of a μ**VM**. The evaluation function consists of four `DynamoRIO` basic blocks: fetching and dispatching the next instruction, executing an END_OP, and two blocks for executing a CONTINUE_OP, one performing a recursive call to `eval`, and the other performing the sum and returning. These are shown in Figure 4-4. In portraying the basic blocks we have included the starting application instruction pointer value.

The execution of μ**VM** defies `DynamoRIO`'s trace heuristics and furthermore `DynamoRIO` does not perform partial evaluation. `DynamoRIO` attempts to amortize the cost of building traces using the heuristic that a trace will be followed completely through; here, the head of the `eval` function will be recognized as a great trace head but execution will differ for the interpretation of each operation type[1]. Even more importantly, `DynamoRIO` does not take advantage of the fact that `instrs` is an immutable region of memory. Within a trace, it is possible to dereference the memory lookup and replace

[1]This isn't actually a big concern here, since each program is defined to have only one END_OP, and thus most operations will be CONTINUE_OPs. However, a slight modification of μ**VM** (e.g. the addition of a subtract operation) will effect the described results.

```
byte *instrs; /* array of program instructions */                    1
int eval(int pc); /* prototype for eval function */                  2
                                                                      3
int main(int argc, char *argv[]) {                                    4
  int length;                                                         5
  length = read_instructions(&instrs, argc, argv);                    6
  DECLARE_IMMUTABLE_REGION(instrs, length*sizeof(byte));              7
  return eval(0); /* return value is obtained by evaluating start of program */  8
}                                                                     9
```

Figure 4-2: The common structure around the different implementations of μ**VM**. The annotation establishing `instrs` as an immutable region is removed when not compiling under `DynamoRIO-with-Log-PC`

```
int eval(int pc) { /* returns sum of args, starting at pc */         1
  int op;                                                             2
  op = instrs[pc];                                                    3
                                                                      4
  if (op == END_OP) { /*return with value of next arg */             5
    return instrs[pc+1];                                              6
  } else if (op == CONTINUE_OP) { /*add value of this arg to rest */  7
    int result;                                                       8
    result = eval(pc+2); /*recursive call on next instruction */     9
    return instrs[pc+1] + result;                                     10
  }                                                                   11
}                                                                     12
```

Figure 4-3: Example code for a recursive implementation of μ**VM**'s `eval`$_r R$ function.

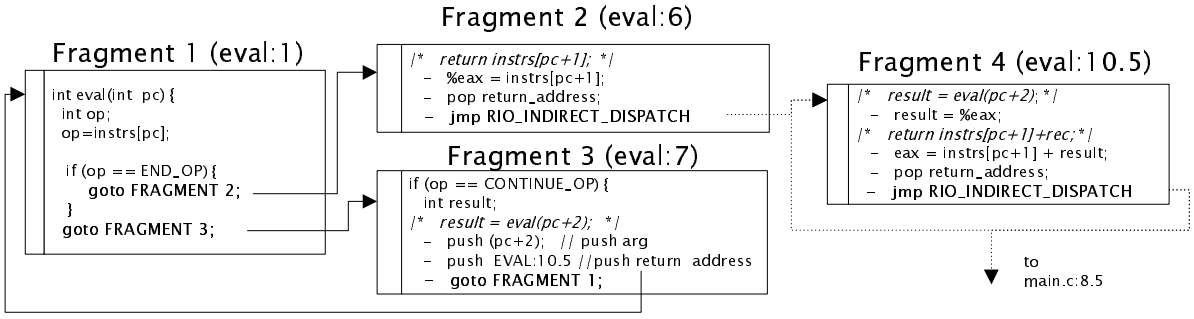

Figure 4-4: The generated Basic Block fragments for `eval`$_r R$ when run under `DynamoRIO`.

it with a constant, then perform constant propagation/folding.

4.2.2 Recursive μVM under DynamoRIO-with-Log-PC

DynamoRIO-with-Log-PC solves both of the issues discussed regarding DynamoRIO. Each time the head of the eval function is reached the *logical pc* determines the code executed in the body. DynamoRIO generates at most a single trace starting at the head of the eval function containing only a single execution of the body; thus, the only time the trace will be beneficial is when control flow follows the same path as that at the time of trace creation (i.e. only when the interpreted operation is the same). DynamoRIO-with-Log-PC augments this with a trace for each value of pc, improving the chance that the entire trace will be executed. Additionally, by specializing traces against the value of pc, DynamoRIO-with-Log-PC is able to fold constant references and remove some conditional jumps within a trace.

Example Traces Generated under DynamoRIO-with-Log-PC

To further explain some of the limitations of DynamoRIO-with-Log-PC we now work through the example code (Figure 4-5) on the example input in Figure 4-1.

Figure 4-5 presents a first attempt at annotating the recursive μ**VM** with DynamoRIO-with-Log-PC annotations, and Figure 4-6 presents the corresponding traces generated when running on the example in Figure 4-1 (assuming either a hot_threshold of 1, or that the main program is in a loop). The first trace begins with the first execution of eval(0). In DynamoRIO-with-Log-PC, traces may only begin after a logical jump, here the DIRECT_JUMP on line $\texttt{eval}_r L\texttt{:4}$ of the code fragment. The first trace then is annotated specifying that whenever the logical pc value is 0, and the native pc value is $\texttt{eval}_r L\texttt{:5}$, the value of the variable pc will always be the same as it is on this trace (namely, the constant 0 since pc is the concrete realization of the logical pc). These assumptions allow the memory address calculation in line $\texttt{eval}_r L\texttt{:6}$ to be held fixed as a constant as well. Since the memory address is fixed and the memory region is immutable, the entire statement is replaced by moving a constant value into

```
int eval(int pc) { /* returns sum of args, starting at pc */                1
  int op;                                                                    2
  SET_LOGICAL_PC(pc);                                                        3
  DIRECT_JUMP();                                                             4
  TRACE_CONSTANT_STACK(pc);                                                  5
  op = instrs[pc];                                                           6
                                                                             7
  if (op == END_OP) { /* return with value of next arg */                   8
    return instrs[pc+1];                                                     9
  } else if (op == CONTINUE_OP) { /* add value of this arg to rest */        10
    int result;                                                             11
    result = eval(pc+2);                                                    12
    return instrs[pc+1] + result;                                           13
  }                                                                         14
}                                                                           15
```

Figure 4-5: Example code for a recursive implementation of μ**VM**'s `eval` function, annotated in the style used in `DynamoRIO-with-Log-PC`.

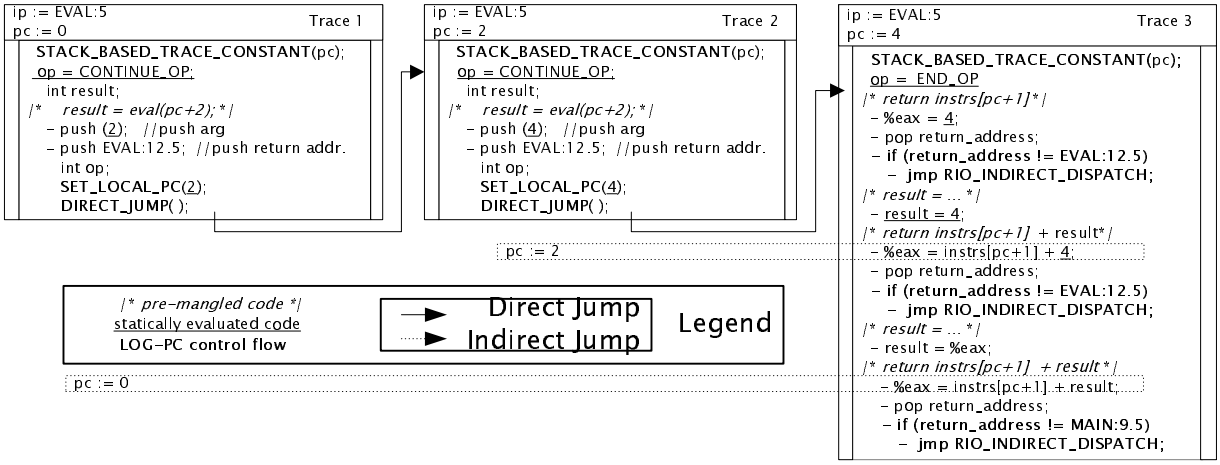

Figure 4-6: Example of traces created under `DynamoRIO-with-Log-PC` on source code shown in Figure 4-5. The legend notes convention we follow through our figures: underlined text has been specialized, *italic* text represents C code that has been mangled, **bold** text represents `DynamoRIO`-inserted control flow. Trace assumptions that `DynamoRIO-with-Log-PC` makes are represented in the box above the trace code, where `ip` represents the architecture native instruction pointer (register %eip), and `pc` represents the abstract logical program counter.

`op` (`op = CONTINUE_OP;`). Now that `op` is a constant as well, the conditional check at line $\texttt{eval}_r L\texttt{:8}$ can be resolved to `false`, and the whole block can be removed. Additionally, the check at line $\texttt{eval}_r L\texttt{:10}$ can be resolved to `true` and the code executed without the check. Execution now follows to line $\texttt{eval}_r L\texttt{:11}$ and then the evaluation of the next instruction at `instrs[2]`. `DynamoRIO` mangles the recursive call to `eval` into pushing the arguments on the stack, followed by pushing the return address and execution of the code at the target ($\texttt{eval}_r L\texttt{:0}$). The first instruction in the recursive call sets the logical pc and the next performs a direct logical jump. (Thus, Trace 1 can be directly linked to Trace 2). The next trace, Trace 2, has the same considerations and is treated similarly.

The third trace (and third interpreted instruction) contains the base case to exit the recursion, and is thus different from the first two. Here, the current `op` is found to be an END_OP and thus the conditional check at line $\texttt{eval}_r L\texttt{:8}$ resolves to `true` now. Thus, the return statement on line $\texttt{eval}_r L\texttt{:9}$ is to be executed. `DynamoRIO` has mangled this call into storing the return value in the return register (`%eax`), followed by an inlined runtime check against the return address recorded at trace creation time. If the check fails, then the trace is exited and execution makes its way to `RIO_INDIRECT_DISPATCH`. Otherwise, execution continues in the trace. Here the return value is the trace constant `instrs[pc+1]` which is 4. Following the return instruction are the latter parts of the first two calls to `eval`, including subsequent runtime guards on the return addresses.

A careful study of the traces generated reveals that the first `eval` execution is interrupted midway through to start another trace corresponding to the second execution of `eval`. The second is likewise interrupted with the third execution. However, the third trace actually contains not only the entire third execution of `eval`, but also the latter half of the first and second executions. Having the third trace contain these sections of code is counterintuitive and runs contrary to many goals of `DynamoRIO-with-Log-PC`; the likelihood of execution staying in the trace diminishes, and optimizations on the contained code are impossible since they are classified under the wrong logical pc.

Second Attempt at Recursive μVM under `DynamoRIO-with-Log-PC`

After analyzing the traces and encountering the difficulties mentioned, one could add annotations following the return instruction to start a new trace. The modified annotations are shown in Figure 4-7 and corresponding traces are presented in Figure 4-8. Some salient traits are that the expected traces are generated and optimized as we had wanted. One interesting note is that the conditional execution in Figure 4-6 has been replaced with both conditional execution and indirect jumps in Figure 4-8. Notice that these indirect jumps are much more costly than the conditional execution inlined in the earlier traces. Also notice the amount of annotation necessary for better optimization; not only is the annotation cumbersome for the user, but each annotation adds at least a function call overhead until the call is removed in a trace. This means that if a trace is not created, then this overhead is paid on every execution; it also adds more initial overhead that must be compensated for. More importantly, the second round of annotation requires too much expertise and knowledge of `DynamoRIO-with-Log-PC`'s inner workings.

These are some issues in `DynamoRIO-with-Log-PC` stemming just from the minimal recursive μ**VM**. The next section discusses an iterative version, detailing the problems found there. After more problems are related in the iterative case, the next chapter presents a modified `DynamoRIO` system and shows how it solves the problems mentioned in this chapter.

4.3 Iterative Implementation

This section provides an iterative implementation of a μ**VM** interpreter and analyses its performance natively under `DynamoRIO`, and with annotations under `DynamoRIO-with-Log-PC`.

The purpose of analysing the iterative implementation is to show how loops are handled in `DynamoRIO` and `DynamoRIO-with-Log-PC`. There is a tradeoff between code blowup and the possibilities for specialization: loop unrolling is nice because it allows each iteration to be specialized, and reduces the number of jumps; however,

```
int eval(int pc) { /* returns sum of args, starting at pc */                    1
  int op;                                                                        2
  SET_LOGICAL_PC(pc);                                                            3
  DIRECT_JUMP();                                                                 4
  TRACE_CONSTANT_STACK(pc);                                                      5
  op = instrs[pc];                                                               6
                                                                                 7
  if (op == END_OP) { /* return with value of next arg */                       8
    return instrs[pc+1];                                                         9
  } else if (op == CONTINUE_OP) { /* add value of this arg to rest */            10
    int result;                                                                  11
    result = eval(pc+2);                                                         12
    SET_LOGICAL_PC(pc);                                                          13
    INDIRECT_JUMP();                                                             14
    TRACE_CONSTANT_STACK(pc);                                                    15
    TRACE_CONSTANT_STACK(result);                                                16
    return instrs[pc+1] + result;                                                17
  }                                                                              18
}                                                                                19
```

Figure 4-7: Example code for a recursive implementation of μ**VM**'s `eval` function, annotated in the style used in `DynamoRIO-with-Log-PC`(second attempt).

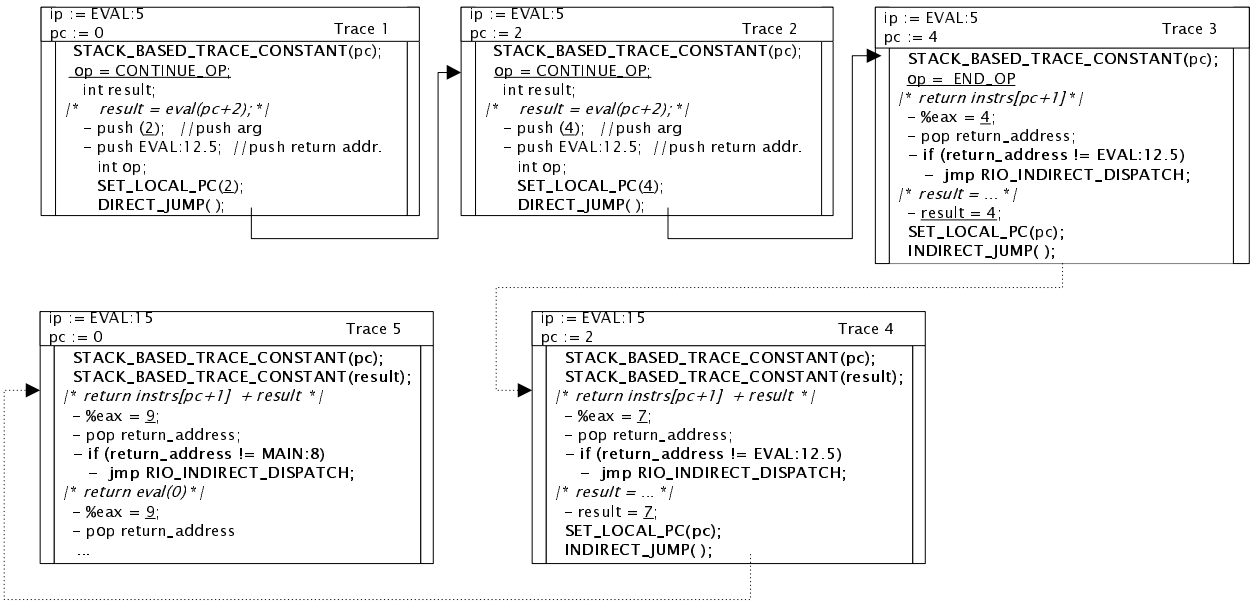

Figure 4-8: Example of traces created under `DynamoRIO-with-Log-PC` on re-annotated source code shown in Figure 4-8. We use the same convention as in previous figures.

```
int eval(int pc) { /* returns sum of args, starting at pc */                1
  int sum; /* Accumulation register */                                       2
  int op;                                                                    3
  sum = 0;                                                                   4
  while (END_OP != op) {                                                     5
    op = instrs[pc];                                                         6
    switch (op) { /* dispatch on op-code */                                 7
    case CONTINUE_OP:                                                        8
      sum += instrs[pc+1];                                                   9
      break;                                                                 10
    }                                                                        11
    pc+=2; /* update pc */                                                   12
  }                                                                          13
  sum += instrs[pc+1];                                                       14
  return sum;                                                                15
}                                                                            16
```

Figure 4-9: Example code for a iterative implementation of μ**VM**'s `eval` function.

if loops are unrolled without limit then a single loop can expand to fill the cache. Additionally, `DynamoRIO` generates a substantial amount of code for loop unrolling: each possible exit from the loop requires its own trace exit stub. We address a parallel remark concerning recursion at the end of this chapter.

Each system uses its own extreme policy for handling loops, and thus both perform suboptimally. In basic `DynamoRIO` loops are never unrolled in traces. This conserves room in the cache, at the cost of not allowing traces to contain any loops (traces are defined as single entrance, multiple exit, and so a jump cannot be performed into the middle of a trace). Alternatively, `DynamoRIO-with-Log-PC` unrolls however many times the loop is executed when that trace is generated[2]. This allows `DynamoRIO--with-Log-PC` to optimize what it can during these loop iterations, but also bloats the code.

[2]Actually, one can limit the maximum size of a trace. In `DynamoRIO-with-Log-PC` then, when a loop is unrolled enough to expand the trace size to the upper bound, the trace will end and unrolling will stop. In this case, no optimizations are performed after the trace ends, until the next time a logical jump occurs.

```
int eval(int pc) { /* returns sum of args, starting at pc */                    1
  int sum; /* Accumulation register */                                          2
  int op;                                                                       3
  SET_LOCAL_PC(pc);                                                             4
  DIRECT_JUMP();                                                                5
  TRACE_CONSTANT_STACK(pc);                                                     6
  sum = 0;                                                                      7
  while (END_OP != op) {                                                        8
    op = instrs[pc];                                                            9
    switch (op) { /* dispatch on op-code */                                     10
    case CONTINUE_OP:                                                           11
      sum += instrs[pc+1];                                                      12
      break;                                                                    13
    }                                                                           14
    pc+=2; /* update pc */                                                      15
  }                                                                             16
  sum += instrs[pc+1];                                                          17
  return sum;                                                                   18
}                                                                               19
```

Figure 4-10: Example code for an iterative implementation of μ**VM**'s `eval` function, annotated in the style used in `DynamoRIO-with-Log-PC`.

4.3.1 Iterative μVM under `DynamoRIO`

Figure 4-9 presents the code for the interpreter to be run under `DynamoRIO`. `DynamoRIO` will create a trace starting at the head of the `while` loop at line `eval`$_i$`R:4`. This trace will only contain one iteration of the loop (and the subsequent direct jump back to the loop-head). As mentioned in the previous section, only one path through the loop will be recorded (that for a CONTINUE_OP). Also, again, no partial evaluation is performed under `DynamoRIO`.

4.3.2 Iterative μVM under `DynamoRIO-with-Log-PC`

We next present annotated code for the iterative μ**VM** in Figure 4-10. Notice that we had a choice in whether to place the annotations before the loop (at `eval`$_i$`R:3`), or inside (at `eval`$_i$`R:7`). We discuss a few considerations about placement later. Placing the logical jump annotation before the loop causes the recorded trace to completely unroll the loop (until reaching the trace-size limit). In the example shown, this is

actually beneficial, as all dispatch in the loop can be removed, all conditional exits before the final iteration can be removed and all that remains is a list of `sum += const` instructions. If any more instructions were dynamic, the trace size would grow quickly - especially if the exit condition were not statically computable.

Placing the annotations inside the loop would result in two reasons for worse performance: one obvious, the other not so much. First, many more traces would be created, one for each iteration. Asymptotically this just results in executing a hardware jump for each iteration (once the traces are all directly linked); as well as a lot of lengthy exit stubs and bookkeeping overhead for `DynamoRIO-with-Log-PC`. Second, however, is the fact that at the assembly level, the compiler has probably loaded a register with the value of the base address for the `instrs` array; if so, then `DynamoRIO-with-Log-PC` may not be able to recognize the opportunity for constant folding.

4.4 Summary of Motivation

We now summarize issues we introduced across the various implementations of μ**VM** under `DynamoRIO-with-Log-PC`, since the main issue with `DynamoRIO` is its lack of partial evaluation.

Cumbersome Annotations Requirements Notice the cumbersome annotations required for correctness and optimization. The second attempt definition of $\texttt{eval}_r L$ is twice the size of $\texttt{eval}_r R$. Furthermore, the annotations for better optimization required extensive knowledge of internal `DynamoRIO-with-Log--PC` structure.

Issues with Linking In the second definition of $\texttt{eval}_r L$, the final two traces are not able to be linked directly. Indirect links force re-entrance into the `DynamoRIO` subsystem for dispatch at extensive cost. One is forced to choose between lack of optimizations in the first version against indirect jumps in the second.

Extreme Choices for Code Growth The underlying `DynamoRIO` system performs

no loop unrolling in its traces, while `DynamoRIO-with-Log-PC` has no ability to stop. Furthermore, if one were able to stop `DynamoRIO-with-Log-PC` from unrolling loops, there is no way to reestablish all the trace invariants discovered before stopping loop unrolling (one can provide annotations for named memory locations, but not for derived or temporary locations). This issue is addressed next.

Propagating Information During the course of optimizing a trace, `DynamoRIO-with-Log-PC` is able to infer the values of some variables based on the incoming assumptions and the TRACE_CONSTANT annotations (such as `op` in $\texttt{eval}_r L\texttt{:6}$). However, unless the interpreter writer annotates it, this information is lost if a new trace should begin execution. For an example of this, examine the new trace head declared at the second definition of $\texttt{eval}_r L\texttt{:14}$. When returning from the recursive call to $\texttt{eval}_r L$ at line $\texttt{eval}_r L\texttt{:12}$, the value of `op` is the same as before the call; however, the new trace does not take advantage of this fact. Not only does this impose an added burden on the annotator, temporary values or other values not easily referenced cannot be annotated and are thus lost. If the information is propagated to the new trace, then better optimizations can be performed.

Limited to a Single Dispatch Variable Although the examples shown did not exhibit any need for it, a more general system could handle *polyvariant division*. This would allow the same execution point to be specialized against the value of more than a single variable to exploit more static knowledge.

Abundance of Specialized Traces As alluded to in the iterative section, a single static program point annotated as a logical jump may be specialized against any number of values for logical pc. If there are too many the code growth could be substantial, adversely affecting performance. A logical jump at the head of a loop, or the head of a resursive function, such as $\texttt{eval}_r L$, are likely candidates for causing such behaviour.

This chapter presented the μ-language and analysed the performance of both a recursive and iterative version, each under `DynamoRIO` and `DynamoRIO-with-Log-PC`. The next chapter presents the features and characteristics of a partial evaluation system designed to address these restrictions.

Chapter 5

Proposed extensions to DynamoRIO

This chapter proposes an alternative extension to DynamoRIO than DynamoRIO-with--Log-PC, DynamoRIO-PE. We then show how the extension is able to better handle both the recursive and iterative μ**VM** implementations introduced in the preceding chapter. Finally, we present a few alternative designs and reasons to not use them.

5.1 Generalized DynamoRIO with DynamoRIO-PE

This section describes the features and annotations necessary to correct the shortcomings summarized in the previous section. We begin by describing the annotation differences and the translation from DynamoRIO-with-Log-PC to the new annotation style, known as DynamoRIO-PE.

We maintain the annotations of immutable memory regions introduced in DynamoRIO-with-Log-PC. However, instead of singling out one value to specialize a trace against (the logical pc), we let annotators specify any number and type of *Locations of Interest* (LoI). An LoI, is simply a memory location (either absolute address or stack-based offset) that the annotator has determined useful to specialize against: this includes locations that affect control flow, locations that assume only a fixed set of values, etc. These LoIs are treated as static when optimizing a recorded trace.

The annotator specifies relevant LoIs at a *Point of Interest* (**PoI**). A **PoI** is a place in execution where a new set of LoIs take effect. A **PoI** is similar to the beginning

DynamoRIO-with-LogPC	→	POI/LOI notation
SET_LOGICAL_PC(val)	→	logpc = val
LOGICAL_DIRECT_JUMP()	→	POINT_OF_INTEREST(LOI)
LOGICAL_INDIRECT_JUMP()	→	POINT_OF_INTEREST(LOI)
TRACE_CONSTANT(&addr, val)	→	LOI = (LOI . ¡addr¿)
UNALIASED_MEMORY(&addr)	→	LOI = (LOI . ¡addr¿)

Table 5.1: A table of translations from annotations in `DynamoRIO-with-Log-PC` to annotations in `DynamoRIO-PE`. The left column presents the original annotations in `DynamoRIO-with-Log-PC`, while the right shows how to represent the same information in `DynamoRIO-PE`. Notice that the number of total annotations is reduced to a single annotation declaring all LoIs.

of a scope, or a dynamic region; a **PoI** also marks the end of the previously executed **PoI**. This command subsumes both the SET_LOG_PC call and the subsequent LOGICAL_DIRECT_JUMP or LOGICAL_INDIRECT_JUMP call. It also incorporates SET_TRACE_CONSTANT calls through either direct inclusion in the list of LoIs or through discovering more LoIs as the LoIs are propagated. For a full translation of annotations provided by `DynamoRIO-with-Log-PC`, see Table 5.1. The signature for **PoI** is:

void POINT_OF_INTEREST(**int** stack_count, **int** *stack_addr$_0$, ..., **int** *stack_addr$_{stack_count-1}$, **int** global_count, **int** *global_addr$_0$, ..., **int** *global_addr$_{global_count}$);

We call a collection of LoIs an *optimization template*, and the mapping of all LoIs to values an *optimization context* (**OC**). Every program point in a trace has an associated **OC**, defined as follows. The head of the trace has an **OC** determined by the optimization template presented at the **PoI** call point. A subsequent instruction `instr`'s **OC** is based on the **OC** of the previous instruction, modified by the results of `instr`. The results of an instruction could: add a new LoI if all instruction sources are LoI and the destination is not; modify an LoI if all instruction sources and the destination are LoIs; or remove an LoI if the instruction destination was considered an LoI, but the sources are not.

Although `DynamoRIO-with-Log-PC` is able to intuitively define the notions of direct or indirect logical jumps, `DynamoRIO-PE` has to more carefully extend the notions to multiple LoIs. We say that a transition from one **PoI** to another is *direct* if the

OC at the destination **PoI**'s start is contained in the **OC** of the instruction making the transition; otherwise the transition is *indirect* (i.e. going from a specific context to a more generalized context is a direct jump). We show an example of this criterion in the next section.

Internally, `DynamoRIO-PE` uses the **PoI** points to mark trace heads and trace exits in a manner that parallels `DynamoRIO-with-Log-PC`'s handling of logical jumps. When `DynamoRIO-PE` encounters a **PoI** it dispatches to a trace specialized against the current values of all LoIs. If no such trace exists, and the number of attempts reaches a threshold, then a new trace will be recorded and specialized against the current **OC**. Furthermore, `DynamoRIO-PE` can choose to supplement the provided LoIs in a **PoI** with any other LoIs it may happen to know of at that program point. Finally, `DynamoRIO-PE` also has the ability to mark any instruction that begins a basic block as a **PoI** with any LoI that are known about at that point.

There are two major changes in this new system: a trace can be selected and specialized based on any number of trace parameters, and there are only two annotations. With these two main changes, we can overcome the previously reported deficiencies with `DynamoRIO` and `DynamoRIO-with-Log-PC`. The next section shows how the features described here address the problems with recursion that we summarized in the previous chapter.

5.2 Example 1: Recursive μVM

Figure 5-1 presents the source code for the recursive μ**VM** annotated in the new style. Figure 5-2 then shows traces generated under the proposed `DynamoRIO-PE` system.

We now step through Figure 5-2, explaining how and why each trace is generated. As in $\texttt{eval}_r L$, all conditional code is resolved at the time of partial evaluation.

Suppose the trace threshold for the **PoI** in `eval` has been reached and `eval` has been called with `pc` = 0. The first trace begins after the first call to **PoI**, where `pc` is declared as a LoI. (Thus, the **OC** for this trace consists of `ip` := $\texttt{eval}_r P\texttt{:4}$, and `pc` := 0). Since `pc` is a LoI and the `instrs` array has been declared immutable,

```
int eval(int pc) { /* returns sum of args, starting at pc */                    1
  int op;                                                                        2
  POINT_OF_INTEREST(1, pc, 0);                                                   3
  op = instrs[pc];                                                               4
                                                                                 5
  if (op == END_OP) { /*return with value of next arg */                        6
    return instrs[pc+1];                                                         7
  } else if (op == CONTINUE_OP) { /*add value of this arg to rest */             8
    int result;                                                                  9
    result = eval(pc+2); /*recursive call on next instruction */                10
    return instrs[pc+1] + result;                                                11
  }                                                                              12
}                                                                                13
```

Figure 5-1: Example code for a recursive implementation of μ**VM**, annotated in the style suggested for use in `DynamoRIO-PE`.

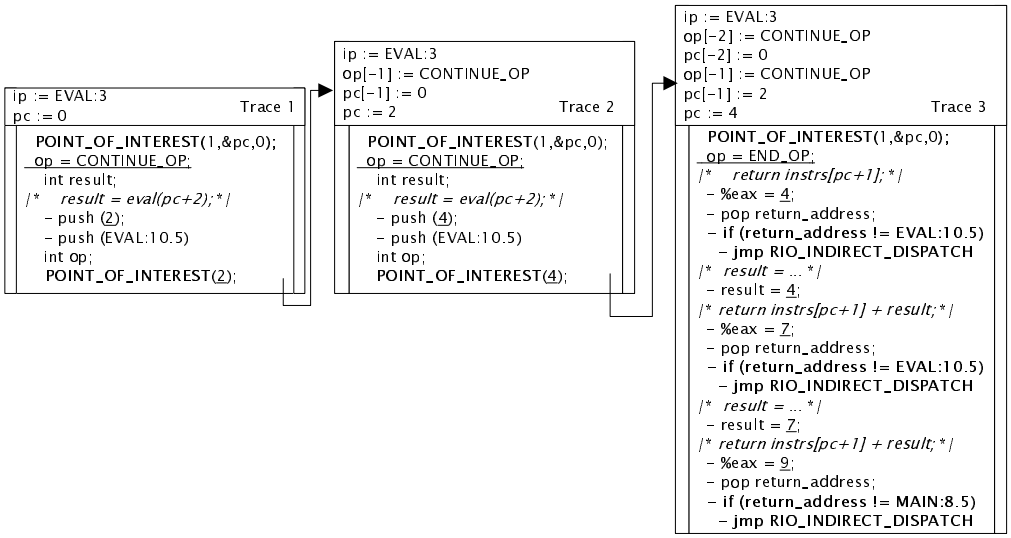

Figure 5-2: Example of traces created under `DynamoRIO-PE` on source code shown in Figure 5-1. Trace assumptions that `DynamoRIO-PE` makes are represented in the box above the trace code, where `ip` represents the architecture native instruction pointer, and any other parameters were provided through annotation at a **PoI**. Variables with a `[n]` imply that the variable is defined in a scope `n` frames away from the current one. Solid lines represent direct links from one trace to another.

`instrs[pc]` can be folded to a trace constant. Since `op` is assigned a trace constant in line $\texttt{eval}_r P\texttt{:4}$, `op` too becomes a LoI. The following conditional checks are then resolved as constants, and `eval(2)` is executed. We then encounter the next **PoI**; since the new **OC** is known (because `pc` is a trace constant of the first trace), the control transfer is direct, and therefore the traces can be linked directly. We also show here the ability of `DynamoRIO-PE` to include known LoI at the end of trace 1 into the **OC** for trace 2. The LoIs included are the values of `pc` and `op` in the previous stack frame. Similarly we are able to directly link from the second to the third trace, and add more LoIs to trace 3.

The third trace has an extensive list of LoI at its head, and immediately adds the `op` field in the current frame. It then has to emulate the return instruction, and thus compares the return address against the value at the time of trace recording. The return statement also regresses a stack frame, and so now the newest value of `pc` (4) is invalid and the previous `op` (with value `CONTINUE_OP`) and `pc` (with value 2) are valid. Since the current value of `pc` is known, partial evaluation can continue, replacing the expression `instrs[pc+1] + result` with the constant value `7`. Another return occurs (again regressing a stack frame), and the final result of `9` is known. Notice that the trace cannot remove the multiple assignments of `results` or `%eax`, since execution could leave the trace at emulation of `return` instructions. It is possible to include the return address as another LoI, thus enabling those runtime checks to be removed from trace 3 and the multiple assignments eliminated as dead code. This is another example of the tradeoff between more information leading to better optimizations but also less generality or applicability, since we are adding more trace requirements.

The traces generated with $\texttt{eval}_r P$ mirror those in the first definition of $\texttt{eval}_r L$. If we included another **PoI** annotation after the return from $\texttt{eval}_r P$ at line $\texttt{eval}_r P\texttt{:11}$ (as we did for $\texttt{eval}_r L$), we could again split the final trace into 3 different traces. This time, however, the links would be direct, since the requirements at the head of trace 3 are restrictive enough to guarantee correctness. The next section presents the method for handling loops in `DynamoRIO-PE`.

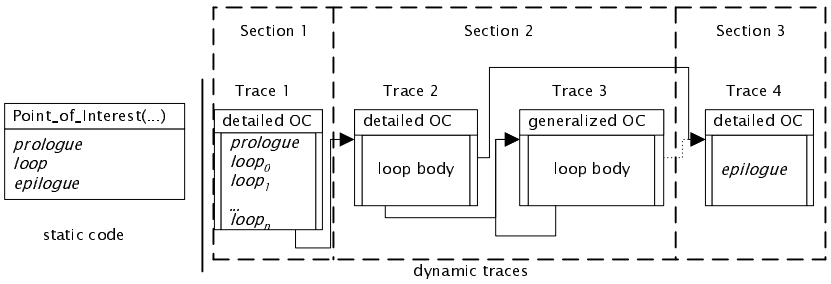

Figure 5-3: Figure detailing different run-time traces generated for code that contains a loop. The first generated trace contains the prologue (part between the call to **PoI** and the loop head), and the first n iterations of the loop (where n is the run-time parameter for number of times to unroll a loop). The second generated trace contains a single iteration of the loop specialized against the same optimization context. The third generated trace is also a single iteration of the loop, but is specialized against a more general optimization context. The OC has been generalized enough that the end of the trace is able to jump back to its own head. The final trace contains the epilogue and is specialized against a more detailed optimization context propagated from the first trace.

5.3 Example 2: Iterative μVM

5.3.1 Handling Loops

This section describes the mechanisms necessary to handle loops well. The goal is to provide a limit to code growth caused by loop unrolling without sacrificing knowledge. The general idea is to split a loop-containing trace into three sections: the first section contains the instructions until the start of the loop, as well as a limited number of loop iterations. The second section contains the body of a single iteration of the loop. If this second trace loops back to the instruction marking its head, a third trace is created. This third trace also contains just a single loop iteration, although with a more generalized trace so that the last transition can be made directly to its head. The third section is the final trace containing the epilogue. Figure 5-3 displays the three sections and generated traces as described, for an abstract loop. The rest of this section describes each trace in detail.

Before the loop

The first trace is created using the mechanism described earlier: the trace begins recording at the point of the annotation to **PoI** prior to the prologue. Once a loop is detected to have occurred *max_loop_unroll* times, trace generation for the first trace stops.

After the trace is ended it is optimized, and a final **OC** is generated. Here we encounter a general dilemma: a new trace is going to be created; should the current **OC** be propagated to the new trace, or not? Propagation is useful if the parameters occur frequently, and enables partial evaluation. However, unnecessary specifications reduce the applicability of traces. `DynamoRIO-PE` associates the next basic block (the head of the loop) with the resultant optimization template. To force recording a new trace on the subsequent instruction, `DynamoRIO-PE` can set the count to be the value of *hot_threshold*.

The loop body

When `DynamoRIO-PE` encounters the basic block at the head of the loop, it finds an associated optimization template and an execution count that has reached the threshold; thus, a new trace is recorded. If this trace includes a direct jump back to the head of the trace, then it contains a single iteration of the loop and so `DynamoRIO-PE` stops recording. The conditional exit from the loop is marked with an optimization template based on the **OC** of the conditional loop exit instruction. This will be used to create the final trace.

The goal of the loop body trace is to have general enough requirements that the final backwards jump can be a direct link to itself. Thus, when optimizing a loop body any LoI whose value changes within the trace body is removed from the final **OC**. At the end of the trace, only LoIs that have remained unchanged are left in the **OC**. If the **OC** is the same as that at the head, `DynamoRIO-PE` can immediately make a direct link; otherwise, `DynamoRIO-PE` needs to record another loop iteration with the more general **OC**. If so, we then begin recording a new trace headed by the

final **OC** from the previous trace. This trace stops recording at a backwards jump. We repeat the process of testing for a compatible direct link, and recording a new trace if the end **OC** and head **OC** are not compatible. This process is guaranteed to eventually terminate, since an empty **OC** would meet the criterion; also, due to the dynamic nature of `DynamoRIO-PE`, we will stop trying to record a general loop body once the application exits the loop.

`DynamoRIO-PE` will continue dynamically executing the generalized loop body until the program exits the loop. At this point `DynamoRIO-PE` begins recording the final trace.

After the Loop

Once the application exits from the loop, `DynamoRIO-PE` will begin recording a new trace beginning at the epilogue, specialized against an **OC** based on the optimization template provided from the first trace. Note that the values of the LoIs may have changed since the first trace, and so the new trace should not be optimized against those values, just the locations. Notice also that if the loop body trace had to be generalized for correctness, its exit cannot be a direct link to the new trace. Instead, `DynamoRIO-PE` will have to first guarantee that the values of all LoIs match, or will have to record a new trace specialized against different values.

Other Possible Loop Behaviors

The scenario described above is what occurs when the number of iterations for the loop is greater than the *max_loop_unroll* runtime variable. If the number of iterations is less than *max_loop_unroll*, then only one trace, with the loop unwound, will be recorded.

5.4 Alternatives to `DynamoRIO-PE`

This section presents some alternative designs to `DynamoRIO-PE` that could have solved the motivating problems.

Rather than unconditionally breaking traces at logical jump points, we could have modified `DynamoRIO-with-Log-PC` to heuristically keep longer traces. This solves the problem of propagating information to the return site, since the return site will be included in the enormous trace. It also solves any issues with linking, since there will be no need to link across traces. The two main drawbacks with this approach are related to each other: code bloat and loss of generality. The suggested approach will have huge traces that might have a very low chance of being executed the whole way through. [4] relates that long traces are a performance problem in `DynamoRIO-with--Log-PC`; this approach would create more traces that are long. Additionally, if a new trace includes the entire `eval(0)` $\downarrow$ `eval(2)` $\downarrow$ `eval(4)` $\uparrow$ `eval(2)` $\uparrow$ `eval(0)` sequence, there will be no trace containing just `eval(2)` in the cache. Thus, there will necessarily be duplicated code in the cache requiring duplicate efforts at specialization. And finally, this approach contains an additional knob to tweak (when to break traces). The best setting for this parameter is dependant on the executing program, and so would either need an externally modifiable policy, or require the annotator to add hints specific to this choice.

The approach we suggest here is really more of an extension to `DynamoRIO` than to `DynamoRIO-with-Log-PC`; we could have kept the same interface with `DynamoRIO--with-Log-PC`, and simply modified the system internally. We could have used the same concept of a controlling logical pc, and simply added the **OC** to the internal traces generated. (Adding the **OC** is necessary to propagate information, something desired for both the recursive and iterative μ**VM**). This would have presented the same interface to the annotator, but achieved some benefits internally. The reason we decided to revamp the API is that we thought it too restrictive and too complicated. We believe that the `DynamoRIO-PE` system lends itself more readily to a wider class of program than interpreters, the target of `DynamoRIO-with-Log-PC`. We also feel that the same modifications would have to be made internally whether we kept the same API or not, and so it makes sense to externalize the generalized capabilities.

One additional option we explored was to have a `DynamoRIO`-maintained return stack. In the recursive μ**VM** case, the action we wanted is to return from the recursive

call back to the same **OC** that was in effect before the call. It makes the most sense to achieve this with a native return to a `DynamoRIO` constructed trace. However, to maintain program transparency, we cannot simply replace the return address on the stack with the address of such a trace. Thus, we would need `DynamoRIO` to maintain such a return stack. This means we would have to add extra overhead to every call/return point that we want to use the new system, and maintain consistency.

This chapter presented the external API, and the requirements necessary for handling the recursive and iterative μ**VM**s in a better manner. The next chapter delves into the internal details of implementation necessary to effect the desired behaviour from `DynamoRIO-PE`.

Chapter 6

Implementation

This chapter provides an overview of the changes necessary to dispatch on multiple LoIs. This chapter is divided into two sections: the internal changes necessary to `DynamoRIO`, and a detailed explanation of a specific generated traces file.

6.1 Internal Changes

For a full walk-through of basic `DynamoRIO` execution, see Appendix A. We begin by introducing the newly created structures and modules, noting whether these structures are architecture-dependent or general. We then walk through emulation under `DynamoRIO-PE`, focusing on those parts that have changed from `DynamoRIO` execution.

6.1.1 Added Modules

The newly introduced modules are `poi` (general) and `oc` (specific). The `poi` module is responsible for interacting with the `monitor` module in determining when to start and stop trace recording, and under what conditions direct links are possible. The `oc` module maintains the data structures `OptimizationContext`, `POIBasicBlock`, and some utility structures for representing LoI, `GlobalOpParam` and `StackOpParam`. The `OptimizationContext` has already been described; it contains a starting application pc, and a list of LoIs, implemented as instances of either `GlobalOpParam` or

StackOpParam. The POIBasicBlock represents traces and candidate traces. Each POIBasicBlock has an associated **OC**, a counter, and a possible Trace. The counter keeps track of how *hot* the POIBasicBlock is; that is, the number of times we have tried to emulate starting at the application pc with the values of LoIs in the **OC**.

6.1.2 Modified Execution

We now describe the differences in execution between DynamoRIO-PE and the unmodified DynamoRIO system. These differences are often similar to those between DynamoRIO-with-Log-PC and DynamoRIO. We begin by detailing how the interp module responds differently when constructing a basic block containing a call to **PoI** (basic blocks not containing such a call are handled exactly as in DynamoRIO).

Building the Basic Block

The first time that a basic block containing a call to **PoI** is to be executed, the dispatch module will not find a corresponding fragment and so will return to interp. interp then finds the target application pc, and constructs a new basic block through its call to build_basic_block, which in turn calls build_bb_ilist. build_bb_ilist decodes the executable loosely (L1 decoding, only the opcode) at the target until it encounters a control-transfer instruction (CTI). The instruction bundle containing all instructions between the start of decoding and the CTI are appended to the block's instruction list. Since this basic block includes a call to **PoI**, the first CTI will be a function call. For reasons we detail shortly, the first function call found will target the pre_poi function in the poi module. DynamoRIO-PE performs a check to ascertain that the function call targets pre_poi. When successful, the instruction list currently being created is stored temporarily, and a new instruction list is created to record new exit stub code.

New instructions are appended to the exit stub until a call to **PoI** is seen. The reason instructions start getting appended to the exit stub is that if we ever want to link directly from this site to a trace, we want to avoid the overhead of the call to

PoI. Placing instructions in the exit stub is related to why the first call is to `pre_poi` instead of **PoI** itself; the call to `pre_poi` marks the start of code that can safely be and should be placed in the exit stub. When a call to **PoI** is encountered, `DynamoRIO--PE` appends a few arguments: a `push app_pc`, `push %ebp`, and a placeholder `push 0` instruction are inserted immediately prior to the call. This matches the internal signature for **PoI**:

`point_of_interest(Linkstub *ls, int *base, app_pc target,`
`int` *`stack_loi_count`*`, ..., int` *`global_loi_count`*`, ...)`.

The basic block is then finalized, mangled, and emitted before control returns to `dispatch`.

Finalization of the block includes associating the generated exit stub with the final instruction in the block and marking the following basic block `FutureFragment` as a candidate trace head. The effects of the code mangling and emitting modules are similar to that in `DynamoRIO`. The only difference is that while emitting code in `emit`, a call occurs to finalize the arguments for **PoI**. This function replaces the placeholding push mentioned earlier with pushing the address of the Linkstub structure corresponding to the exit stub in which the call to **PoI** has been placed.

Emulating the Basic Block

Upon being reinvoked from `interp`, `dispatch` then jumps to the newly emitted code. Emulation eventually makes its way to the exit stub where the setup for and call to **PoI** occurs.

The actual implementation of **PoI** is minimal. Its only purpose is to mark the following instruction as a candidate tracehead by associating it with an optimization template. Thus, the function simply tests to see if the following instruction has already been marked; if not, then it is marked with the current **OC** as a template.

Once emulation of the **PoI** call occurs, execution returns to the `dispatch` module in `DynamoRIO-PE`. As mentioned in Appendix A, the `dispatch` module informs the `monitor` module before beginning emulation of a basic block. In `DynamoRIO-PE`, the `monitor` module then makes one of two calls into the `poi` module depending on

context. If the `monitor` is currently recording a trace, before extending the trace with the next basic block, it makes a call into `extend_poi_trace` asking if the block should be added to the current trace, or if the trace should end. Alternatively, if no trace is being recorded, the `monitor` checks the flags of the basic block provided by `dispatch` to see if the block denotes a candidate trace head. If so, then the `monitor` calls `start_poi_trace` in the `poi` module. The next section discusses the implementation of `extend_poi_trace` and `start_poi_trace`.

Implementation of Trace Policies

This section describes the implementation of two policy guiding functions in the `poi` module. We first describe the function that determines when a new trace should be recorded, `start_poi_trace`; this function is also responsible for determining if a trace exists that can be emulated. We then describe the related function `extend_poi_trace` that determines whether a trace should be extended with a basic block.

```
bool start_poi_trace(DynamoContext *dcontext, Fragment **f);
```

`start_poi_trace` is called on any basic block that has been flagged as a candidate trace head. The first step in this function is testing to see if the provided `Fragment` has an associated optimization template. If not, then no trace should start, and thus `false` is returned. If there is an optimization template, then a current **OC** is created by recording the current values of the LoIs stored in the template. We then use this **OC** as the lookup key in the table of `POIBasicBlock`s. If no `POIBasicBlock` yet exists for this **OC**, we have a choice; we can either add a new entry for this **OC**, or find a more general **OC** (fewer LoIs) that does exist, or add a new entry for a more general **OC**. If the application pc of the basic block already has too many traces associated with it, we try and find a more generalized **OC** that already exists; if no such entry exists, then we insert a new `POIBasicBlock` with a more generalized **OC**.

Once we have a `POIBasicBlock`, we check to see if an accompanying trace already exists. If so, then we update the argument `Fragment` to the corresponding `Trace`. This is also the point at which direct linking is performed; if emulation has just come from a trace with a valid **OC** that is compatible with that of the found trace, then

the most recent exit stub is directly linked to the target trace in the code cache. In any case, if an existing trace is found, we return `false` (telling `monitor` not to start a new trace, but just to dispatch to the new target). If no trace exists, we increment the `POIBasicBlock`'s counter and compare the value to the hot threshold value. If the `POIBasicBlock` has reached the threshold, we return `true` implying to start recording a new trace. Otherwise, we return `false`. We next describe the policy for whether to extend a trace with a given basic block.

```
bool extend_poi_trace(DynamoContext *dcontext, Fragment *f);
```

We define trace stopping points in a similar manner to those defined by `Dynamo-RIO-with-Log-PC`. Specifically, we always stop recording a trace at the point of a new call to **PoI**. Additionally, we stop recording when the number of basic blocks exceeds the runtime threshold. The last condition we check for not extending the trace is if the trace contains a loop that has been unrolled a number of times equal to the threshold. When we stop extending a trace, we also mark the last instruction with an empty optimization template whose tag is the instruction heading the subsequent basic block. This is also the point at which we annotate any synthetic calls to point of interest.

Following `DynamoRIO`'s lead, we could mark every conditional exit point as a candidate trace head. Instead, however, we only selectively mark certain instructions. Remember that at the point of trace recording, the `poi` module has no knowledge of the actual **OC**; every **OC** except for that at the start of the trace will not be known until the trace is being optimized, since each instruction could modify the **OC**. Thus, during trace extension, we mark that certain instructions should have their live **OC** associated as a template for the head of a basic block. For instance, if we wanted to propagate information from before a call to after the call, we would mark the call instruction with an empty optimization context here. This **OC** would have as its application pc tag the address of the instruction following the call. Later, after the **OC** live at the call point is filled in, we will see that the target tag is the instruction following the call point, and so mark that instruction as a candidate trace head, with an optimization template given by the **OC**. Alternatively, in a loop we could mark

the instruction following the loop with a detailed optimization template so that if the **OC** is generalized to avoid loop unrolling, it is still possible to regain information after the loop. Thus, we propagating information past recursion and iteration through this single mechanism.

6.2 Trace Changes

We now take as a concrete example the recursive implementation of μ**VM**, walking through the traces file and how it is generated. Figure 6-1 and Figure 6-2 present the first trace generated (see Appendix B for the full traces file). The full file consists of three different trace segments; the traces shown in Figure 5-2 (page 52). There are two traces representing the start of two calls to the `eval`$_r P$ function, and a final trace containing the third call and the remnant of the first two.

6.2.1 Trace Header and Original Code

We begin by analyzing Figure 6-1 line by line in its entirety. The first line provides a unique identifier in the list of traces; this is `TRACE` number two (in a zero-based counting scheme), even though it was the first recorded. The next line presents the unique `Fragment` identifier assigned to this trace. Either of these can be used to track the trace's development in the log file. The third line presents the `Tag` associated with the trace; this is the starting pc from the emulated application's perspective. The next line, `CachePC` displays the physical address at which this trace's corresponding code is placed in `DynamoRIO-PE`'s code cache.

The fifth line begins describing the `optimization context` against which this trace was optimized. Each optimization context automatically includes the tag (same as the trace's tag mentioned earlier) as a LoI. The `frame` field is used to keep track of when stack-based parameters are valid; more will be said about this later. The next line presents the number of stack-based LoIs contained in this **OC**. This particular **OC** has 1 stack based LoI, at an offset of 8 from the frame pointer (`%ebp`) with the value of 0. Since the number in square brackets ([0]) and the **OC**'s frame match, this

```
1  TRACE # 2
2  Fragment # 184
3  Tag = 0x0804833f
4  CachePC = 0x40064004
5  Optimization_Context (tag = 0x0804833f, frame = 0)
6          # of stack_based params = 1
7          stack param 0: 8(%ebp[0]) := 00
8          # of globally addressed params = 0
9  Thread = 16214
10
11 ORIGINAL CODE:
12 basic block # 0: start pc = 0x0804833f
13 [eval() @ bench1.c:100]  operation = instrs[pc];
14   0x0804833f   8b 0d 84 08 12 08    mov    0x8120884 -> %ecx
15   0x08048345   8b 45 08             mov    0x8(%ebp) -> %eax
16 [eval() @ bench1.c:102]  if (operation == END_OP) {
17   0x08048348   83 c4 10             add    $0x10 %esp -> %esp
18 [eval() @ bench1.c:100]  operation = instrs[pc];
19   0x0804834b   0f be 14 08          movsx  (%eax,%ecx,1) -> %edx
20 [eval() @ bench1.c:102]  if (operation == END_OP) {
21   0x0804834f   85 d2                test   %edx %edx
22   0x08048351   75 0d                jnz    $0x8048360
23 exit pc = 0x08048353
24 basic block # 1: start pc = 0x08048360
25 [eval() @ bench1.c:105]  } else if (operation == CONTINUE_OP) {
26   0x08048360   83 fa 01             cmp    %edx $0x01
27   0x08048363   75 1d                jnz    $0x8048382
28 exit pc = 0x08048365
29 basic block # 2: start pc = 0x08048365
30 [eval() @ bench1.c:108]    rec = eval(pc+2);
31   0x08048365   83 c4 f4             add    $0xf4 %esp -> %esp
32   0x08048368   83 c0 02             add    $0x02 %eax -> %eax
33   0x0804836b   50                   push   %eax %esp -> %esp (%esp)
34   0x0804836c   e8 b3 ff ff ff       call   $0x8048324
35 direct call at 0x0804836c
36       continuing in callee at 0x08048324
37 [eval() @ bench1.c:0]
38   0x08048324   55                   push   %ebp %esp -> %esp (%esp)
39   0x08048325   89 e5                mov    %esp -> %ebp
40   0x08048327   83 ec 08             sub    $0x08 %esp -> %esp
41 [eval() @ bench1.c:97]    POI(1,&pc,0);
42   0x0804832a   e8 bd 2a 01 00       call   $0x805adec <pre_poi>
43 direct call at 0x0804832a
44 NOT inlining call to 0x0805adec
45   0x0804832f   83 c4 fc             add    $0xfc %esp -> %esp
46   0x08048332   6a 00                push   $0x00 %esp -> %esp (%esp)
47   0x08048334   8d 45 08             lea    0x8(%ebp) -> %eax
48   0x08048337   50                   push   %eax %esp -> %esp (%esp)
49   0x08048338   6a 01                push   $0x01 %esp -> %esp (%esp)
50   0x0804833a   e8 25 27 01 00       call   $0x805aa64 <point_of_interest>
51 RIOEXT: Found poi call at 0x0804833a, so ending block
52 exit pc = 0x0804833f
53 END ORIGINAL CODE
```

Figure 6-1: This figure presents the header and original code section of a trace from `DynamoRIO-PE`'s `trace` output file. This particular trace is the first presented in Figure 5-2 (page 52). The header presents some general information, while the original code section intersperses the originating C code with the produced assembly (also noting the boundary of `DynamoRIO` basic blocks). Each line of C code is prefaced with the containing function, file, and line number. Each line of assembly contains the address (as the application sees it), the actual representation in memory (organized into bytes represented as 2 hexadecimal digits), and an operation followed by the source and target operands. Due to instruction reordering and the complex nature of C statements, the layout of generated assembly is not entirely synchronized with the original C code.

parameter is valid in the current stack frame. This **OC** has no global LoI. The last line of this section specifies the thread identifier provided by the operating system, `Thread` 16214.

The next section in the Figure displays the code on which this trace is based. The first basic block begins at application pc `0x0804833f`. Line 13 presents an excerpt from the generating C source-code file. `eval()` is the name of the containing function, `bench4.c` is the file from which the line was taken, and the line was the `100`th in the file. The next two lines present some assembly code that (in part) achieves the effects of the C statement (`operation = instrs[pc]`. First, the address of the `instrs` array is loaded into the `%ecx` register. Then, the value at an offset of 8 from the current frame (the stack address allocated for `pc`) is loaded into the `%eax` register. While the memory referencing instructions are working, the stack is prepared for the next scope at lines 16 and 17. The next instruction dereferences the memory location for `instrs[pc]`, and sign extends the value into the `%edx` register. The expression `END_OP` is defined as the value zero, and so the next instructions test if `%edx` is zero. The conditional jump if not zero ends the first `DynamoRIO-PE` basic block at application pc `0x08048353`.

Line 24 begins the next basic block, at the application pc value `0x08048370`. This implies that the conditional jump was taken while recording the trace; we'll later discover that specializing against the value `pc` as zero guarantees that the conditional jump will be taken. The value of `op` (in register `%edx`) is then compared against `CONTINUE_OP`, which is the value one. The next conditional jump ends the second basic block.

Line 29 begins the third basic block, which begins the recursive call into $\texttt{eval}_r P$. The first assembly instruction prepares the stack to be aligned on a 16-byte (paragraph) boundary. The following assembly instruction (Line 32) prepares register `%eax` with the value `pc+2`, which is then pushed as the argument for the call. We see on lines 35 and 36 that the call instruction has been continued without creating a basic block boundary. Lines 38 and 39 are the standard entry in a subroutine, saving a copy of the current frame pointer and copying the current stack pointer into the

frame pointer. Line 40 again aligns the stack pointer on a 16-byte boundary (since the return address and the previous frame pointer, two double-words, were pushed, alignment requires another 8 bytes). Line 42 is recognized as a `DynamoRIO-PE`-internal function (`pre_poi`), and so the call is not inlined. The following instructions push the arguments to **PoI** in reverse order. As shown in line 41, the arguments are: one, the address of `pc`, and 0. This tells `DynamoRIO-PE` to expect no global LoI and one stack-based LoI (the address of `pc`). Thus, lines 45-49 prepare the stack for alignment, push the value zero, the address of `pc`, and finally the value one. Since `DynamoRIO-PE` has seen a call to **PoI** now, this ends the current trace.

This section has walked through the exact desired behaviour of this trace, and where each instruction originated. The next section walks through the actual assembly for this trace that achieves the desired behaviour, given the values of LoIs in this trace's **OC**.

6.2.2 Optimized Trace

This section discusses every aspect of the generated trace for the first call to $\texttt{eval}_r P$, with argument `pc` set to zero. The generated trace is shown in Figure 6-2, which has been generated from the original code and **OC** defined in Figure 6-1.

The first line presents the size of the entire trace in bytes, which in this case is 98 bytes. The rest of this figure presents the body of the trace. The indirect branch target entry and the prefix entry finish performing the context switch necessary to resume emulation after executing `DynamoRIO-PE`, coming from different contexts. The first disassembled instruction copies the constant value of the address of `instrs[0]` into register `%ecx`. Although `gcc` was only using `%ecx` as a temporary register, and this assignment is unnecessary, `DynamoRIO-PE` faithfully keeps the assignment.

The next instruction is a combination of the stack alignment prior to the call to $\texttt{eval}_r P$ and the reclamation of stack space from line 17 of Figure 6-1. Notice that the assignment of `pc` to register `%eax` has been removed entirely. Also, the dereferencing operation to load `operation` into register `%edx` has been replaced with an assignment from `CONTINUE_OP`. These optimizations are possible because we have

```
 1  Size = 98
 2  Body:
 3    -------- indirect branch target entry: --------
 4    0x40064004   a1 c0 14 00 40       mov    0x400014c0 -> %eax
 5    -------- prefix entry: --------
 6    0x40064009   8b 0d c8 14 00 40    mov    0x400014c8 -> %ecx
 7    -------- normal entry: --------
 8    0x4006400f   c7 c1 d0 12 12 08    mov    $0x081212d0 -> %ecx
 9    0x40064015   83 c4 04             add    $0x04 %esp -> %esp
10    0x40064018   c7 c2 01 00 00 00    mov    $0x00000001 -> %edx
11    0x4006401e   c7 c0 02 00 00 00    mov    $0x00000002 -> %eax
12    0x40064024   68 02 00 00 00       push   $0x00000002 %esp -> %esp (%esp)
13    0x40064029   68 71 83 04 08       push   $0x08048371 %esp -> %esp (%esp)
14    0x4006402e   55                   push   %ebp %esp -> %esp (%esp)
15    0x4006402f   89 e5                mov    %esp -> %ebp
16    0x40064031   83 ec 08             sub    $0x08 %esp -> %esp
17    0x40064034   e9 56 00 00 00       jmp    $0x4006408f <trace 185>
18    -------- exit stub 0: -------- <target: 0x0804833f>
19  Exit stub has following optimization context:
20  Optimization_Context (tag = 0x0804833f, frame = 1)
21          # of stack_based params = 3
22          stack param 0: 8(%ebp[0]) := 00
23          stack param 1: 4(%ebp[1]) := 0x08048371
24          stack param 2: 8(%ebp[1]) := 02
25          # of globally addressed params = 0
26    0x40064039   83 c4 fc             add    $0xfc %esp -> %esp
27    0x4006403c   6a 00                push   $0x00 %esp -> %esp (%esp)
28    0x4006403e   8d 45 08             lea    0x8(%ebp) -> %eax
29    0x40064041   50                   push   %eax %esp -> %esp (%esp)
30    0x40064042   6a 01                push   $0x01 %esp -> %esp (%esp)
31    0x40064044   68 3f 83 04 08       push   $0x0804833f %esp -> %esp (%esp)
32    0x40064049   55                   push   %ebp %esp -> %esp (%esp)
33    0x4006404a   68 f0 39 08 40       push   $0x400839f0 %esp -> %esp (%esp)
34    0x4006404f   e8 10 6a ff c7       call   $0x805aa64 <point_of_interest>
35    0x40064054   83 c4 0c             add    $0x0c %esp -> %esp
36    0x40064057   a3 c0 14 00 40       mov    %eax -> 0x400014c0
37    0x4006405c   b8 f0 39 08 40       mov    $0x400839f0 -> %eax
38    0x40064061   e9 4f 02 fd ff       jmp    $0x400342c0 <fcache_return>
39  END TRACE 2
```

Figure 6-2: This figure presents the contents of a generated trace. It includes a full dissassembled listing of all prefixes, the main body, and the collection of exit stubs. Each exit stub is marked with its corresponding optimization context (if one exists).

optimized against the fact that `pc` has a value of zero at the start of this trace, and that the `instrs` array is a region of immutable data. As with the assignment to register `%ecx`, register `%edx` is also unnecessary at this point, but without inter-trace optimizations, this cannot be detected.

Line 11 stores the value of the next `pc` (here, optimized to the value `2`) into register `%eax` and also pushes this value as the argument to $\texttt{eval}_r P$. Line 13 is the only instruction added by the trace that is not obviously in the original code; it is the translation of the `call` instruction on line 34 in Figure 6-1 to push the return address. Since the frame pointer, register `%ebp`, is still considered dynamic data, the trace cannot optimize its reference into a constant on line 14. This also applies to the reference to the stack pointer, register `%esp`, on line 15. Finally, the stack alignment on line 16 must also remain in the trace. The last instruction in this trace has been *linked* to another recorded trace, with `Fragment` identifier 187. This trace is identical to this trace, with the following two exceptions. It has been optimized against location `pc` having the value two and it has a broader **OC** that is described later in this section. If the direct link had not been made, line 17 would instead be a jump to exit stub 0, at cache address `0x40064039`.

We note that the trace's body contains 42 bytes of instruction data compared to 43 in the original code. It also has 10 instructions compared to 15 in the original code. We also remark that the generated trace has had all conditional jumps removed, and all static values propagated as constants rather than memory or register references. The only memory references left are necessary stack manipulations.

We now examine this trace's only exit stub, noting that if there had been any conditional jumps left in the body there would be one more exit stub for each jump. The purpose of this exit stub was described earlier in this Chapter, in Section 6.1. Specifically, it was created to avoid the overhead of preparing an **OC** and calling **PoI** when a direct link from this trace to another is possible.

In explaining this exit stub we begin with some general observations about the exit stub, and an explanation of the derived **OC**. We then explain each disassembled instruction.

The first thing we see on line 19 is that this exit stub is direct, and the target is to the application pc `0x0804833f`. Line 19 presents the **OC** associated with this exit stub produced from optimization of the trace body. This **OC** is similar to that of the head of this trace (shown in Figure 6-1), but has notable differences. The first apparent difference is that this **OC**'s frame is the value one. The next difference is that there are three known stack parameters, spanning across multiple stack frames. The first stack LoI is in the frame one higher than the **OC**'s current frame and has been propagated from this trace's original **OC**, since the address has not been overwritten in the trace body. The next stack LoI was detected through optimization of the trace body, specifically the instruction pushing the return address in line 13 of Figure 6-2. The third, and final, stack-based LoI results from similar circumstance in line 12 of the trace body.

The instructions comprising this exit stub are a mixture of those generated by the C code presented in Figure 6-1 and extra instructions added by `DynamoRIO-PE` as described earlier in this Chapter. First, we remark that neither the trace body nor the exit stub contains the call to `pre_poi` shown in the original source code. `DynamoRIO-PE` has removed this instruction as unnecessary when generating the trace. The first five instructions in the exit stub (lines 26-30) are passed directly from the original code. They prepare the stack for alignment and push the arguments provided by the annotator to **PoI**. The next three pushes were added by `DynamoRIO-PE`, as explained earlier in this chapter, to inform **PoI** of the target application pc, current value of the frame pointer, and address of the Linkstub structure representing this exit stub, respectively. The subroutine call on line 34 performs the actual call into `DynamoRIO-PE`'s **PoI** function, and the add instruction on line 35 pops the space stored for the three `DynamoRIO-PE`-provided function arguments.

The final three instructions prepare the arguments and register state for the return into `DynamoRIO-PE` through the `fcache_return` function. Line 36 saves the value of register `%eax` into the current `DynamoContext`'s `MachineState` structure (notice the parallel load of `%eax` on line 4, when entering the code cache). The next instruction loads the address of this exit stub's `Linkstub` structure (same as in the call

to **PoI**) in preparation for the final instruction, the jump to the `fcache_return` function. The `fcache_return` function has to save the rest of the registers into the `MachineState` structure before returning control into `DynamoRIO-PE`. Eventually, when `fcache_return` makes its way back into the `dispatch` module, the value of `%eax` will be used as the last exit back into `DynamoRIO-PE`. This pointer is then used in the `poi` module to determine that the link can be made direct (and bypass both the call to **PoI** and any lookup code). As mentioned earlier, a link can be made direct if every LoI in the successive trace is contained in this exit stub's **OC**.

This section walked through one generated trace line-by-line, starting with the trace header, continuing with the original code and finishing with the generated trace instructions, both the trace body and the exit stubs. The chapter presented the changes to `DynamoRIO`'s structure necessary to effect the desired behaviour of `Dynamo-RIO-PE`, as detailed in Chapter 5. The next chapter describes the current state of `DynamoRIO-PE` and presents some very preliminary performance numbers for different interpreters.

Chapter 7

Analysis

This chapter provides more information on the current progress of `DynamoRIO-PE`. We first discuss the mechanisms implemented very generally, then show some performance numbers on a more realistic interpreter than μ**VM**, TinyVM. Next, we discuss ease of annotating more interpreters, and provide some very preliminary performance numbers for two widely used interpreters, OCaml and Ruby.

7.1 Capabilities of `DynamoRIO-PE`

This section discusses the mechanisms that are currently implemented in `DynamoRIO-PE`. We remark that **PoI** annotations for an arbitrary number of LoIs create traces as described earlier in this thesis. We have also implemented naive algorithms for detecting and linking direct transitions among traces. The rest of this section mentions some other mechanisms currently implemented in **PoI**.

`DynamoRIO-PE` is able to add LoIs that it knows about to a given **PoI** call. This capability is demonstrated in the traces file presented in Appendix B. With this mechanism, `DynamoRIO-PE` is able to generate precisely the desired traces for $\texttt{eval}_r P$ shown in Figure 5-2 (page 52). This mechanism still needs much work on policy guidance, though.

Finally, `DynamoRIO-PE` is able to split apart a trace containing a loop into different sections as described and presented in Chapter 5. This includes: 1) not extending a

trace when it has unrolled a loop a specified number of times, 2) creating a new trace for a single loop iteration, 3) generalizing a single loop iteration, if necessary, to allow a direct link (to avoid further unrolling), and 4) resuming a specialized context after generalization in step 3.

The next section presents some performance numbers for emulation of TinyVM interpreting a range of programs under DynamoRIO-PE.

7.2 Performance of TinyVM under DynamoRIO-PE

The TinyVM interpreter is a much more realistic virtual machine than μ**VM**: it includes operations for function calls and returns, conditional branching, and a collection of primitive operations (e.g. arithmetic, logical). Although we have not focused on performance yet, initial results are very encouraging. For some input programs, DynamoRIO-PE can even achieve better performance than DynamoRIO-with-Log-PC, although it is slightly slower in general. We next present some performance numbers and briefly describe the characteristics of some test input programs.

	native	DynamoRIO	DynamoRIO-with-Log-PC	DynamoRIO-PE
bubble	24.01	44.28	14.68	15.08
fibonacci	27.20	36.35	15.65	14.92
matrix	4.00	8.07	4.67	6.22
matrix2	29.43	55.65	22.24	24.48
sieve	28.37	52.14	18.17	18.68

Table 7.1: Timing information comparing the performance of TinyVM interpreting 5 applications natively, and under DynamoRIO, DynamoRIO-with-Log-PC, and DynamoRIO-PE.

Table 7.1 presents performance numbers on five input programs. All of these benchmarks only perform a single indirect logical jump (the logical return), and thus all traces formed are able to be linked directly. DynamoRIO-PE will pay a much higher cost when the link cannot be made direct, because of its intrinsically higher dispatch cost. Another characteristic of these programs is that they all express only a single LoI, the logical pc. DynamoRIO-PE's performance on these benchmarks is very

encouraging. For these five test programs, the runtime parameters were uniformly: '-hot_threshold 10 -constant_prop 32 -stack_adjust -remove_dead_code 32 -call_return_

7.3 Applicability to Larger Systems

This section presents the application of DynamoRIO-PE to some realistic interpreters. We have so far tested interpreters for OCaml and Ruby running under DynamoRIO--PE execution. This work was made easier by the fact that these interpreters had already been annotated for use with DynamoRIO-with-Log-PC. As shown in Table 5.1 (page 50), there is a direct translation from DynamoRIO-with-Log-PC annotations to those used by DynamoRIO-PE.

	TinyVM	OCaml	Ruby
Number of lines inserted for DynamoRIO	8	7	9
Number of lines inserted for immutable regions	7	2	240
Number of lines inserted for DynamoRIO-PE	8	3	2
Total lines inserted	23	12	251
Lines in Interpreter Source File	495	1036	9283
Number of lines inserted for DynamoRIO-PE	8	3	2
Number of lines inserted for DynamoRIO-with-Log-PC	22	13	11

Table 7.2: Breakdown of the number of lines of code added to three interpreters to effect each module of execution under DynamoRIO-PE. The row relating lines in interpreter source file only counts the number of lines in the main interpreter file (generally interp.c or eval.c). The last line presents the number of lines necessary for annotating the interpreter for use under DynamoRIO-with-Log-PC as a comparison to lines for DynamoRIO-PE.

Table 7.2 presents the number of added lines to make three interpreters run under DynamoRIO-PE's control. We report the number of lines necessary to call into DynamoRIO to gain control (DynamoRIO can inject itself into executables to take control, but we have specifically given control to DynamoRIO). We then present the number of lines to annotate the immutable regions. Note that these first two line counts would be the same when annotating for DynamoRIO-with-Log-PC as well. We then report the number of lines necessary to annotate **PoI** and their corresponding LoIs. We report the number of lines that had to be changed for correct execution in the following

tables. We finally remark that the existence of annotations for `DynamoRIO-with--Log-PC` allowed all changes necessary to allow these three interpreters to run under `DynamoRIO-PE` to be done in less than a day. Further, the annotations added are very simple and straightforward.

OCaml	native	`DynamoRIO`	`DynamoRIO-with-Log-PC`	`DynamoRIO-PE`
ary3	7.95	7.99	5.86	26.96
ackermann	28.65	27.40	55.62	798.60
fib	12.09	14.48	18.17	369.24
hash2	5.17	5.21	6.27	34.07
matrix	11.22	10.75	17.67	25.83
methcall	4.63	4.62	1.88	39.31
nestedloop	15.92	15.39	5.18	198.08
sieve	6.23	6.41	4.65	36.57
tak	7.72	7.57	6.86	149.63

Table 7.3: Some timing information for running OCaml on various applications, under various systems. The different systems presented here are: native execution, under `DynamoRIO`, under `DynamoRIO-with-Log-PC`, and under `DynamoRIO-PE`.

Table 7.3[1] presents some performance numbers comparing the performance of OCaml interpretation under various systems. Although `DynamoRIO-PE`'s performance is much worse, it is worthy to note that the various programs currently run.

Ruby	native	`DynamoRIO`	`DynamoRIO-with-Log-PC`	`DynamoRIO-PE`
ary3	6.40	8.79	7.85	25.04
fib	251.11	312.62	286.73	953.22
hash2	11.47	14.90	*	1752.86
matrix	2.44	3.55	3.58	12.46
methcall	38.56	51.69	40.23	146.61
nestedloop	190.53	242.52	208.91	565.42
sieve	9.65	12.98	10.60	41.68
tak	190.96	250.69	222.22	1048.96

Table 7.4: Some timing information for running Ruby on various applications, under various systems. The different systems presented here are: native execution, under `DynamoRIO`, under `DynamoRIO-with-Log-PC`, and under `DynamoRIO-PE`.

Table 7.4 presents some performance numbers comparing the performance of Ruby

[1]Most of these benchmarks were taken from http://www.bagley.org/ doug/shootout/bench/. For the two that were not: `tak` was taken from http://www.lib.uchicago.edu/keith/crisis/benchmarks/, and `fib` was written by the author as a naive, recursive function.

interpretation under various systems. Although DynamoRIO-PE's performance is much worse, it is worthy to note that the various programs currently run. Another interesting note about both OCaml and Ruby is that native execution, DynamoRIO, and DynamoRIO-with-Log-PC generally perform equivalently, while DynamoRIO-PE shows a factor of 4 slowdown. The fib and ackermann benchmarks are enormously recursive (in fact, Ruby quickly fails the ackermann benchmark due to excessive stack depth). Whenever such logical recursion is present, there will be a high number of interpretations of logical indirect jumps, leading to indirect trace transitions. These benchmarks exemplify DynamoRIO-PE's poor performance for dispatch.

We have not yet investigated the cause of the heavy performance penalty of DynamoRIO-PE on either the OCaml or Ruby benchmark suites. Beyond the inherent inefficiency of creating and dispatching to traces, we believe that the current data structures in use scale very poorly to a large number of traces and candidate trace heads. More work can be done to determine the actual bottleneck and construct more scalable structures.

This chapter has given a brief analysis of the current state of DynamoRIO-PE. We first showed that it can perform the basic mechanisms described elsewhere in this thesis on simple input programs, and then show that it can perform on a level equivalent with DynamoRIO-with-Log-PC, better than both DynamoRIO and native execution, on a more complex version. On realistic interpreters, however, although DynamoRIO-PE can emulate the interpreters, performance suffers heavily. The next section presents some future work and concludes this thesis.

Chapter 8

Conclusion and Future Work

This chapter presents possibilities for future work and concludes the thesis. We begin by discussing current limitations in our work that we would like to remove in future versions. We then discuss possible areas of expansion. Finally, we summarize this thesis.

8.1 Future Work

One technical limitation in our current work is that LoIs are assumed to be integer sized (4 bytes). We would like to remove this restriction and allow arbitrary-length LoIs. We would also like to add register-based LoI, to supplement stack-based and globally addressed LoIs. Additionally, some of our analyses and optimizations rely on certain constraints (e.g. the C calling convention); we would like to isolate and relax these constraints. We are also looking to expand the expressiveness of our annotations, similar to the policy guidance in DyC. Along with each LoI, the annotator could provide a list of policy choices to guide partial evaluation, and the handling of individual LoIs. The partial evaluator has many decisions to make; controllable policies would help in obtaining better results.

We would like to devise a fast, efficient, algorithm for generalisation of optimization contexts. We can count the number of optimization contexts generated for a given program point, and if too many are being generated then create a few more

generalized contexts instead. In general, our algorithms are merely designed to be a proof of concept for including partial evaluation in `DynamoRIO`. As such, a full revision of algorithms and data structures would greatly help in the running time of optimizations. We would also like to apply our partial evaluation system to more complex examples than μ**VM** and TinyVM.

We would like to examine the feasability of automating annotations for `DynamoRIO-PE`. Currently, the only annotations we use are points of interest and marking immutable memory regions. There are interesting avenues to pursue for automatically classifying locations as LoIs, based on analyses of how beneficial classifying a location as static could be. To reduce our reliance on immutable memory region annotations, our work could be used in conjunction with mondrian memory protection [31] (MMP); this fine-grained protection scheme allows permissions control at the granularity of words. Using this scheme, we can make assumptions about the immutability of certain regions and mark any such region to cause a memory protection fault on an update. When handling this trap, we can then invalidate any traces relying on now invalid assumptions.

Another interesting area to look into is support for different *phases*. Applications generally can be divided into different phases of initialization, followed by processing. We currently support such a two-phase application by allowing the user to annotate regions as immutable after having initialized them. We would like to investigate how much more applicable our system could be if we provided an annotation for temporary immutability, or for marking a region as mutable.

Finally, we are interested in the notion of lazy **PoIs**. Currently, the value of every LoI is tested before dispatching to a particular trace. If the trace never specializes against the LoI, then this is unnecessarily specific. In a lazy **PoI**, LoIs are only tested immediately prior to the first point at which their value has been specialized against. This provides a whole new range of possibilities for performance improvement. For instance, if a certain LoI is purely a function of another then the check/dispatch can be removed completely. It is interesting to note that lazy **PoIs** are very similar to the way that `DynamoRIO` handles indirect jumps in traces. The trace has been optimized

against a particular target for an indirect jump; however, DynamoRIO waits until the point at which the jump would occur before testing to see if the current target is the same as the inlined target. If the test succeeds, execution remains in the trace, while otherwise, the trace is exited.

Finally, we are interested in achieving full-blown functional memoization purely through partial evaluation. Specifically, if a method has no side effects and depends only on its input parameters, it should be possible to specialize the program against all inputs and replace the function call with a lookup in a table. Currently, there are many hindrances to such a simple approach, such as stack modifications, memory writes, indirect jumps, and the x86's complex flag structure.

8.2 Conclusion

We began this thesis with a brief introduction to the problem domain and an overview of the thesis layout. We then provided related work across a range of different fields, finally narrowing in on dynamic optimization in DynamoRIO. After describing DynamoRIO and DynamoRIO-with-Log-PC in more detail, we demonstrated how polyvariant division and specialization can improve performance of a wide range of applications under DynamoRIO-PE, including interpreters. We then described some of the changes necessary to achieve our goals, and analysed the results. Finally, in this chapter we presented different ways to solidify and expand on our work.

Appendix A

Internal DynamoRIO Structure

This section describes the internal layout of DynamoRIO, going over the main structures used, and the interaction among different modules. DynamoRIO has been designed with a clean division between general DynamoRIO features and architecture-dependent features. Since the only supported architecture is still Intel's x86, all architecture-dependent files are in the x86 subdirectory and a general interface is provided for DynamoRIO's use. When discussing the internal structure, we make a note of whether a particular feature is architecture dependent through classification of either *general* or *specific*. Also, as a general convention, we describe execution of code in DynamoRIO's code cache as *emulation*, and execution of internal DynamoRIO's as *execution*.

A.1 DynamoRIO Structures

There are a number of different structures used to implement DynamoRIO. The main structure is the DynamoContext structure (general). This contains all of the current context known about the program; including the machine context (values of all visible registers, specific), the application's next targeted pc, error information, current trace information (when recording a trace), pointers for various memory management and a pointer to the most recent Linkstub to have exited the code cache. There is a DynamoContext for each active application thread. Additionally, there is a singleton

class `Options` (general) detailing parameters for execution (including maximum trace size, which optimizations to perform, etc.).

The related structures `Instr` (specific), `Opnd` (specific) and `InstrList` (general) are used to represent instructions in the application and any instructions we generate. An Instr represents a single instruction containing an `opcode`, and a list of source and destination operands (`Opnd`s). Useful functions dealing with instructions are decoding (creating a semantic instruction from its binary representation), encoding (the reverse function), and modifying source or destination operands. An `InstrList` contains any number of serial instructions. After an instruction list has been translated and placed in the code cache, a corresponding `Fragment` (general) structure is created and the instruction list is deleted; a related structure is the `Trace` (general).

The last structure we mention in this chapter is the `Linkstub` (general), whose use will be described later when discussing building block creation. Some fields in the `Linkstub` are a pointer to the target, flags indicating whether the jump is direct or indirect, and a pointer to the containing `Fragment`. This section has left out some structures irrelevant to this work.

A.2 Emulation of an Application

This section first introduces some modules involved in emulation, and then walks through the execution sequence of `DynamoRIO`. The main set of modules is: `dispatch` (general), `interp` (specific), `mangle` (specific), `emit` (general), `link` (general), and `monitor` (general).

When `DynamoRIO` assumes control of a program's execution, it begins execution in the `interp` module, `interp` function. The `interp` module is basically responsible for architecture dependent emulation of the program: it performs the translation of basic blocks upon demand and determines when emulation should stop. It also requests linking of fragments when translating a new basic block. Although `interp` builds the basic blocks, it uses the (general) `dispatch` function to actually emulate them. The invocation of `dispatch` only returns to `interp` when the targeted address is not in the

code cache; otherwise it handles jumping into the code cache to perform emulation, and retains control when emulation returns from the code cache. Additionally, `interp` makes calls into `monitor` to determine when to start a trace and other monitoring. We now walk through what happens when a basic block is to be emulated for the first time.

The first time that a basic block is to be executed, `dispatch` will not find a corresponding fragment and so will return to `interp`. `interp` then finds the target application pc, and constructs a new basic block through its call to `build_bbasic_block`, which in turn calls `build_bb_ilist`. `build_bb_ilist` decodes the executable loosely (L1 decoding, only the opcode) at the target until it encounters a control-transfer instruction (CTI). The instruction bundle containing all instructions between the start of decoding and the CTI are appended to the block's instruction list. If the CTI is a direct call or jump, and the basic block is still not too large, then the basic block continues accumulating at the point of the target. Otherwise, the basic block is *mangled* and *emitted*.

Mangling an instruction list is where the actual translation mentioned earlier occurs: `call` instructions are replaced with `push`ing the return address, indirect jumps prepare the registers for the eventual call to `dynamorio_indirect_branch_lookup`, etc.

The `emit` function is where a fragment is encoded and any necessary exit stubs are generated[1]. Each exit stub is represented by a corresponding instance of the `Linkstub` class. This `Linkstub` instance is used for *linking* or *unlinking* its source CTI. When linked, the CTI targets the destination fragment and bypasses the exit stub entirely. If the destination fragment is flushed from the cache or otherwise altered, then the CTI needs to be unlinked. Once the necessary exit stubs are generated, every instruction and the list of exit stubs is encoded into a `Fragment` that is placed in the code cache.

[1]An exit stub is necessary for any CTI that would leave the fragment. Such CTIs are then modified to target their corresponding exit stub, so that `DynamoRIO` can ensure total control. Basic Block fragments ending in conditional branches will contain two exit stubs: one for the taken branch and one for the fall-through case. Basic Block fragments ending in indirect branches will only contain one exit stub for the unconditional indirect jump. Traces, however, could contain any number of exit stubs.

Finally, at the tail end of emitting, the newly created fragment is linked. The list of `Linkstub`s is traversed looking for any jumps that can be linked (through a call to the `monitor` interface). If a direct jump is found targeting another fragment that already exists then the exit stub is linked to the destination fragment. Additionally, any direct jumps targeting fragments that have not yet been created are instead linked to a `FutureFragment` representing the eventual `Fragment`. Finally, candidate trace heads are marked while `monitor` checks for jumps that can be linked.

After the instruction list has been emitted, `dispatch` is called on the returned `Fragment`. The body of `dispatch` is a loop around actual dispatch to emulate the `Fragment`. Before `dispatch` actually jumps to the `Fragment`, however, it makes a call into `monitor` through `internal_dynamo_monitor`; this call lets `monitor` be aware that the `Fragment` is going to be run and update its internal counters for how hot a candidate trace head is. This is the point at which `monitor` decides whether to begin recording a trace. Should a trace be recorded at this point, `DynamoRIO` discards the extant basic block fragment, since the trace is equally applicable and probably more optimized. We discuss the details of trace generation shortly. In either case, execution eventually makes its way back into `dispatch` and the `Fragment` is emulated. Notice that emulation could actually traverse multiple basic blocks before returning to `dispatch`, if the `Fragment` is linked. Once execution returns from emulation, `dispatch` enters the next iteration of its loop, with the destination `Fragment` as the new target. If this `Fragment` does not yet exist, `dispatch` returns to `interp`.

We now give an overview of trace generation. If `monitor` decides to begin recording a trace, it initializes some variables in the `DynamoContext`, specifically the `trace_tag` field to the application address of the next fragment, and an `InstrList` to record the instructions. `monitor` then makes a call into the `interp` module to copy the instructions in the next fragment into the trace list. During this call, `interp` might reverse the condition for the most recent conditional jumps if the jump was taken in this trace. This is so that the jump is only taken if it would leave the trace; if the condition evaluates to the same as during trace recording we want it to stay in the trace. Also, the targeted fragment has all of its exit CTIs unlinked to force re-

entrance into `dispatch` (and thus `monitor`). Execution now returns to `dispatch` as mentioned before (and the `monitor` records the fact that it is currently recording a trace).

Eventually, in a future execution of `internal_dynamo_monitor`, the `monitor` decides to end the trace. When this happens, the accumulated instruction list is passed off to the `optimize` (specific) module for optimization. The resultant instruction list is then emitted exactly as described earlier for basic blocks. Finally, the original `Fragment` from which the trace was generated is removed so that future lookups return the generated `Trace`.

This section presented the structure and basic steps for emulating basic block fragments: emitting and linking fragments, recording traces, and dispatching to a fragment for emulation.

Appendix B

Generated Trace File

```
Shared library assumptions:

================================================================================

TRACE # 0
Fragment # 186
Tag = 0x0804833f
CachePC = 0x40064104
Optimization_Context (tag = 0x0804833f, frame = 2)
        # of stack_based params = 5
        stack param 0: 8(%ebp[0]) := 00
        stack param 1: 4(%ebp[1]) := 0x08048371
        stack param 2: 8(%ebp[1]) := 02
        stack param 3: 4(%ebp[2]) := 0x08048371
        stack param 4: 8(%ebp[2]) := 04
        # of globally addressed params = 0
Thread = 16214

ORIGINAL CODE:
basic block # 0: start pc = 0x0804833f
[eval() @ bench1.c:100]  operation = instrs[pc];
  0x0804833f   8b 0d 84 08 12 08    mov    0x8120884 -> %ecx
  0x08048345   8b 45 08             mov    0x8(%ebp) -> %eax
[eval() @ bench1.c:102]  if (operation == END_OP) {
  0x08048348   83 c4 10             add    $0x10 %esp -> %esp
[eval() @ bench1.c:100]  operation = instrs[pc];
  0x0804834b   0f be 14 08          movsx  (%eax,%ecx,1) -> %edx
[eval() @ bench1.c:102]  if (operation == END_OP) {
  0x0804834f   85 d2                test   %edx %edx
  0x08048351   75 0d                jnz    $0x8048360
exit pc = 0x08048353
basic block # 1: start pc = 0x08048353
[eval() @ bench1.c:103]    return instrs[pc+1];
  0x08048353   0f be 44 01 01       movsx  0x1(%ecx,%eax,1) -> %eax
  0x08048358   eb 28                jmp    $0x8048382
direct jump at 0x08048358
        continuing at target 0x08048382
[eval() @ bench1.c:133]}
  0x08048382   89 ec                mov    %ebp -> %esp
  0x08048384   5d                   pop    %esp (%esp) -> %ebp %esp
  0x08048385   c3                   ret    %esp (%esp) -> %esp
exit pc = 0x08048386
basic block # 2: start pc = 0x08048371
[eval() @ bench1.c:114]    tmp = instrs[pc+1];
```

```
  0x08048371   8b 55 08             mov    0x8(%ebp) -> %edx
[eval() @ bench1.c:108]    rec = eval(pc+2);
  0x08048374   89 c1                mov    %eax -> %ecx
[eval() @ bench1.c:114]    tmp = instrs[pc+1];
  0x08048376   a1 84 08 12 08       mov    0x8120884 -> %eax
  0x0804837b   0f be 44 10 01       movsx  0x1(%eax,%edx,1) -> %eax
[eval() @ bench1.c:117]    return tmp + rec;
  0x08048380   01 c8                add    %ecx %eax -> %eax
[eval() @ bench1.c:133]}
  0x08048382   89 ec                mov    %ebp -> %esp
  0x08048384   5d                   pop    %esp (%esp) -> %ebp %esp
  0x08048385   c3                   ret    %esp (%esp) -> %esp
exit pc = 0x08048386
basic block # 3: start pc = 0x08048371
[eval() @ bench1.c:114]    tmp = instrs[pc+1];
  0x08048371   8b 55 08             mov    0x8(%ebp) -> %edx
[eval() @ bench1.c:108]    rec = eval(pc+2);
  0x08048374   89 c1                mov    %eax -> %ecx
[eval() @ bench1.c:114]    tmp = instrs[pc+1];
  0x08048376   a1 84 08 12 08       mov    0x8120884 -> %eax
  0x0804837b   0f be 44 10 01       movsx  0x1(%eax,%edx,1) -> %eax
[eval() @ bench1.c:117]    return tmp + rec;
  0x08048380   01 c8                add    %ecx %eax -> %eax
[eval() @ bench1.c:133]}
  0x08048382   89 ec                mov    %ebp -> %esp
  0x08048384   5d                   pop    %esp (%esp) -> %ebp %esp
  0x08048385   c3                   ret    %esp (%esp) -> %esp
exit pc = 0x08048386
basic block # 4: start pc = 0x080482bf
[main() @ bench1.c:56]    val = eval(0);
  0x080482bf   89 c6                mov    %eax -> %esi
[main() @ bench1.c:55]  for(index=0; index < NUM_LOOPS; index++) {
  0x080482c1   83 c4 10             add    $0x10 %esp -> %esp
  0x080482c4   4b                   dec    %ebx -> %ebx
  0x080482c5   79 ee                jns    $0x80482b5
exit pc = 0x080482c7
basic block # 5: start pc = 0x080482b5
[main() @ bench1.c:56]    val = eval(0);
  0x080482b5   83 c4 f4             add    $0xf4 %esp -> %esp
  0x080482b8   6a 00                push   $0x00 %esp -> %esp (%esp)
  0x080482ba   e8 65 00 00 00       call   $0x8048324
direct call at 0x080482ba
      continuing in callee at 0x08048324
[eval() @ bench1.c:0]
  0x08048324   55                   push   %ebp %esp -> %esp (%esp)
  0x08048325   89 e5                mov    %esp -> %ebp
  0x08048327   83 ec 08             sub    $0x08 %esp -> %esp
[eval() @ bench1.c:97]    POI(1,&pc,0);
  0x0804832a   e8 bd 2a 01 00       call   $0x805adec <pre_poi>
direct call at 0x0804832a
NOT inlining call to 0x0805adec
  0x0804832f   83 c4 fc             add    $0xfc %esp -> %esp
  0x08048332   6a 00                push   $0x00 %esp -> %esp (%esp)
  0x08048334   8d 45 08             lea    0x8(%ebp) -> %eax
  0x08048337   50                   push   %eax %esp -> %esp (%esp)
  0x08048338   6a 01                push   $0x01 %esp -> %esp (%esp)
  0x0804833a   e8 25 27 01 00       call   $0x805aa64 <point_of_interest>
RIOEXT: Found poi call at 0x0804833a, so ending block
exit pc = 0x0804833f
END ORIGINAL CODE

Size = 289
Body:
  -------- indirect branch target entry: --------
  0x40064104   a1 c0 14 00 40       mov    0x400014c0 -> %eax
  -------- prefix entry: --------
  0x40064109   8b 0d c8 14 00 40    mov    0x400014c8 -> %ecx
  -------- normal entry: --------
```

```
  0x4006410f   c7 c0 04 00 00 00    mov    $0x00000004 -> %eax
  0x40064115   33 d2                xor    %edx %edx -> %edx
  0x40064117   89 ec                mov    %ebp -> %esp
  0x40064119   5d                   pop    %esp (%esp) -> %ebp %esp
  0x4006411a   c7 05 c8 14 00 40 d0 mov    $0x081212d0 -> 0x400014c8
               12 12 08
  0x40064124   59                   pop    %esp (%esp) -> %ecx %esp
  0x40064125   8d 89 8f 7c fb f7    lea    0xf7fb7c8f(%ecx) -> %ecx
  0x4006412b   e3 0b                jecxz  $0x40064138 %ecx
  0x4006412d   8d 89 71 83 04 08    lea    0x8048371(%ecx) -> %ecx
  0x40064133   e9 81 00 00 00       jmp    $0x400641b9 <exit stub 0>
  0x40064138   c7 c2 02 00 00 00    mov    $0x00000002 -> %edx
  0x4006413e   c7 c0 07 00 00 00    mov    $0x00000007 -> %eax
  0x40064144   89 ec                mov    %ebp -> %esp
  0x40064146   5d                   pop    %esp (%esp) -> %ebp %esp
  0x40064147   c7 05 c8 14 00 40 04 mov    $0x00000004 -> 0x400014c8
               00 00 00
  0x40064151   59                   pop    %esp (%esp) -> %ecx %esp
  0x40064152   8d 89 8f 7c fb f7    lea    0xf7fb7c8f(%ecx) -> %ecx
  0x40064158   e3 0b                jecxz  $0x40064165 %ecx
  0x4006415a   8d 89 71 83 04 08    lea    0x8048371(%ecx) -> %ecx
  0x40064160   e9 64 00 00 00       jmp    $0x400641c9 <exit stub 1>
  0x40064165   33 d2                xor    %edx %edx -> %edx
  0x40064167   c7 c0 09 00 00 00    mov    $0x00000009 -> %eax
  0x4006416d   89 ec                mov    %ebp -> %esp
  0x4006416f   5d                   pop    %esp (%esp) -> %ebp %esp
  0x40064170   c7 05 c8 14 00 40 07 mov    $0x00000007 -> 0x400014c8
               00 00 00
  0x4006417a   59                   pop    %esp (%esp) -> %ecx %esp
  0x4006417b   8d 89 41 7d fb f7    lea    0xf7fb7d41(%ecx) -> %ecx
  0x40064181   e3 0b                jecxz  $0x4006418e %ecx
  0x40064183   8d 89 bf 82 04 08    lea    0x80482bf(%ecx) -> %ecx
  0x40064189   e9 4b 00 00 00       jmp    $0x400641d9 <exit stub 2>
  0x4006418e   8b 0d c8 14 00 40    mov    0x400014c8 -> %ecx
  0x40064194   c7 c6 09 00 00 00    mov    $0x00000009 -> %esi
  0x4006419a   83 c4 10             add    $0x10 %esp -> %esp
  0x4006419d   4b                   dec    %ebx -> %ebx
  0x4006419e   0f 88 eb 57 05 00    js     $0x400b998f <fragment 187>
  0x400641a4   83 c4 f4             add    $0xf4 %esp -> %esp
  0x400641a7   6a 00                push   $0x00 %esp -> %esp (%esp)
  0x400641a9   68 bf 82 04 08       push   $0x080482bf %esp -> %esp (%esp)
  0x400641ae   55                   push   %ebp %esp -> %esp (%esp)
  0x400641af   89 e5                mov    %esp -> %ebp
  0x400641b1   83 ec 08             sub    $0x08 %esp -> %esp
  0x400641b4   e9 56 fe ff ff       jmp    $0x4006400f <trace 184>
  -------- exit stub 0: -------- <target: 0x40034340>
  0x400641b9   89 1d c4 14 00 40    mov    %ebx -> 0x400014c4
  0x400641bf   bb 80 53 08 40       mov    $0x40085380 -> %ebx
  0x400641c4   e9 77 01 fd ff       jmp    $0x40034340 <indirect_branch_lookup>
  -------- exit stub 1: -------- <target: 0x40034340>
  0x400641c9   89 1d c4 14 00 40    mov    %ebx -> 0x400014c4
  0x400641cf   bb a4 53 08 40       mov    $0x400853a4 -> %ebx
  0x400641d4   e9 67 01 fd ff       jmp    $0x40034340 <indirect_branch_lookup>
  -------- exit stub 2: -------- <target: 0x40034340>
  0x400641d9   89 1d c4 14 00 40    mov    %ebx -> 0x400014c4
  0x400641df   bb c8 53 08 40       mov    $0x400853c8 -> %ebx
  0x400641e4   e9 57 01 fd ff       jmp    $0x40034340 <indirect_branch_lookup>
  -------- exit stub 3: -------- <target: 0x080482c7>
  0x400641e9   a3 c0 14 00 40       mov    %eax -> 0x400014c0
  0x400641ee   b8 ec 53 08 40       mov    $0x400853ec -> %eax
  0x400641f3   e9 c8 00 fd ff       jmp    $0x400342c0 <fcache_return>
  -------- exit stub 4: -------- <target: 0x0804833f>
Exit stub has following optimization context:
Optimization_Context (tag = 0x0804833f, frame = 0)
        # of stack_based params = 2
        stack param 0: 4(%ebp[0]) := 0x080482bf
        stack param 1: 8(%ebp[0]) := 00
        # of globally addressed params = 0
```

```
  0x400641f8   83 c4 fc             add    $0xfc %esp -> %esp
  0x400641fb   6a 00                push   $0x00 %esp -> %esp (%esp)
  0x400641fd   8d 45 08             lea    0x8(%ebp) -> %eax
  0x40064200   50                   push   %eax %esp -> %esp (%esp)
  0x40064201   6a 01                push   $0x01 %esp -> %esp (%esp)
  0x40064203   68 3f 83 04 08       push   $0x0804833f %esp -> %esp (%esp)
  0x40064208   55                   push   %ebp %esp -> %esp (%esp)
  0x40064209   68 10 54 08 40       push   $0x40085410 %esp -> %esp (%esp)
  0x4006420e   e8 51 68 ff c7       call   $0x805aa64 <point_of_interest>
  0x40064213   83 c4 0c             add    $0x0c %esp -> %esp
  0x40064216   a3 c0 14 00 40       mov    %eax -> 0x400014c0
  0x4006421b   b8 10 54 08 40       mov    $0x40085410 -> %eax
  0x40064220   e9 90 00 fd ff       jmp    $0x400342c0 <fcache_return>
END TRACE 0

================================================================================

TRACE # 1
Fragment # 185
Tag = 0x0804833f
CachePC = 0x40064084
Optimization_Context (tag = 0x0804833f, frame = 1)
        # of stack_based params = 3
        stack param 0: 8(%ebp[0]) := 00
        stack param 1: 4(%ebp[1]) := 0x08048371
        stack param 2: 8(%ebp[1]) := 02
        # of globally addressed params = 0
Thread = 16214

ORIGINAL CODE:
basic block # 0: start pc = 0x0804833f
[eval() @ bench1.c:100]  operation = instrs[pc];
  0x0804833f   8b 0d 84 08 12 08    mov    0x8120884 -> %ecx
  0x08048345   8b 45 08             mov    0x8(%ebp) -> %eax
[eval() @ bench1.c:102]  if (operation == END_OP) {
  0x08048348   83 c4 10             add    $0x10 %esp -> %esp
[eval() @ bench1.c:100]  operation = instrs[pc];
  0x0804834b   0f be 14 08          movsx  (%eax,%ecx,1) -> %edx
[eval() @ bench1.c:102]  if (operation == END_OP) {
  0x0804834f   85 d2                test   %edx %edx
  0x08048351   75 0d                jnz    $0x8048360
exit pc = 0x08048353
basic block # 1: start pc = 0x08048360
[eval() @ bench1.c:105]  } else if (operation == CONTINUE_OP) {
  0x08048360   83 fa 01             cmp    %edx $0x01
  0x08048363   75 1d                jnz    $0x8048382
exit pc = 0x08048365
basic block # 2: start pc = 0x08048365
[eval() @ bench1.c:108]    rec = eval(pc+2);
  0x08048365   83 c4 f4             add    $0xf4 %esp -> %esp
  0x08048368   83 c0 02             add    $0x02 %eax -> %eax
  0x0804836b   50                   push   %eax %esp -> %esp (%esp)
  0x0804836c   e8 b3 ff ff ff       call   $0x8048324
direct call at 0x0804836c
      continuing in callee at 0x08048324
[eval() @ bench1.c:0]
  0x08048324   55                   push   %ebp %esp -> %esp (%esp)
  0x08048325   89 e5                mov    %esp -> %ebp
  0x08048327   83 ec 08             sub    $0x08 %esp -> %esp
[eval() @ bench1.c:97]    POI(1,&pc,0);
  0x0804832a   e8 bd 2a 01 00       call   $0x805adec <pre_poi>
direct call at 0x0804832a
NOT inlining call to 0x0805adec
  0x0804832f   83 c4 fc             add    $0xfc %esp -> %esp
  0x08048332   6a 00                push   $0x00 %esp -> %esp (%esp)
  0x08048334   8d 45 08             lea    0x8(%ebp) -> %eax
  0x08048337   50                   push   %eax %esp -> %esp (%esp)
  0x08048338   6a 01                push   $0x01 %esp -> %esp (%esp)
```

```
  0x0804833a   e8 25 27 01 00       call   $0x805aa64 <point_of_interest>
RIOEXT: Found poi call at 0x0804833a, so ending block
exit pc = 0x0804833f
END ORIGINAL CODE

Size = 98
Body:
  -------- indirect branch target entry: --------
  0x40064084   a1 c0 14 00 40       mov    0x400014c0 -> %eax
  -------- prefix entry: --------
  0x40064089   8b 0d c8 14 00 40    mov    0x400014c8 -> %ecx
  -------- normal entry: --------
  0x4006408f   c7 c1 d0 12 12 08    mov    $0x081212d0 -> %ecx
  0x40064095   83 c4 04             add    $0x04 %esp -> %esp
  0x40064098   c7 c2 01 00 00 00    mov    $0x00000001 -> %edx
  0x4006409e   c7 c0 04 00 00 00    mov    $0x00000004 -> %eax
  0x400640a4   68 04 00 00 00       push   $0x00000004 %esp -> %esp (%esp)
  0x400640a9   68 71 83 04 08       push   $0x08048371 %esp -> %esp (%esp)
  0x400640ae   55                   push   %ebp %esp -> %esp (%esp)
  0x400640af   89 e5                mov    %esp -> %ebp
  0x400640b1   83 ec 08             sub    $0x08 %esp -> %esp
  0x400640b4   e9 56 00 00 00       jmp    $0x4006410f <trace 186>
  -------- exit stub 0: -------- <target: 0x0804833f>
Exit stub has following optimization context:
Optimization_Context (tag = 0x0804833f, frame = 2)
        # of stack_based params = 5
        stack param 0: 8(%ebp[0]) := 00
        stack param 1: 4(%ebp[1]) := 0x08048371
        stack param 2: 8(%ebp[1]) := 02
        stack param 3: 4(%ebp[2]) := 0x08048371
        stack param 4: 8(%ebp[2]) := 04
        # of globally addressed params = 0
  0x400640b9   83 c4 fc             add    $0xfc %esp -> %esp
  0x400640bc   6a 00                push   $0x00 %esp -> %esp (%esp)
  0x400640be   8d 45 08             lea    0x8(%ebp) -> %eax
  0x400640c1   50                   push   %eax %esp -> %esp (%esp)
  0x400640c2   6a 01                push   $0x01 %esp -> %esp (%esp)
  0x400640c4   68 3f 83 04 08       push   $0x0804833f %esp -> %esp (%esp)
  0x400640c9   55                   push   %ebp %esp -> %esp (%esp)
  0x400640ca   68 10 3b 08 40       push   $0x40083b10 %esp -> %esp (%esp)
  0x400640cf   e8 90 69 ff c7       call   $0x805aa64 <point_of_interest>
  0x400640d4   83 c4 0c             add    $0x0c %esp -> %esp
  0x400640d7   a3 c0 14 00 40       mov    %eax -> 0x400014c0
  0x400640dc   b8 10 3b 08 40       mov    $0x40083b10 -> %eax
  0x400640e1   e9 cf 01 fd ff       jmp    $0x400342c0 <fcache_return>
END TRACE 1

================================================================================

TRACE # 2
Fragment # 184
Tag = 0x0804833f
CachePC = 0x40064004
Optimization_Context (tag = 0x0804833f, frame = 0)
        # of stack_based params = 1
        stack param 0: 8(%ebp[0]) := 00
        # of globally addressed params = 0
Thread = 16214

ORIGINAL CODE:
basic block # 0: start pc = 0x0804833f
[eval() @ bench1.c:100]  operation = instrs[pc];
  0x0804833f   8b 0d 84 08 12 08    mov    0x8120884 -> %ecx
  0x08048345   8b 45 08             mov    0x8(%ebp) -> %eax
[eval() @ bench1.c:102]  if (operation == END_OP) {
  0x08048348   83 c4 10             add    $0x10 %esp -> %esp
[eval() @ bench1.c:100]  operation = instrs[pc];
  0x0804834b   0f be 14 08          movsx  (%eax,%ecx,1) -> %edx
```

```
[eval() @ bench1.c:102]  if (operation == END_OP) {
  0x0804834f   85 d2               test   %edx %edx
  0x08048351   75 0d               jnz    $0x8048360
exit pc = 0x08048353
basic block # 1: start pc = 0x08048360
[eval() @ bench1.c:105]  } else if (operation == CONTINUE_OP) {
  0x08048360   83 fa 01            cmp    %edx $0x01
  0x08048363   75 1d               jnz    $0x8048382
exit pc = 0x08048365
basic block # 2: start pc = 0x08048365
[eval() @ bench1.c:108]    rec = eval(pc+2);
  0x08048365   83 c4 f4            add    $0xf4 %esp -> %esp
  0x08048368   83 c0 02            add    $0x02 %eax -> %eax
  0x0804836b   50                  push   %eax %esp -> %esp (%esp)
  0x0804836c   e8 b3 ff ff ff      call   $0x8048324
direct call at 0x0804836c
      continuing in callee at 0x08048324
[eval() @ bench1.c:0]
  0x08048324   55                  push   %ebp %esp -> %esp (%esp)
  0x08048325   89 e5               mov    %esp -> %ebp
  0x08048327   83 ec 08            sub    $0x08 %esp -> %esp
[eval() @ bench1.c:97]    POI(1,&pc,0);
  0x0804832a   e8 bd 2a 01 00      call   $0x805adec <pre_poi>
direct call at 0x0804832a
NOT inlining call to 0x0805adec
  0x0804832f   83 c4 fc            add    $0xfc %esp -> %esp
  0x08048332   6a 00               push   $0x00 %esp -> %esp (%esp)
  0x08048334   8d 45 08            lea    0x8(%ebp) -> %eax
  0x08048337   50                  push   %eax %esp -> %esp (%esp)
  0x08048338   6a 01               push   $0x01 %esp -> %esp (%esp)
  0x0804833a   e8 25 27 01 00      call   $0x805aa64 <point_of_interest>
RIOEXT: Found poi call at 0x0804833a, so ending block
exit pc = 0x0804833f
END ORIGINAL CODE

Size = 81
Body:
  -------- indirect branch target entry: --------
  0x40064004   a1 c0 14 00 40      mov    0x400014c0 -> %eax
  -------- prefix entry: --------
  0x40064009   8b 0d c8 14 00 40   mov    0x400014c8 -> %ecx
  -------- normal entry: --------
  0x4006400f   c7 c1 d0 12 12 08   mov    $0x081212d0 -> %ecx
  0x40064015   83 c4 04            add    $0x04 %esp -> %esp
  0x40064018   c7 c2 01 00 00 00   mov    $0x00000001 -> %edx
  0x4006401e   c7 c0 02 00 00 00   mov    $0x00000002 -> %eax
  0x40064024   68 02 00 00 00      push   $0x00000002 %esp -> %esp (%esp)
  0x40064029   68 71 83 04 08      push   $0x08048371 %esp -> %esp (%esp)
  0x4006402e   55                  push   %ebp %esp -> %esp (%esp)
  0x4006402f   89 e5               mov    %esp -> %ebp
  0x40064031   83 ec 08            sub    $0x08 %esp -> %esp
  0x40064034   e9 56 00 00 00      jmp    $0x4006408f <trace 185>
  -------- exit stub 0: -------- <target: 0x0804833f>
Exit stub has following optimization context:
Optimization_Context (tag = 0x0804833f, frame = 1)
        # of stack_based params = 3
        stack param 0: 8(%ebp[0]) := 00
        stack param 1: 4(%ebp[1]) := 0x08048371
        stack param 2: 8(%ebp[1]) := 02
        # of globally addressed params = 0
  0x40064039   83 c4 fc            add    $0xfc %esp -> %esp
  0x4006403c   6a 00               push   $0x00 %esp -> %esp (%esp)
  0x4006403e   8d 45 08            lea    0x8(%ebp) -> %eax
  0x40064041   50                  push   %eax %esp -> %esp (%esp)
  0x40064042   6a 01               push   $0x01 %esp -> %esp (%esp)
  0x40064044   68 3f 83 04 08      push   $0x0804833f %esp -> %esp (%esp)
  0x40064049   55                  push   %ebp %esp -> %esp (%esp)
  0x4006404a   68 f0 39 08 40      push   $0x400839f0 %esp -> %esp (%esp)
```

```
  0x4006404f   e8 10 6a ff c7       call   $0x805aa64 <point_of_interest>
  0x40064054   83 c4 0c             add    $0x0c %esp -> %esp
  0x40064057   a3 c0 14 00 40       mov    %eax -> 0x400014c0
  0x4006405c   b8 f0 39 08 40       mov    $0x400839f0 -> %eax
  0x40064061   e9 4f 02 fd ff       jmp    $0x400342c0 <fcache_return>
END TRACE 2
```

Bibliography

[1] Erik R. Altman, Kemal Ebcioğlu, Michael Gschwind, and Sathaye Sumedh. Advances and future challenges in binary translation and optimization. *Proceedings of the IEEE*, November 2001.

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, 35(10):47–65, October 2000.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, June 2000.

[4] Iris Baron. Dynamic optimization of interpreters using DynamoRIO. Master's thesis, M.I.T., 2003.

[5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1st International Symposium on Code Generation and Optimization (CGO-03)*, March 2003.

[6] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[7] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997.

[8] C. Chambers and D Ungar. Making pure object-oriented languages practical. *Proceedings OOPSLA '91*, pages 1–15, November 1991.

[9] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gillies. Mojo: A dynamic optimization system. *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[10] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *Symposium on Partial Evaluation*, 1998.

[11] C. Consel, J.L. Lawall, and A.-F. Le Meur. A Tour of Tempo: A Program Specializer for the C Language. *Science of Computer Programming*, 2004. To appear.

[12] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java Just in Time. *IEEE Micro*, 1997.

[13] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. *Conference Record of POPL '84: Symposium on Principles of Programming Languages*, pages 297–302, Jan 1984.

[14] Kemal Ebcioğlu and Erik Altman. DAISY: dynamic compilation for 100% architectural compatibility. In *ISCA*, 1997.

[15] Dawson R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Design and Implementation (PLDI '96)*, pages 160–170, Philadelphia, Pennsylvania, May 1996.

[16] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'c: A language for high-level, efficient, and machine-independent dynamic code generation. In *Symposium on Principles of Programming Languages*, pages 131–144, 1996.

[17] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 263–272, San Jose, California, 1994.

[18] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[19] U. Holzle and D Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[20] *IA-32 Intel Architecture Software Developer's Manual*. 2001.

[21] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993.

[22] Alexander Klaiber. The technology behind Crusoe processors. Transmeta Corporation, January 2000.

[23] D Michie. "memo" functions and machine learning. *Nature*, pages 218(1):19–22, Apr 1968.

[24] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *International Symposium on Microarchitecture*, pages 291–302, 2000.

[25] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.

[26] Ramesh Radhakrishnan, N. Vijaykrishnan, Lizy Kurian John, Anand Sivasubramaniam, Juan Rubio, and Jyotsna Sabarinathan. Java runtime systems: Characterization and architectural implecations. *IEEE Transactions on Computers*, 50(2), February 2001.

[27] Armin Rigo. Representation-based Just-in-time Specialization. April 2004.

[28] S. Subramanya Sastry and Rastislav Bodk. Program specialization for dynamic optimizers.

[29] V. C. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. pages 208–218, June 2000.

[30] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. *IVME03*, June 2003.

[31] E. Witchel, J. Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of ASPLOS-X*, Oct 2002.

[32] Byung-Sun Yang, Moon Soo-Mook, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioğlu, and Erik Altman. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation. *In International Conference on Parallel Architectures and Compilation Techniques*, October 1999.