Dynamic Optimization of Interpreters using DynamoRIO

by

Iris Baron

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

Author . . . .
Department of Electrical Engineering and Computer Science
June 27, 2003

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

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

Dynamic Optimization of Interpreters using DynamoRIO

by

Iris Baron

Submitted to the Department of Electrical Engineering and Computer Science
on June 27, 2003, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Interpreters are widely used in domain-specific, dynamic, and scripting languages. While their implementation is relatively simple, their performance is poor. In this thesis we improve the performance of interpreters using dynamic optimization techniques. This task is challenging because the information regarding the interpreted program is not known until the interpreter is run. Efficient implementations of interpreters (such as Just-in-Time compilers) require a runtime system that performs optimization dynamically, using program specialization. Such solutions incur high engineering cost. Furthermore, they are usually language-specific and hardware-dependent, and thus are not portable or reusable.

We create a single software dynamic optimizer that can be applied to different interpreters. Our system, DynamoRIO-with-LogPC is based on DynamoRIO's infrastructure, which is a dynamic optimization tool. While DynamoRIO is very efficient in a broad range of applications, typical interpreters' implementation foils DynamoRIO's strategy of trace collection. In addition, DynamoRIO's low-level optimization is unable to extract sufficient high-level semantics of the interpreted program to perform many of the necessary optimizations.

We provide simple annotations to the interpreters and modify the trace creation methodology of DynamoRIO such that its traces correspond to frequent sequences of code in the high-level program (the interpreted program), rather than in the interpreter. Then, we use standard optimization techniques such as constant propagation, dead code elimination, and partial evaluation to further improve the traces created. We also present two techniques to avoid unrolling native loops. Such loops might create too long traces that harm performance. Our experimental results on Kaffe and TinyVM interpreters show that DynamoRIO-with-LogPC outperforms DynamoRIO and is faster than the native program by 50-90%. While these results are not comparable with those of a hand-crafted native compiler, we believe that it provides an appealing point on the language implementation spectrum.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

First, I thank my advisor, professor Saman Amarasinghe for giving me the opportunity to work in the exciting area of dynamic optimization. I am grateful for his support during my studies. He always knew how to encourage me on both the professional and personal levels. A special thank is also due to Gregory Sullivan from the AI Lab for his time and invaluable comments. Our meetings were always fruitful and it was a pleasure working with him. In addition I want to thank him for providing the TinyVM interpreter that we used. I want to thank the DynamoRIO group members for their help: Derek Bruening, for guiding me in the maze of the DynamoRIO's code, Timothy Garnett for providing the basic optimizations techniques, and Vladimir Kiriansky for handing help when needed. I also want to thank Gleb Chuvpilo, Michael Karczmarek, and David Maze for their friendship.

I would like to thank my parents, Ella and Bruno for teaching me the value of hard work. Their continues support and understanding helped me to overcome the unavoidable obstacles of life. I thank my brother, Lior for the beautiful memories we share. Traveling together filled me with energy that carried me throughout these years. To my grandmothers, Omi and Buna, I thank for the motivation and encouragement they provided abundantly. To my parents in law - Bracha and Itzik, and Zvika and Hana, and to Meyrav and Mordechai I thank for their open ears, wise advices, and love. To my family, although you are far away, my heart and mind are always with you.

A very special thank is to my son Ethan and to my loving husband Opher. I thank Ethan, for the joy he brings into our life with each and every smile and cry. And most of all I thank Opher for sharing this exciting path in life with me through bad times and good times.

"Two roads diverged in a wood, and I -
I took the one less traveled by,
And that has made all the difference."

Robert Frost

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Objectives

Interpreters are widely used in domain-specific, dynamic, and scripting languages. Domain-specific languages, for example, are a very powerful problem solving mechanism. They simplify programming by providing a level of abstraction. This allows programmers to focus on the specific class of problem they are facing. Moreover, their high-level notion allows for semantic analysis and good debugger tools.

Building an effective compiler for these languages requires a huge effort; thus, we only see one or two new languages per decade with good compiler support. For this reason, domain specific languages are typically implemented using interpreters. Furthermore, many domain-specific languages include dynamic features such as dynamically dispatch calls, polymorphism and user-defined control structures that make it difficult or impossible for a compiler to perform static optimizations.

While interpreters' implementation is relatively simple, their performance is several orders of magnitude worse than a low-level compiled language like C. Thus, despite the domain-specific languages' appealing properties, they are far less common than we expect.

Efficient implementations of interpreters (such as Just-in-Time compilers) require a run-time system that performs optimization dynamically, using program specialization. These solutions incur high engineering cost. Furthermore, such implementations are usually language-specific and hardware-dependent, and thus not portable or reusable.

Thus, the motivation for this research is the need to optimize interpreters and VMs, without incurring the costs of developing an expensive custom implementation.

Our main objective is to create a single software dynamic optimizer that can be applied to different interpreters. Our system, DynamoRIO-with-LogPC is based on DynamoRIO's infrastructure with the appropriate changes to fit interpreters. DynamoRIO is a dynamic optimization tool, jointly developed at HP Labs and MIT. While it is very efficient in a broad range of applications, typical interpreter implementations foil DynamoRIO's strategy of trace collection. In addition, DynamoRIO's low-level optimization is unable to extract sufficient high-level semantics of the interpreted program to perform many of the important optimizations.

The design and development of DynamoRIO-with-LogPC includes three phases. In the first one, we modify DynamoRIO such that it will target the interpreted program rather than the interpreter for trace creation. In the second phase, we provide an interface for communicating high-level application information from the interpreter to DynamoRIO-with-LogPC. In the third phase, we implement additional optimization techniques to be performed on the generated traces in the new system.

We demonstrate the use of DynamoRIO-with-LogPC by improving the execution times of benchmarks on two very different interpreters: Kaffe and TinyVM. We consider improving the performances of these two VMs as the practical objective of this thesis. This will serve as a proof-of-concept for applying dynamic optimization in software to interpreters.

Our experimental results are encouraging. They suggest that DynamoRIO-with-LogPC has a high potential for improving performance of interpreters in general. Using simple instrumentation that required only a little familiarity with the interpreter and by changing/adding small percentage of the code (e.g., in Kaffe VM we changed 70 lines of code out of the 25,000 of line codes that deal with the interpreter), we improved native execution times of interpreters by 50-90%. While these results are not comparable with the ones of a hand-crafted native compiler, we believe that this system provides an appealing point on the language implementation spectrum.

Some of the results and methodologies of this thesis were presented at [47].

1.2 Thesis Overview

In this chapter we introduce the notions of compilers and interpreters and explain why interpreters are typically slower than compiled programs. We also present two, essentially different, examples for VMs: the TinyVM, which is a simple bytecode interpreter written in C (by Dr. Greg Sullivan of AI Lab at MIT), and Kaffe, which is an open-source VM for Java. In chapter 2 we survey different approaches to improve interpreters' performance. We explain each technique and point to relevant literature in that field. Among the techniques discussed in that chapter are: run-time code generation, dynamic compilation, and JITs. Chapter 3 discusses the dynamic optimization technique, and gives an overview of DynamoRIO, the dynamic optimizer infrastructure that we use. DynamoRIO identifies and collects traces, which are native sequences of frequently executed code, during runtime. Then it optimizes these traces, and emits them to the cache. This process improves the performance of many applications; however, it is not effective for interpreters and VMs, because it generates traces for the interpreter rather than for the interpreted program. These traces are typically short and not well utilized.

In chapter 4, we explain how we modified DynamoRIO to support optimization of interpreters and VMs. We designed DynamoRIO-with-LogPC so that it would target the interpreted application instead of the interpreter. The use of DynamoRIO-with-LogPC requires simple annotations to the interpreters. These annotations provide DynamoRIO-with-LogPC with the information regarding the high-level (interpreted) program's representation. Using theses annotations DynamoRIO-with-LogPC creates traces that correspond to frequently used code sequences in the application. These traces are longer and better utilized than the ones generated by DynamoRIO.

In chapter 5, we extend the optimization capability of DynamoRIO-with-LogPC. We use three classes of optimizations: "traditional" (used by DynamoRIO), avoiding too long traces, and functional memorization. Some of these optimizations may benefit from adding optional annotations to the interpreters. The first class is enhancing the traditional optimization techniques: constant-propagation, dead-code-elimination, and call-return-matching. To improve the efficiency of constant-propagation and dead-code-elimination, we use immutable regions and introduce trace-head constants (for both global and stack-allocated addresses). For the second class we first explain the need to avoid too long traces. Then, we propose two techniques to shorten traces: The first one is to constrain the number of basic-blocks in a trace to be smaller than a given maximum; The second technique is not to inline functions that contain loops. The third class of optimizations, partial-evaluation, uses a more aggressive optimization: for specific type of functions, it completely eliminates the function call and replaces it with the function's returned value.

Chapter 6 presents extensive experimental results from Kaffe and TinyVM interpreters, for both execution time and memory usage (trace length). We chose these VMs as they are very different in their characteristics. TinyVM is a simple stack VM, and Kaffe has a more complex internal representation. We compared the results of DynamoRIO, and DynamoRIO-with-LogPC with and without optimizations, to the native run time. For both VMs, DynamoRIO-with-LogPC and optimizations is by $30-80\%$ faster than the native run time.

These results show that performance improvement can be obtained with DynamoRIO-with-LogPC for small programs such as the TinyVM benchmarks, where native execution times are less than a minute, as well as for large programs such as the SpecJVM98 benchmarks, where native execution times are tens of minutes.

Finally, chapter 7 outlines possible directions for future research and summarizes this thesis.

In the next section we provide a basic background on interpreters and VMs.

1.3 Interpreters

Computers are programmed in multiple levels of abstraction. Low-level languages, such as assembly, are closer to the architecture and directly map to the instruction set of the host CPU. In contrast, high-level languages, such as C and Matlab, are closer to human languages. In high-level languages, each instruction represents one or more machine language instructions.

High level languages can be implemented using a compiler, an interpreter, or some combination of both. A compiler translates the high level code (*source code*) into a machine code (*target code*) that can be run later on. Running the program after changing the source code requires an additional compilation. Interpreters execute the high level program during runtime without creating a target code. This is known as a "one-level of interpretive indirection." Because of this interpretive indirection, interpreters are, typically, slower than target codes. Therefore, in this thesis we look for techniques to improve interpreters' performance.

The advantages of interpreters are threefold. First, they are easy to write and understand, because they follow the logic of the high level language, and they do not require the generation of a corresponding low level program. Thus, they are simple and easily portable. Second, because of their nature (executing the source program directly) interpreters, usually, have good error messages and include a good source-level debugger. Both help in the programming process. The third advantage is the dynamic flexibility of interpreters. Some languages, as Lisp, Prolog and Scheme, can add code to themselves or change the behavior of part of their code during runtime. The dynamic nature of such languages requires an interpreted implementation.

There are implementations that combine compilers and interpreters. First, a compiler translates the high-level code into an intermediate representation of the code's (*bytecodes*). Then, a *virtual-machine* (VM) executes this intermediate code, as an interpreter does. This allows the code to be written and compiled to an intermediate code by the programmer, and then to be executed using an appropriate VM on all the machines running the user's application. A concrete example of a language that is implemented in this form is Java. However, VMs, similarly to interpreters, suffer from an interpretive overhead.

Figure 1-1 (from [45]) shows the relations between a compiler, an interpreter, and a VM.

When running a program under an interpreter or a VM, one pays extra time for executing the interpreter-related logic, in addition to the "real work" of the application. This overhead is known as "interpretive overhead," and it is the main reason that interpreters are slower than target codes. In addition, compilers can afford to spend more time on optimizations, because the compilation is done only once and is not a part of the program's run time. As a result, compilers generate efficient target code.

Another reason for interpreters to be slower than target codes is that compilers can make

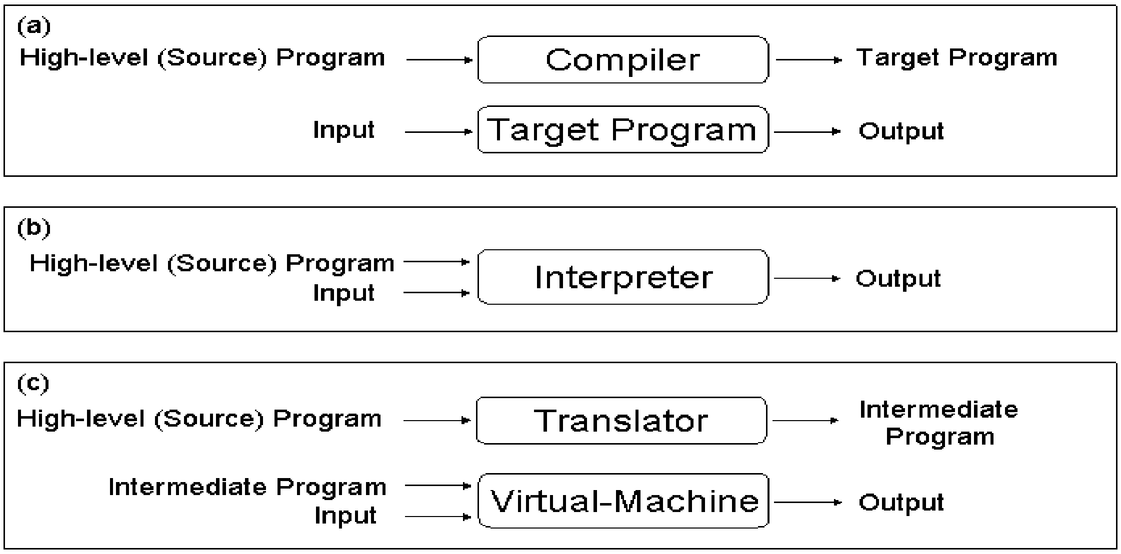

Figure 1-1: Schematic description of a compiler (a), an interpreter (b) and a virtual-machine (c).

early-binding decisions, and use them to generate efficient code. In contrast, interpreters use late-binding technique and analyze the declaration of variables each time the program runs. Moreover, interpreters analyze the declaration of local variables within subroutines every time the subroutine is called.

A "Pure Bytecode Interpreter" consists of a loop that increments the instruction pointer, fetches the next bytecode from memory, dispatches to the routine that implement that bytecode (involves some conditional control flow such as a switch statement), jumps to bytecode-specific instructions, and loops (increments the instruction point and jumps to the head of the loop).

Another typical form of interpreter implementation is the "read-eval-print" loop. It is used in many scripting languages and debuggers. The interpreter displays a prompt and is waiting for input. Upon receiving the required input, it reads an expression, evaluates it, prints the result and displays another prompt.

All these implementations suffer from the interpretive overhead. They can be improved using different optimization techniques. An overview of these techniques is given in the next chapter.

In the next subsections we give concrete examples of two interpreters: TinyVM and Kaffe.

In the rest of this thesis, we use both interpreters to demonstrate and examine the concepts we develop. Both interpreters are implemented using *switch-based* technique, derives from the pure bytecode interpreter.

1.3.1 TinyVM Interpreter

TinyVM is a simple VM written in C. Its main dispatch loop deconstructs the element of the bytecode vector *instrs*, using *pc* as an index into the array of bytecodes; then it finds the matching case in a *switch* statement, executes the relevant code, and jumps back to the head of the loop. Figure 1-2 illustrates the implementation of the interpreter's dispatch loop in TinyVM.

1.3.2 Kaffe VM

Kaffe is an open source Java VM written in C. Similar to TinyVM, the interpreter's implementation is a switch-based. Kaffe uses recursive tree-walking implementation for its Java method invocation. Each method invocation, which is not a native method is loaded by the *VirtualMachine* function and then executed by the *runVirtualMachine*. The interpreter dispatch loop is an infinite loop which executes the bytecode sequentially, until reaches one of the return-from-method bytecodes. The cases are structured in the kaffe.def. They use macros of the instruction definition, which are defined in icode.h file. Figure 1-3 illustrates the implementation of the Kaffe interpreter's dispatch loop for the *IFEQ* bytecode.

```
loop:
  op = instrs[pc].op;
  arg = instrs[pc].arg;
  switch (op) {
...

  case CALLOP:
    ... setup new call frame ...
    pc = arg; /* go to start of function body */
    goto loop;

  case RETOP:
    ... clean up stack ...
    pc = pop_raw();    /* pop the return PC */
    *(++sp) = val;     /* put return value back on stack */
    goto loop;

  case BEQOP:
    n = pop_int();     /* top must be an int */
    if (n) {           /* not equal 0 */
      pc++;            /* just continue */
    } else {           /* equal 0 */
      pc = arg;        /* do the branch */
    }
    goto loop;
  ...
 }
```

Figure 1-2: TinyVM's dispatch loop.

in machine.c:

```
void runVirtualMachine(methods *meth, slots *lcl, slots *sp, uintp npc,
       slots *retval, volatile vmException *mjbuf, Hjava_lang_Thread *tid)
{
   bytecode *code = (bytecode*)meth->c.bcode.code;
   ...

   for (;;) {
      register uintp pc = npc;
      assert(npc < meth->c.bcode.codelen);
      mjbuf->pc = pc;
      npc = pc + insnLen[code[pc]];

      switch (code[pc]) {
      default:
         dprintf("Unknown bytecode %d\n", code[pc]);
         throwException(NEW_LANG_EXCEPTION(VerifyError));
         break;

#include "kaffe.def"
      }
      }
end:
      return;
}
```

****

in kaffe.def:

```
...
define_insn(IFEQ)
{
   ...
   idx = (int16)((getpc(0) << 8) | getpc(1));
   ...
   begin_sync();
   cbranch_int_const_eq(rstack(0), 0, reference_code_label(pc+idx));
   pop(1);
   end_sync();
}
```

****

in icode.h: (macros)

```
...
#define cbranch_int_const_eq(s1, s2, w) if ((s1)[0].v.tint == (s2)) { w; }
#define reference_code_label(l) npc = (l)
...
```

Figure 1-3: Kaffe's dispatch loop, and the IADD bytecode implementation.

Chapter 2

Optimizing Interpreters

In this chapter we present different approaches for improving interpreters' performance. Section 2.1 describes ad-hoc optimization for more efficient implementation of interpreters. Another type of optimization is program specialization, as described in section 2.2. This area includes many techniques; we present a few of them, such as run-time code generation and dynamic compilation. Dynamic translators and Just-In-Time compilers can also be considered program specializers. Due to their popularity, we dedicate section 2.3 to them. Finally, in section 2.4 we mention the dynamic optimization technique that is further described in the next chapter.

2.1 Ad-hoc Bytecode Interpreter Optimization

An interpreter's performance is directly correlated to its implementation. The pure bytecode interpreter described in section 1.3, suffers from large overhead: each bytecode implementation involves at least two memory references and two jumps, aside from the dispatch lookup (the switch). This overhead is more substantial when executing simple bytecodes. An example of such an interpreter is seen in Figure 2-1 (copied from [41]).

To overcome these problems, *threaded code* was first used as part of the Forth programming language, described by Moore and Leach [39], and then used as *direct threading* in [19]. The direct threaded code technique improves performance by removing the dispatch lookup

```
compiled code:
unsigned char code []={...,
        Bytecode_push3,
        Bytecode_push4,
        Bytecode_add,...};
bytecode implementations:
unsigned char *instructionPointer=code-1;
for(;;) {
        unsigned char
             bytecode=*++instructionPointer;
        switch(bytecode){
        /*  . . .  */
        case bytecode_push3:
                *++stackPointer=3;
                break;
        case bytecode_push4:
                *++stackPointer=4;
                break;
        case bytecode_add:
                --stackPointer;
                *stackPointer+=stackPointer[1];
                break;
        /* . . .  */
        }
}
```

Figure 2-1: Pure bytecode interpreter.

and eliminating the centralized dispatch. In direct threading, each bytecode represents the implementation address. Thus, the dispatch involves fetching the bytecode and jumping directly to its implementation address; Furthermore, each direct threaded bytecode implementation ends with code that dispatches to the next bytecode. This procedure eliminates the need for a centralized dispatch. Figure 2-2 (copied from [41]) displays a direct threaded code for the bytecode interpreter from Figure 2-1.

To obtain *direct* threaded code, we can dynamically translate bytecodes into their direct threaded representation, before executing the code. The translation process involves a loop that reads the bytecode, looks up its implementation address, and creates the direct threaded code. An example for this is given in Figure 2-3 (copied from [41]).

Dynamic translation to threaded code allows additional bytecodes to exist. Moreover, by specifying different implementations for the same bytecode ("overloaded" bytecodes) during translation time, it also allows the creation of a "macro" bytecode from a common sequence of bytecodes (during execution time). In addition, this technique removes the dispatch

```
compiled code:
void *code[]={... ,
        &&opcode_push3,
        &&opcode_push4,
        &&opcode_add,...};
opcode implementations:
/*dispatch next instruction*/
#define NEXT() goto **++instructionPointer

void **instructionPointer=code-1;
/*start execution: dispatch first opcode */
        NEXT();
/*opcode implementations...*/
opcode_push3:
        *++stackPointer=3;
        NEXT();
opcode_push3:
        *++stackPointer=4;
        NEXT();
opcode_add:
        --stackPointer;
        *stackPointer +=stackPointer[1];
        NEXT();
        /* . . . */
```

Figure 2-2: Direct threaded code.

overhead, improves code locality, and permits inter-bytecode optimization by the compiler. Dynamic translation to threaded code was used in Squeak, a portable "pure bytecode" implementation [23], in BrouHaHa, a portable Smalltalk VM [37], [7], and in Objective Caml [34].

An extension to this approach is to dynamically find the best choice of common sequences of bytecodes to translate. It is done by selective inlining during the translation of bytecode to threaded-code or as a separate pass over the threaded code. This technique was introduced by Piumarta and Riccardi [41] and results in very good performances for any application.

2.2 Program Specialization

Program specialization is a program optimization technique that is also known as "partial evaluation". We first explain the meaning of it in the general context, and then demonstrate

```
traslation table:
void *opcodes[];
/* . . . */
&&opcodes[bytecode_push3]=&&opcode_push3;
&&opcodes[bytecode_push4]=&&opcode_push4;
&&opcodes[bytecode_add]=&&opcode_add;
/* . . . */
dynamic translator
unsigned char *bytecodePointer=firstBytecode;
Void **opcodePointer=translateCodeFunction;
While (moreBytecodesToTransalte)
      *opcodePointer++=opcodes[*bytecodePointer++]
```

Figure 2-3: Dynamic translation of bytecodes into threaded code.

its use in optimizing interpreters. Standard references to the subject of partial evaluation are [25] and [24].

Partial evaluation's basic definition is as a source-to-source transformation, i.e. program P and part of its input data, $in1$, are transformed into a new specialized program, P_{in1}, using a partial evaluator (PE), *mix*. Running P_{in1} on the rest of the input data should yield the same results that P yields when running on all of the input data.

$[[P]][in1,\ in2] = [[\ [[mix]][P, in1]\ ]]\ in2 = [[P_{in1}]]\ in2$

In other words, all calculations that involve only $in1$ are pre-computed in P; however, new code is generated for calculations that depend on other input data. P is said to be specialized with respect to $in1$. Figure 2-4 presents a schematic partial evaluator.

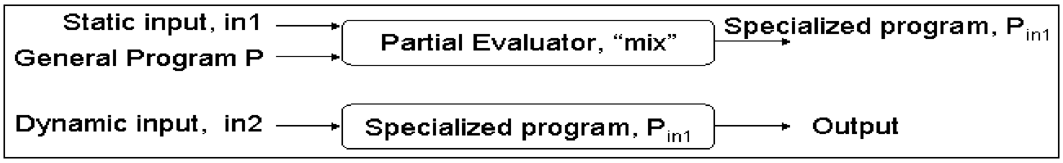

Figure 2-4: Schematic partial evaluator.

Programs that might benefit from partial evaluation are:

1. Highly parameterized programs: In such programs a long time is spent inspecting parameters which are often the same.

2. Similar subcomputations: Programs that can benefit from computing a common part only once and use it again when needed.

3. Highly interpretive programs (such as interpreters and simulators): By specializing the interpreter with respect to the program being run, the interpretive overhead can be removed. In simulators, specializing with respect to the object being simulated eliminates the object's scanning time.

4. Database query search algorithms: We can specialize the program with respect to often used queries.

The rationale behind such specializations is that a more efficient code can be generated, as the level of available information increases. Conceptually, partial evaluation is simple, and therefore it was implemented in many areas, such as computer graphics, Fast Fourier Transformations (FFT), circuit simulation, compiler and compiler generator, pattern matching, and more. Partial evaluators in practice started in the mid-70s with partial evaluators for Lisp [26] and Prolog [31]. Today they are used for many different languages, such as Scheme (Similix [6] and Schism [12]) and C (C-mix [2] and Tempo [13]).

The notion of partial evaluation can be used with interpreters to specialize them with respect to a specific program. This technique is called "compiling interpreters". Many of the practical solutions, such as dynamic code generation and dynamic compilation, rely on this technique. Another use of partial evaluation in the context of interpreters is "generating compilers and compiler-generators" from the interpreter. In the next subsections we briefly review these techniques.

2.2.1 Compiling Interpreters

Assuming there is an interpreter, *int*, and a source program, P. The result of running partial evaluator, *mix*, on these two is a target code:

$target = [[mix]][int,\ P]$.

The *mix* function takes a source program and an input and produces a specialized version of the source program (in our case, the interpreter). The target program is a specialized interpreter, in the language that the interpreter is written in.

We expect a specialized interpreter to be faster than the original one when running on P, because the specialization removes any unnecessary overhead (including the interpretive overhead). Yet, we do not expect the specialization to reach the performance of a well-designed compiler, because it cannot perform optimizations based on the semantics of the interpreted code (the input, P).

There are two types of partial evaluators that can be invoked at compile-time; online and offline. The online partial evaluator takes as input the program and the subset of input data that is static; then, by doing "symbolic computation" of the available data, it obtains a specialized program. The offline partial evaluator, has two steps. In the first one, it performs binding-time analysis in order to determine which part of the input is static (known) and which part is dynamic (unknown). In the second step, the offline partial evaluator uses the real specialization values to generate the specialized program. The result is source code for the specialized program, which may be compiled to obtain an executable program.

In dynamic languages the specialization values are not known at compilation time (this is also the reason that it is difficult for a compiler to perform optimizations statically). This constrains the effectiveness of offline partial evaluators, and motivates the use of run-time specialization. Run-time specializers do not perform source-to-source transformation, rather they produce executable code from the specializations. They are based on offline specialization, but the real values are filled in at runtime. Run-time specializers are widely used in run-time code generation, dynamic compilation and automatic specialization. Next, we describe the use of these techniques and explain how they can be used in relation with interpreters.

Run-Time Code Generation

Run-time code generation (also called "Dynamic code generation") is a process in which new code is added to the instruction stream of an executed program. It usually uses run-time information for generating specialized code. This type of information is not available statically; thus, it cannot be used during the compilation process. Keppel et al. [30] gives an example for the potential of run-time code generation. They considered an interactive program, in which the user enters an expression that is to be evaluated once for each item in a data set (such as matching regular expression with lines in a file). Using dynamic code generation, we can dynamically compile the expression into a special-case function and then use this function for each input data in the datum.

Dynamic code generation has been used to improve performance in many areas: operating systems, dynamically typed languages, simulators, matrix manipulation and compiling interpreters. Here are additional examples for its use:

1. Hardwiring infrequently changing runtime values into the executable program: this allows for constant-propagation, constant-folding, removing branches that depend on these values, and removing unnecessary load time. Of course, it should be regenerated each time that these values change [22, 43].

2. Matrix manipulation: programs can be specialized with respect to the matrix's characteristics, e.g. in multiplication of sparse matrices [20].

3. Compilation of small languages (such as query languages): we can benefit from dynamically compiled queries, in which the program is specialized with respect to some fields, values, and boolean operators. This improves performance, especially in large databases [9].

4. Frequently interpreted parts of the program: these parts can be compiled and directly executed. Interpreters are a perfect application for dynamic compilation in which the interpreted program is a runtime constant [15],[35].

In [15], a run-time code generation implementation for smalltalk-80 bytecode VM was described. This implementation dynamically collects a sequence of VM instructions, compiles them to native code while performing simple optimizations, and caches them. This technique showed better performance than the ad-hoc optimizations described in section 2.1. Another application that uses run-time code generation with interpreters is SELF, presented in [10].

Some dynamic code generators have a specific language that is used to write programs that generate dynamic code at runtime. Such languages provide high-level notations that allow users to write programs that manipulate code.

DCG [18] provides an interface for the programmers to manually construct routines. It uses the intermediate representation of the LCC compiler to dynamically generate code.

These routines are compiled by DCG and result in an executable code, which can be invoked by an indirect call. Then, at runtime, the compiled code is executed, and a fast code generator is invoked to compile the new code. In order to save time, the dynamic code generator used in DCG, is very simple; it provided no instruction scheduling or elaborated register allocation.

'C (Tick C), developed by Engler et al. [17], originally used DCG as its dynamic code generator. It provides an extension of the C language (extending the ANSI C library), to support dynamic code generation. The programmer uses the extended library to write programs that explicitly manipulate, compose, and compile program-fragments at runtime. These programs look like fragments of C code, which eases the programming burden. In addition, 'C provides an opportunity to perform static analysis, in order to reduce the overhead of generating code at runtime. After the introduction of VCODE [16], which is (more than an order of magnitude) faster than DCG, 'C was re-implemented using VCODE as its dynamic code generator [42]. VCODE is a template-based system for dynamic code generation that provides a similar interface to the one of an idealized RISC architecture. VCODE also generates code in-place, without representing the code in data structures. Thus, the resulting dynamic compiler made 'C a very fast and efficient dynamic code generator. In addition, the target code it produces is highly efficient.

The run-time code generation implementations described above allow programmers to manually program the dynamic code generation. This is known as an imperative/operational approach. On the one hand, it is controllable and flexible. On the other hand, it increases the programming effort. It also requires the programmer to well understand the code generation process and therefore it is error-prone.

Dynamic Compilation

Another approach to run-time specialization is the declarative approach, in which users declare (by annotation) guidelines for the dynamic code generator. The term "dynamic compilation" specifies the part of the compilation done during the execution of the program. Similar to run-time code generation, dynamic compilation systems also perform run-time specialization, in order to take advantage of information available at runtime. But unlike run-time code generation, which is written manually, these systems create templates that are automatically filled at runtime, using only hints from the programmer.

Dynamic compilation simplifies the programmers' part and puts the complexity at the dynamic compiler's end. Thus, the main focus of dynamic compilation systems is on reducing time spent in the dynamic compiler. This is done by performing extensive offline pre-computations, and, similarly to run-time code generation systems, by restricting the optimization performed by the dynamic compiler.

This technique is sometimes called "deferred compilation". Leone and Lee [33] describe a "lightweight" approach for dynamic code generation, where compilation of some parts of the program is postponed until runtime, to better optimize programs.

An annotation-based dynamic compilation system for C, was presented by Auslander et al [4]. This system defines annotations that indicate parts of the program that should be compiled dynamically (*dynamic regions*) and source variables that remain constant during the execution of the dynamic regions. This system is sometimes referred to as "selective dynamic compilation" since it performs run-time specialization only on a portion of the code. Its goal is to move much of the burden of dynamic compilation to the static compilation phase. This system is a template-based system that contains a static compiler and a dynamic one. During static compilation the dynamic regions are analyzed based on the programmer's annotations. This analysis determines which variables and data structures would be invariable (constants) at runtime. The static compiler generates set-up code that calculates values of the derived runtime constants. It also creates pre-compiled machine-code templates, whose instructions contain holes that would be filled with runtime constant values. Then, the static compiler generates directives. These directives instruct the dynamic compiler how to produce executable code from templates and computed constants of the set-up code's (e.g., to copy blocks and update offsets of branches). The static compiler also performs optimizations on the templates it created.

The dynamic compiler needs only to follow the directives to instantiate the machine-code templates. It runs the set-up code once to get the values of the constants, fills in the holes based on labels, and executes all of the directives. Figure 2-5, taken from [4] describes this schematic system.

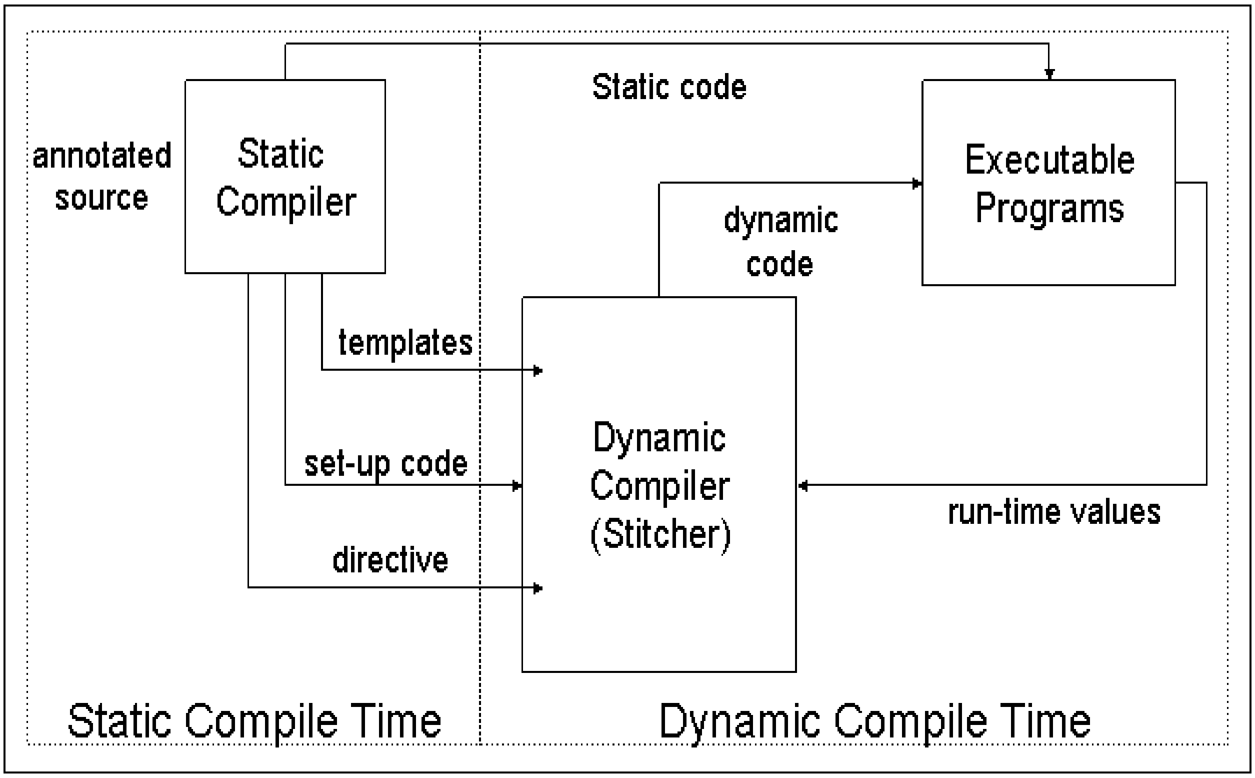

Figure 2-5: A schematic description of a dynamic compilation system.

During execution the specialized code can be reused if the runtime values match those that were used to specialize it. If a dynamic region is invoked with different runtime values, multiple versions of the specialized code can be stored and maintained. Each time the execution enters a dynamic region, a *dispatcher* selects the appropriate specialized version, based on the runtime values, or invokes the dynamic compiler to produce a new specialized version.

The declarative approach leads to a shorter dynamic compilation time in comparison to imperative/operational approach. However, since the correctness of the specialization depends on the annotations, it is still error-prone. Moreover, it is less flexible and controllable. An improved system, based on the system described above, is DyC presented by Grant et al. [21]. In this system, the annotations are more expressive, flexible, and controllable, while the core system is similar to its predecessor [4].

Automatic Specialization

It is hard to predict the performance gain from a run-time specializer. Therefore, using manual annotations requires programmers to perform a trial-and-error loop in order to tune the choices of dynamic regions and runtime variables. Automatic specialization replaces the manual annotation presented earlier with automatic system that infers annotation based only on a description of the known inputs. This technique is more reliable, because it requires little or no modification of the interpreter.

Calpa [38] is an automatic generator annotation for DyC that combines program analysis and profile information. It first creates an instrumented version of the program that generates and summarizes value and frequent data. Then, Calpa runs this version on some representative inputs, yielding an execution profile for that program. An annotation selection tool analyzes all opportunities for run-time specialization in terms of cost versus benefit and chooses the optimal ones. Then, Calpa runs DyC, which was mentioned before. Other automatic program specialization are C-Mix, developed at DIKU [2], and TEMPO, developed at IRISA [13].

2.2.2 Generating Compilers and Compiler-Generators

Program specialization can be further extended to generate a compiler from an interpreter [27] and [28]. It involves a self-application that uses *mix* to specialize itself. In the context of interpreters, this creates a compiler, which is a specialized version of the interpreter:

$$compiler = [[mix]][mix,\ int].$$

Now, the compiler is applied to the source program P, and generates the target:

$$target = [[mix]][int,\ P] = [[\ [[mix]][mix,\ int]\ ]]\ P = [[compiler]]\ P$$

Using the same concept we can define a compiler generator. This would be a program that transforms interpreters into compilers:

$$
\begin{aligned}
&target = [[P]][in1,\ in2] = \\
&= [[\ [[mix]][P,\ in1]\ ]]in2 = \\
&= [[\ [[\ [[mix]][mix,\ P]\ ]]\ in1\ ]]\ in2 = \\
&= [[\ [[\ [[\ [[mix]][mix,\ mix]\ ]]\ P\ ]]\ in1\ ]]\ in2 = \\
&= [[\ [[\ [[compiler - gen]]\ P\ ]]\ in1\ ]]\ in2
\end{aligned}
$$

2.3 Dynamic Translation and JITs

The basic idea of the dynamic translation field is to translate frequently executed code into native code during runtime; and then cache this native code. As a result, further references to this code can be executed directly from the cache. This technique was pioneered by the commercial smalltalk implementation (in 1980), currently known as VisualWorks. Its main use was in non-native system emulations. This type of use is called "binary translation." Today, this technique is named Just-In-Time (JIT) compilation and it is widely used in implementations of the Java Virtual Machine (JVM). In this section we present the evolution of JITs. We focus on JIT implementations for Java, since Java is the most popular object-oriented programming language, and it is, typically, implemented by a VM (JVM).

The first generation of JVMs, included simple interpreters that executed bytecodes sequentially. Their performance was very poor, because of the interpretive overhead. The calls for virtual and interface methods are time consuming, because they require a method lookup. Moreover, in object-oriented programs, there are, usually, many small methods, and frequent calls to static methods (which do not require method lookup). This prevents intra-procedural optimization by a compiler. Another source of overhead is safety checks and exception checks for accesses to arrays and instance variables.

To overcome the JVMs' poor performances two compiler solutions were suggested. The first one is to compile statically all bytecodes into native code. While this solution allows additional static optimization (at static compilation time), it does not support dynamic class loading. Thus, it cannot be implemented for the Java language. The second solution is the introduction of a JIT compiler/translator. This JIT compiler translates bytecodes into native code when a new method is invoked at runtime. This solution is so popular that it is considered to be the second generation of Java. Java JITs have evolved over time and have become very complex, performing most of the traditional optimization techniques used in static compilers. In addition, they involve many optimizations that are tailored for the Java language. The following is a non-exhaustive list of optimization techniques that are used in JITs' implementation:

1. Method call optimization: This technique uses inlining of method calls. The inlining removes the overhead of the method invocation and further increases opportunities for additional optimizations (by creating large inlined blocks of code). While this is easy to implement for static method calls, it is very difficult for dynamic ones. The reason for this difficulty is that in dynamic object-oriented languages (as Java) there may be a number of implementations for a single method. The naive solution of inlining all implementations of the method that may be called at runtime is not practical; because such a solution will result in an exploding code size. A more practical solution is to perform resolution of the dynamic method calls; this is usually done by profiling. HotSpot [36], for example, gathers information about the program's behavior, including the method implementations that were called. Thus, it can inline only those that are used. Inlining dynamic method calls saves a lot of time in Java, since Java has frequent dynamic method invocations.

2. Exception check elimination: This technique reduces the overhead of exception checks. One way of doing this is by removing the array bounds check, required by the Java language specification. In cases when the JIT can prove that the index to the array is always within the correct bounds or that an earlier check would throw an exception.

3. Type inclusion test: This technique simplifies the type inclusion test needed in Java to support down-casts. One approach is to encode the class hierarchy in a small table. This approach requires both space and time (re-computation is required when classes are loaded and unloaded dynamically). Another approach is to inline code to test the most-frequently occurring cases, such as NULL objects, cases where the class of the referenced object is identical to the required class, etc.

4. Register allocation: This technique is used during the code generation phase of the

JIT. It allocates local variables to physical registers, for both temporary variables of the JIT's computation and local variables within a single method. While this technique is well known in static compilers, it is hard to implement when there is time constraint, as in JITs.

5. Common subexpression elimination: This technique reduces redundancies by looking for identical subexpressions that compute the same value.

Despite improved JITs, there is a room for additional improvement to JITs' performance. The basic idea is that only a small fraction of the code (10-20%) consumes most of the execution time (80-90%). Thus, running the traditional JIT does not yield the optimal performances, because part of the time is spent in an unnecessary dynamic compilation of code. The third generation of Java solves this problem, by performing compilation only on the most frequently executed code. HotSpot, that was mentioned above, was the first one to implement this idea by detecting "hot spots" in the code and performing dynamic compilation to these hot spots. This selective compiling is known as "adaptive optimization". It is also used in JITs for other object-oriented programming languages, such as SELF [10].

The Jalapeno JVM [3] uses an advance form of adaptive optimization. It utilizes idle processors in a Symmetric Multi-Processing (SMP) system to optimize code at runtime. Jalapeno initially performs a low-level optimization of all the code; then, it embeds profiling information and uses it to trigger re-optimization of frequently executed code at higher levels [32].

The main disadvantage of JITs is the enormous engineering effort they require. Additional disadvantages are that JITs are very specific to the VM and that they are hardware-dependent (as a traditional compiler). Thus, they are not easily portable to multiple architectures.

2.4 Dynamic Optimization

Unlike dynamic compilation or dynamic translation, dynamic optimization is not concerned with translating code or compiling it. It takes as input a compiled code (executable code) and optimizes it during runtime. Currently there are hardware and software dynamic optimizers. Their success in improving performance depends directly on the behavior of the program to be optimized.

We identified the potential of running programs using dynamic optimizers, and explored running interpreters and VMs on DynamoRIO, which is a software dynamic optimizer. In the next chapter, we introduce DynamoRIO and further discuss the dynamic optimization technique.

Chapter 3

Dynamic Optimization and DynamoRIO

Dynamic optimization has many applications, but is especially worthwhile for applications with a high degree of dynamism. Today, dynamic language features, such as reflection and dynamic class loading are beginning to appear in mainstream programming languages, such as Java. These dynamic features make static optimization substantially more challenging, or even impossible. We believe that dynamic optimization will facilitate and promote the use of high-level, domain-specific, dynamic languages, by improving their performance.

Other optimization techniques, which were described in chapter 2, require a massive effort in research and development of program analysis and optimization. Such an effort is unattainable for most domain-specific languages. We propose the use of a common dynamic optimization framework to improve performance of interpreters and VMs. We investigate this idea by using DynamoRIO as the dynamic optimizer.

Dynamic optimization is a technique in which optimization is done on the native code during execution time, without any intervention from users.

Hardware dynamic optimization of an instruction stream is performed in superscalar processors. The Trace Cache [44] allows such optimizations to be performed off of the critical path.

API-less software dynamic optimization systems include: Wiggins/Redstone [14], which employs program counter sampling to form traces that are specialized for a particular Alpha machine; Mojo [11], which targets Windows NT running on IA-32, but has no available information beyond the basic infrastructure of the system; Dynamo [5] for PA-RISC, and DynamoRIO which targets X86 architecture.

This chapter's outline is as follows: section 3.1 presents the DynamoRIO dynamic optimizer and explains the dynamic optimization technique; section 3.2 explains the main problems in using DynamoRIO with interpreters, and the approach we take in order to solve them.

3.1 DynamoRIO

Our optimization infrastructure is built on a dynamic optimizer called DynamoRIO. DynamoRIO is based on the IA-32 version [8] of Dynamo [5]. It is implemented for both IA-32 Windows and Linux; and it is capable of running large desktop applications.

DynamoRIO was design to observe and potentially manipulate every single application instruction prior to its execution. The simplest way to do this is with an interpretation engine. However, interpretation via emulation is slow, especially on an architecture like IA-32 with a complex instruction set. Thus, DynamoRIO uses a typical trick to avoid emulation overhead; it caches translations of frequently executed code, so they can be directly executed in the future.

DynamoRIO copies *basic blocks* (sequences of instructions ending with a single control transfer instruction) into a code cache and executes them natively. At the end of each block, the application's machine-state must be saved and control must be returned to DynamoRIO (a *context switch*) to copy the next basic block. If a target basic block is already present in the code cache, and it is targeted via a direct branch, DynamoRIO *links* the two blocks together with a direct jump. This avoids the cost of a subsequent context switch.

Indirect branches cannot be linked in the same way because their targets may vary. To maintain transparency, original program addresses must be used wherever the application stores indirect branch targets (for example, return addresses for function calls). These addresses must be translated into their corresponding code cache addresses in order for the

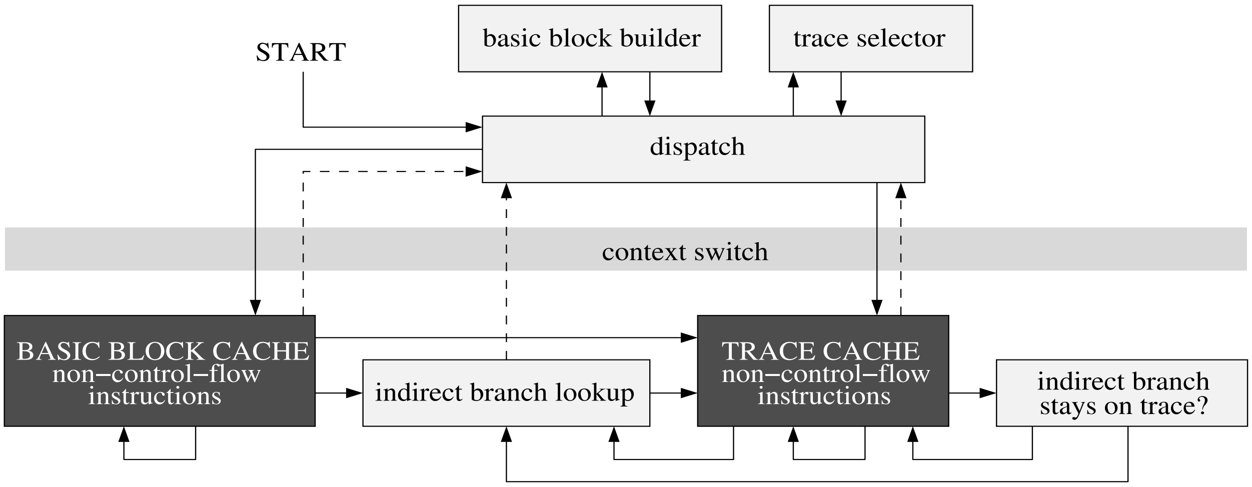

Figure 3-1: Flow chart of the DynamoRIO system. Dark shading indicates application code.

code to continue at the right location. This translation is performed as a fast hashtable lookup.

To improve the efficiency of indirect branches and to achieve better code layout, basic blocks that are frequently executed in a sequence are stitched together into a unit called a *trace*. When an indirect branch occurs within a trace, a check is inserted to ensure that the actual target of the branch will keep its execution in the trace. This check is much faster than the hashtable lookup, but if the check fails the full lookup must be performed. The superior code layout of traces goes a long way toward amortizing the overhead of creating them and often speeds up the program [5, 44].

A flow chart showing the operation of DynamoRIO is presented in Figure 3-1. The figure focuses on the flow of control in and out of the code cache, which is shown at the bottom portion of the figure. The copied application code looks just like the original code with the exception of its control transfer instructions, which are shown with arrows in the figure.

3.2 DynamoRIO and Interpreters

An important heuristic, upon which DynamoRIO's trace identification and collection strategy is based, is that popular targets of jumps are good candidates for *trace heads*. Furthermore, it is assumed that most of the time execution will proceed most or all of the way through a trace. In other words, recording long instruction streams starting at common branch targets will result in an execution code that will spend most of its time in recorded traces.

Unfortunately, typical interpreter implementations foil DynamoRIO's trace collection heuristic. In the case of a simple read-switch-jump loop, the interpreter's hottest loop head is in fact a poor candidate for a trace head, as its body follows a different path for each byte code type. This requires an indirect lookup, which is an expensive operation in DynamoRIO.

Threaded interpreters pose a related problem for DynamoRIO. The pervasive use of indirect jumps for control transfer foils DynamoRIO's trace head identification heuristics. Furthermore, even if a trace head were to be identified, it does not uniquely identify a commonly occurring long sequence of native instructions.

Using DynamoRIO with interpreters does not yield a performance improvement. Rather, it shows a slow-down in most of the tests. (The environment settings and the tests performed are described in more detail in chapter 6.) The reason for this slow-down is that DynamoRIO is a low-level optimizer. It has no knowledge about the high-level semantics of the interpreted programs, which is necessary to perform efficient optimization.

We suggest a new approach that is more suitable for interpreters. This approach is based on a modified version of DynamoRIO. While DynamoRIO is looking to record a long, frequent sequence of native instructions, our new approach aims to record the native instructions corresponding to a long, frequent sequence of bytecodes (or high-level representation). This approach is a combination of native JIT compiler and partial evaluation techniques.

Our approach requires the addition of a small amount of instrumentation to the interpreters; and so is not completely automatic. However, we show that it yields better performance than DynamoRIO alone. The next chapter describes the basic design of this new system.

Chapter 4

DynamoRIO-with-LogPC

As we explained in the previous chapter, DynamoRIO has some limitations when running with interpreters; thus, it does not achieve good performance for such applications. In this chapter we describe the design and implementation of DynamoRIO-with-LogPC that is based on DynamoRIO and aims to address these limitations. We modify DynamoRIO such that it will identify frequent executed sequences of bytecodes in the interpreted program (rather than frequent executed sequences of native instructions in the interpreter).

DynamoRIO looks at a one piece of data, the Program Counter (PC), as the key to the code cache. This address specifies the exact location of the executable code (the interpreter's code in our case). DynamoRIO's decisions about collecting and emitting traces, are based on Control Transfer Instructions (CTI). These are native instructions that change the native PC (e.g., branches, jumps and calls). We are looking for analogous notions for the location in the interpreted program and the program's change of flow. As we explain in this chapter, this can be achieved with some modifications to DynamoRIO. The basic modifications are to add instrumentation that provides this information to DynamoRIO. Using this information, we tuned the functionality of DynamoRIO to create traces that correspond to sequences in the interpreted program.

This chapter's outline is as follows: Section 4.1 describes the modification to DynamoRIO required for supporting the instrumentation interface; Section 4.2 defines *Abstract PC*, and *Logical CTIs* and describes the essential changes in DynamoRIO for handling information regarding the high-level application. Section 4.3 describes the mechanism for selecting, collecting, and emitting traces. Section 4.4 explains the linking mechanism that is used. Finally, section 4.5 summarizes this chapter.

4.1 Annotations

In order to optimize an interpreted program, our system needs to know about the program being interpreted, not just the interpreter itself. Our first modification to DynamoRIO is to introduce an interface whereby higher-level information can be communicated to DynamoRIO's back end. This interface includes a set API functions. These functions should be hooked into the interpreter's code. And the interpreter should be linked to the DynamoRIO's library.

We chose to use API functions as the method of passing information to DynamoRIO, because its main functionality is to interpret the instruction stream on the fly. Thus, it can easily support the instrumentations and perform them with a low overhead.

DynamoRIO keeps a hard-coded list of the addresses of these functions. When it encounters a *call* instruction, it reads its target address. If this address matches any of the instrumentation functions' addresses in its list, then this call is not inlined, and is included in the basic-block. When the control flow reaches this basic-block, the function is called and executed (as a regular assembly instruction). Every *call* instruction adds some overhead to the system; however, we show that the performance improvement more than amortize this cost.

Remark: in all the API functions that we describe in this chapter we suppress the "dynamorio" prefix from the name of the functions. For example, set_logical_pc stands for dynamorio_set_logical_pc.

4.2 Abstract PC and Logical CTI

In this section, we describe the high-level information that needs to be passed to DynamoRIO. We define three new terms: "abstract PC", "logical PC" and "logical CTI", and describe the instrumentation provided for passing the required information to DynamoRIO.

We use "logical" to refer to the high-level interpreted program. Thus, "logical PC" is a unique identifier into the control flow structure of the interpreted application. As an analogy to the notion of PC in the original DynamoRIO we define "Abstract PC;" it is a pair of "native PC" and a "logical PC." This pair uniquely identifies the current overall computation point. In TinyVM (described in section 1.3.1), the logical PC is an offset into the bytecode stream that represents the interpreted program. In Kaffe's VM (described in section 1.3.2), the exact location in the interpreted program is uniquely defined by the method being executed (the pointer to its bytecodes) and the offset to the method's bytecode array (integer number of bytes). Thus, we used the sum of them to represent the logical PC passed to DynamoRIO-with-LogPC.

Remark: Neither the logical nor the native PC is sufficient for defining the location in the interpreted program. A native PC, for example, that corresponds to the start of the handler for a *CALL* bytecode, would be executed for each call site in an interpreted application. However, the logical PC might stay constant over a significant amount of interpreter execution (consider the interpretation of the *InvokeVirtual* JVM instruction). Thus, we need a pair of native PC and logical PC to uniquely specify the location.

Analogous to the native control transfer instructions (CTIs) are the logical CTIs. A Logical CTI is a change in the sequential flow of the interpreted program. This change corresponds to a change in the state of the interpreter. Examples of Logical CTIs are the *CALL*, *JUMP* and *RETURN* bytecodes.

DynamoRIO-with-LogPC provides the interpreter writer with an API function to pass the logical PC, **set_logical_pc(logical_PC)**. It also provides two API functions for identifying the different types of logical CTI:

logical_direct_jump() corresponds to a direct native jump. By calling logical_direct_jump() at a particular point in the interpreter, the interpreter writer guarantees that every time the interpreter reaches this point with the logical PC equal to its current value, it will make the logical control transfer to the same logical PC target. An example to a direct logical control transfer is the *JUMP* bytecode.

logical_indirect_jump() corresponds to an indirect native jump. In this case, the interpreter cannot guarantee the requirements for a logical_direct_jump. A classic case of an indirect logical control transfer is a *RETURN* bytecode; where the actual target is based on runtime data (e.g. a value on the stack), rather than a compile time constant.

Calls to these API functions enable DynamoRIO to identify potential *trace heads* and to link traces together. While calls to logical_set_pc provide only the logical PC value, DynamoRIO needs the exact location in the program's flow (the abstract PC). In order to obtain this information, DynamoRIO provides the native PC value (of the instruction that follows the call to the logical PC API function), while processing logical_*_jump calls. In addition, it adds the global DynamoRIO context and the basic-block's linkstub to the call (to get "from-data" information). Both the *DynamoRIO context* and the *linkstub* are used to identify trace heads, and to link traces. These functionalities are described in the next sections. The translation to assembly-like language of a call to set_logical_pc(logical_PC) followed by a call to logical_direct_jump() at native PC n is given in Figure 4-1.

```
fragment's body:
  push logical_pc          ; from C compiler
  call set_logical_pc      ;
  pop 1                    ; from C compiler
  push dcontext            ; from Dynamorio
  push n+1                 ; from Dynamorio
  push linkstub            ; from Dynamorio
  call logical_direct_jump ;
  pop  3                   ; from Dynamorio
  jmp  exit_stub           ; from Dynamorio
```

Figure 4-1: Translation of set_logical_pc(logical_pc) and logical_direct_jump()at instruction n.

Next subsections demonstrate the use of the above instrumentations in TinyVM (subsection 4.2.1) and in Kaffe VM (subsection 4.2.2).

4.2.1 Instrumenting TinyVM

We use the *CALLOP*, *RETOP*, and *BEQOP* bytecodes to demonstrate the instrumentation of TinyVM. In TinyVM calls to functions are statically dispatched and their targets (bytecode offset) are embedded in the bytecode stream. Thus, a call is a direct jump (i.e., from a given call site control will always transfer to the same target); and we instrument the handling of the *CALLOP* using logical_direct_jump. The *RETOP* bytecode in TinyVM behaves similarly to a native return; thus, we instrument it using logical_indirect_jump.

```
case CALLOP:
  ... setup new call frame ...
  pc = arg; /* go to start of function body */
  set_logical_pc(pc);
  logical_direct_jump();
  goto loop;

case RETOP:
  ... clean up stack ...
  pc = pop_raw();   /* pop the return PC */
  logical_indirect_jump(pc);
  *(++sp) = val;    /* put return value back on stack */
  goto loop;
```

The case of *BEQOP* is more interesting, because the true branch is a direct logical control transfer (remember that the *arg* value comes from the bytecode vector), and the false branch is a regular sequential promotion of the pc. The logical transfer is direct because the native conditional branch has already distinguished between the true and false cases. The instrumentation for *BEQOP* is:

```
case BEQOP:
  n = pop_int();    /* top must be an int */
  if (n) {          /* not equal 0 */
    pc++;           /* just continue */
  } else {          /* equal 0 */
    pc = arg;       /* do the branch */
    set_logical_pc(pc);
    logical_direct_jump();
  }
  goto loop;
```

Macro command	Bytecode	Logical CTI	Comments
call_indirect_Method	InvokeStatic InvokeSpecial	direct	
call	InvokeVirtual InvokeInterface	indirect	
reference_code_label	IF* IF_ICMP* IF_ACMP* JSR, JSR_W GOTO, GOTO_W	direct	cbranch_int_const_* cbranch_int_* cbranch_ref_*
stored_code_label	RET	indirect	
table_code_label	TableSwitch LookupSwitch	indirect	branch_indirect branch_indirect

Table 4.1: Bytecodes in Java and their Logical CTI.

4.2.2 Instrumenting Kaffe

The Kaffe VM imposes two difficulties in instrumenting it for DynamoRIO-with-LogPC: It has a recursive tree walking implementation and it uses macros. In order to instrument Kaffe we first check the Bytecodes that change the control flow and then instrument the macro commands that they use.

Table 4.1 presents the list of instrumented macro commands (column 1), the bytecode in which they are used (column 2) and the type of logical CTI they perform (column 3). The last column explains the use of macros in the bytecodes.

The bytecodes of the form *IF** (the * stands for any of the following extensions: eq, ne, lt, le, gt, ge), perform a comparison of a passed value against zero. If the comparison succeeds, the execution proceeds at a known offset. If it fails, execution proceeds at the address of the instruction following this bytecode instruction. In the same manner, the *IF_ICMP** (comparing two values) and *IF_ACMP** (comparing two references) are performed. All these bytecode instructions use a *cbranch_** macro that performs the relevant comparisons. The target offset is set using the *reference_code_label* macro. That is the reason for which we instrument this macro instruction as shown in Table 4.1.

The *GOTO* and *GOTO_W* functionality is very similar to that of a native *goto* instruction. The execution proceeds at a known offset from the address of that instruction. Thus, these are direct Logical CTIs. The same explanation holds for the *JSR* and *JSR_W* instructions, which are similar to the *GOTO** instructions (with the difference that the return address is stored on the stack). The *RET* instruction (return from subroutine) behaves similarly to the native *return* instruction. Thus, it is an indirect logical CTI. Because *RET* uses the *stored_code_label* macro, we instrument it. Other indirect logical CTI instructions are *LookupSwitch* and *TableSwitch*.

In addition to the above instructions, the control flow of the program is changed whenever a method is invoked. Java has several bytecodes that deal with different kinds of method invocation. Java distinguishes between instance methods and class (static) methods. Class methods are invoked based on the type of the object reference, which is always known at compile-time (using static binding). Thus, *InvokeStatic* is a direct logical CTI. In contrast, the invocation of instance method is based on the actual class of the object, which is only known at runtime. Thus, *InvokeVirtual* is an indirect logical CTI.

InvokeSpecial bytecode is used to invoke instance methods based on the type of the reference: superclass, private, and instance initialization methods (rather than the object's class). This type of invocation uses static binding; thus, it is a direct logical CTI.

The last type of method invocation in Java is the *InvokeInterface*. This invocation performs the same functionality as *InvokeVirtual*; however, it uses a reference to an interface for the invocation (rather than a reference to an object). Thus, it is also an indirect logical CTI.

To declare the logical CTI when invoking a method, we added the instrumentation just before the exact invocation. This type of the instrumentation depends on the type of invocation. During method invocation the method is resolved and the program's execution continues from the first instruction of the method. Thus, control is transfered from the caller method, A, to the invoked method, B (which may invoke other methods). When the execution of method B is completed the control returns to method A, using one of the return-from-method bytecodes. This return action is an indirect logical CTI in respect to the program's flow. Because this return is implemented in Kaffe with a native return (note that the functions *VirtualMachine* and *runVirtualMachine* are called recursively), we need to instrument it as an indirect logical CTI. This is done upon returning from method B. After the control returns to the caller method, A, execution continues from that point. Figure 4-2 gives an example of the way to instrument the *InvokeStatic* bytecode.

```
define_insn(INVOKESTATIC)
{
     ...
call_indirect_method(m):
  softcall_initialise_class(method_class());
  set_logical_pc(m);                        // Logical PC = first bytecode
                                            // in the invoked method

  logical_direct_jump();                    // This is a direct logical CTI

  virtualMachine(m, sp+1, retval, tid);     // the invocation

  set_logical_pc(code + npc);               // Logical PC = next bytecode
                                            // in the current method

  logical_indirect_jump();                  // Return from method is an
                                            // indirect logical CTI
  ...
}
```

Figure 4-2: Instrumenting *InvokeStatic* bytecode in Kaffe.

4.3 Collecting Logical Traces

The mechanism for dealing with traces is similar to that of the original DynamoRIO, but with respect to the abstract PC and the logical CTIs (instead of the native PC and the native CTIs). Like DynamoRIO we keep basic-blocks in a basic-block cache. But in DynamoRIO-with-LogPC, a basic-block ends with a native CTI (like in the original system) or a direct/indirect logical CTI.

DynamoRIO-with-LogPC uses abstract PCs, which are the target of a direct/indirect logical CTIs, as trace heads. All information about potential traces is stored and maintained in a separate table, called the "logical basic block table." Each entry in this table represents a "logical basic block." This structure includes a counter and a pointer to an optional existing trace that starts at the corresponding abstract PC location.

The first time a given abstract PC is encountered (targeted by a Logical CTI), it will not be found in the logical basic block table, and an entry will be added. After the same abstract PC is encountered `hot_threshold` times, from direct or indirect logical jumps, a flag is set to signal DynamoRIO's top level loop to start recording a trace, keyed by the abstract PC.

While trace building is active, DynamoRIO collects each basic-block into a contiguous list of instructions (this is regular DynamoRIO trace collection). It finishes recording the trace when a Logical CTI call is encountered. This is implemented by a flag that signals DynamoRIO to stop building the trace. When DynamoRIO emits the trace into the cache, it associates the trace with the corresponding abstract PC in the logical basic block table.

The next time a logical jump encounters an abstract PC for which a trace exists, it sets next_logical_trace_pc (a field in the thread-local dcontext structure) to this abstract PC. When control returns to the exit stub, the exit stub jumps directly to that target logical trace, without returning to DynamoRIO's dispatch loop. The following pseudo-code summarizes the functionality described in this section. In this example the we use the shorthand *lbb* for "logical basic block."

```
logical_jump(dcontext, native_pc, logical_pc) {
  if (currently building a trace) {
     if (this is trace-ending transfer) {
        set flag telling DynamoRIO to finish trace;
     }
  } else {  /* if not currently building a trace */
     lbb = lookup(abstract_PC),
          adding new entry if necessary;
     lbb->count++;     /* count hits */
     if (abstract_PC entry corresponds to a trace) {
        next_trace_global = lbb->trace;
     } else {  /* lbb not a trace */
        /* should we build a trace? */
        if (lbb->count > hot_threshold) {
           set flags telling DynamoRIO to start building a trace;
        }
     }
  }
}
```

```
logical_jump(dcontext, native_pc, logical_pc) {
  ...
     if (abstract_PC entry corresponds to a trace) {
        if (we are coming from a trace &&
            this is a direct logical jump) {
          link the two traces together;
        }
        next_trace_global = lbb->trace;
     }
  ...
}
```

Figure 4-3: Linking functionality in the Logical CTI function.

4.4 Linking

Similarly to DynamoRIO, DynamoRIO-with-LogPC also links basic-blocks, in order to improve performance. However, unlike DynamoRIO, it does not link basic-blocks that end with logical CTIs. Linking traces in DynamoRIO-with-LogPC is possible only between traces which are related by direct logical transfer control. In other words, assuming *trace A* (tagged by an abstract PC *A*) ends with direct logical CTI that points to an abstract PC *B*, then *B* is a potential trace head, which is represented by a logical basic block entry in the logical basic block table. If *B* has a trace associated with it, say *trace B*, then *trace A* can be linked to *trace B*.

The trace-linking functionality is added to the routine that handles calls to logical_*_jump. Figure 4-3 presents the pseudo-code for handling trace linking in this routine .

Linking logical traces introduces a new performance problem. The calls to logical_direct_jump from the fragment's body would be redundant after linking the traces. These calls would needlessly lookup the trace we already know we are going to. To overcome this difficulty, we changed the layout of instructions dealing with the calls to logical CTI API functions. These changes involve moving calls to the logical CTI functions from the fragment body to the exit stub and adding code (in the exit stub) that supports linking. This means that we check if the target abstract PC has an existing trace associated with it, and if so we directly jump to it. Figure 4-4 is an updated version of the code presented in Figure 4-1; this version allows linking of traces. This is the current implementation in DynamoRIO-with-LogPC. Note that the code in the exit stub showed in Figure 4-4 is executed only before the linking, because linking replaces the "jump to exit stub" instruction with a direct jump to the trace's code in the cache.

```
fragment's body:
  push x                     ; logical pc
  call set_logical_pc        ; saves logpc (x) in a global
  pop 1
  jmp exit_stub
...
exit_stub:
  push dcontext              ; add arg. for logical_jump call
  push n+1                   ; add native PC arg.
  push from_data             ; for linking traces
  call logical_direct_jump
  pop 3
  test next_logical_trace    ; if logical jmp to trace
  jne *next_logical_trace    ; go there
  next_addr = n+1            ; otherwise
  jmp dynamo_dispatch        ; go to dynamorio with native PC = n+1
```

Figure 4-4: Translation of logical_direct_jump(x) at instruction n.

4.5 Discussion

In this chapter we presented the basic design of DynamoRIO-with-LogPC and introduced the annotations that are required for running the system. The correctness of the program relies on the correctness of these annotations. Wrong instrumentation would yield run-time errors and wrong results. An example of a wrong instrumentation is when an indirect logical CTI is annotated as a direct one; this causes the traces to be linked unnecessarily, leading to a malfunctioning program.

Although DynamoRIO-with-LogPC creates logical traces, it is still far from optimal (results can be seen in chapter 6). There are two reasons for this. The first one is that the modifications to DynamoRIO introduce more overhead into the original system, especially in maintaining the additional data structures, such as the logical basic block table. The second reason is that the interpretive indirect overhead is not yet eliminated; it was just inlined into traces. Moreover, DynamoRIO-with-LogPC cannot remove the overhead caused by dynamic features, such as method dispatch. While JITs provide many tailored optimization techniques for these situations, DynamoRIO-with-LogPC still does not have enough information to perform such optimizations.

The use of the logical PC creates, as expected, traces that correspond to frequent sequences of bytecodes. Because each bytecode is implemented using many native instructions, these traces may be very long. While it is generally advantageous to have long traces cached, very long traces may degrade performance. This might happen when traces occupy more space than is available in the cache, resulting in many cache misses (as we demonstrate in chapter 6.

To further improve performance, we apply optimization techniques to the traces generated. Although DynamoRIO supports some optimization techniques, such as constant propagation and dead-code elimination, we extend their functionality and add more optimization techniques, such as partial-evaluation. In the next chapter we describe these optimization techniques.

Chapter 5

Trace Optimization

Up to this point we explained the limitations of DynamoRIO and outlined the DynamoRIO-with-LogPC system - a system that is tailored for interpreters and VMs. In the last chapter we described the methodology used by the DynamoRIO-with-LogPC to collect and record traces. Although creating long traces does not remove all of the overhead introduced by the interpreters, it provides the background for additional optimizations.

After collecting traces, we optimize them and emit their optimized versions. Some optimizations techniques, such as constant propagation and dead code elimination, existed in DynamoRIO. However, in the new system, we can use hints from the interpreter writer regarding the structure of the interpreted program to better optimize traces. Thus, we extend the functionality of some of the optimization techniques that existed in DynamoRIO, and we introduce new ones.

Because most optimizations can be performed on every program, we refer to the general system as DynamoRIO. We emphasize cases in which an optimization is only tailored to DynamoRIO-with-LogPC.

This chapter's outline is as follows: In the first four sections we discuss "traditional" optimization techniques. Section 5.1 introduces the notion of immutable addresses. These addresses are used in constant-propagation, discussed in section 5.2, and in dead-code-elimination, as explained in section 5.3. Section 5.4 gives an overview of the call-return optimization technique. Then, section 5.5 describes optimization techniques to avoid too long traces, such as inlining of unrolled native loops. Section 5.6 presents the Functional-Memorization optimization technique. Lastly, section 5.7 summarizes this chapter.

5.1 Immutable Regions and Trace-head Constants

In this section we present the use of immutable regions in DynamoRIO-with-LogPC and explain their potential in optimizing traces. Then, we introduce the concept of "trace-head constants," which allows for additional optimizations.

5.1.1 Immutable Addresses

The classic definition of "immutable addresses" is: addresses whose content does not change during the execution of the program. DynamoRIO-with-LogPC can benefit from knowing immutable addresses; it can use this information during constant propagation to fold dereferences of the immutable program representation to constant values.

The need for adding information regarding immutable addresses is derived from cases, such as a *switch* instruction. A switch command is an indirect control transfer. Typically, large switch tables reside in the ".rodata" section, and their location is a known constant address. They include an array of addresses that correspond to the cases' handling routines. Dispatching a specific case involves dereferencing the address in the index's position in the switch-table (to obtain the target address). The problem we encountered is that even when the index is known (a constant), the instruction cannot be simplified during constant-propagation, without any knowledge about the addresses in the switch-table. However, if we know that the switch-table is an immutable region, the corresponding address in the switch table can also be treated as constant.

Thus, treating the read-only sections in the programs as immutable regions can improve constant propagation performance. To extend this idea even further, we use all sections that are not writable (i.e. READONLY, EXECUTABLE), as immutable.

However, more optimization could be performed with additional knowledge about other immutable regions of the program. Thus, DynamoRIO-with-LogPC provides the interpreter

```
In addMethodCode(Method* m,Code* c) in kaffe\kaffevm\classMethod.c:
        set_region_immutable(m->c.bcode.code,
            m->c.bcode.code + m->c.bcode.codelen - 1);
        set_region_immutable(&m->c.bcode.code,
            &m->c.bcode.code + sizeof(unsigned char*) - 1);
        set_region_immutable(&m->c.bcode.codelen,
            &m->c.bcode.codelen + sizeof(int) - 1);

In main(int argc, char* argv[]) in kaffe\kaffevm\main.c:
        set_region_immutable(insnLen,
            insnLen + 256*sizeof(uint8) - 1);
```

Figure 5-1: Example to instrumenting Kaffe VM for immutable data.

writer with a simple API function to declare a region of memory as immutable: **set_region_immutable(start, end)**. This instrumentation identifies the region of memory delimited by *start* and *end* (inclusive) as immutable. It may be called for any number of regions. Once it is called for a given region, that region of memory must not be written to for the duration of the program. The set of immutable regions is stored in a sorted linked list.

Here is an example to this instrumentation for the TinyVM interpreter. In TinyVM the array of bytecodes of the program is immutable. Thus, we pass this information to DynamoRIO via the provided API function, as follows:

```
set_region_immutable(instrs,(instrs + num_instrs*sizeof(ByteCode)-1));
```

In Kaffe VM the bytecode array for each method is immutable. Thus, whenever the method was created, while loading the class, set_region_immutable is called. In addition, the dispatch table of the length of the bytecode, InstrLen, is also immutable. We add this instrumentation to the main function of Kaffe. Figure 5-1 presents the instrumentations used in Kaffe for declaring immutable regions.

5.1.2 Trace-Head Constant Addresses

In DynamoRIO-with-LogPC, we allow a more aggressive use of immutable addresses. We define Trace-head constant addresses: addresses which their contents are constant for a given *Abstract PC*.

Note that these addresses are usually a function of the *Logical PC*. Moreover, the value of trace-head constant addresses can be changed within a trace. All that we require is that at the beginning (head) of the trace, these addresses will have a fixed known value. Trace-head constant addresses must not be aliased. That is, if *addr* is identified as a trace-head constant, changes to the value stored in *addr* must be apparent in the instruction sequence, rather than changed indirectly through another memory location. The following example violates the anti-aliasing requirement:

```
ByteCode* pc = ...;
ByteCode* alias = pc;
add_trace_constant_address(&pc);
*alias = some_bytecode;
```

DynamoRIO provides two API functions for declaring trace-head constants. The first one deals with global memory addresses and the second deals with stack-allocated addresses. The instrumentation for declaring addresses as trace-head constants is:

add_trace_constant_address(addr). It identifies an address whose value (i.e. when dereferenced) is guaranteed to be the same whenever control reaches the *Abstract PC* of the call. For example, suppose a call to add_trace_constant_address(&addr) is made when the *Logical PC* is *lpc*, the native PC of the call site is *npc*, and the value of *addr* is 42. If DynamoRIO-with-LogPC makes a trace headed by the *Abstract PC (lpc, npc)*, then any dereferences of *addr* can be folded to the constant 42. Moreover, changes to these addresses can be tracked during the trace. DynamoRIO obtains the values of these addresses by taking a snapshot of their contents immediately when it starts building the traces.

For immutable addresses that are stack-allocated, DynamoRIO provides another API function. The reason for the separation is that for stack-allocated addresses, DynamoRIO needs to calculate the stack offset for the address. The interface is:

```
value eval () {
     int pc = 0;
     ...
loop:
     ...
     switch (op) {
     case CALLOP:
     ...
     set_logical_pc(pc);
     logical_direct_jump();
     set_trace_constant_stack_address(&pc, pc);
     ...
     }
}
```

Figure 5-2: Example to instrumenting TinyVM for trace-head constant addresses.

set_trace_constant_stack_address(addr, val). This interface identifies a stack-allocated variable (at address *addr* on the stack) that currently has the value *val*. In this case the values of the addresses must be passed explicitly.

We demonstrate the use of this instrumentation in the context of TinyVM and Kaffe. In TinyVM the *pc*, which is the pointer to the bytecode's array, is a local variable in the main function. As described in section 1.3, it is used as the *Logical PC* of the interpreter. This address is a trace-head constant address because it is a part of the trace tag and it is unaliased. Because the *pc* is a local variable, we use the set_trace_constant_stack_address for its annotation. Figure 5-2 gives the trace-head constant addresses instrumentation of TinyVM.

In Kaffe, both the method pointer (*meth*) and the offset to its bytecode array (*npc*) are passed as arguments to the function that implements the main interpreter loop. Since both are used in the calculation of the *Logical PC*, like in TinyVM, they are trace-head constant addresses. Moreover, other local variables, such as the address of the bytecode array and the index pointer into it, are also a function of the *Logical PC*; and thus, they are trace-head constant addresses. Figure 5-3 gives the trace-head constant addresses instrumentation of Kaffe.

```
void runVirtualMachine(methods *meth, ..., uintp npc, ...) {
        bytecode *code = (bytecode*)meth->c.bcode.code;
        uintp pc;
        pc = npc;
        ...

        set_logical_pc(code+npc);
        logical_*_jump();
        set_trace_constant_stack_address(&code, code);
        set_trace_constant_stack_address(&npc, npc);
        set_trace_constant_stack_address(&meth, meth);
        set_trace_constant_stack_address(&pc, pc);
        ...
}
```

Figure 5-3: Example to instrumenting Kaffe for trace-head constant addresses.

In the next section we elaborate on the use of immutable addresses and trace-head constants when performing constant-propagation.

5.2 Constant Propagation

The ability to fold references to immutable memory into constants is crucial to the success of constant propagation. As mentioned earlier, we use *set_region_immutable* calls from the interpreter to establish constant memory regions.

DynamoRIO performs classical constant propagation, which is made more difficult by the fact that it targets X86 instruction set. It sequentially scans the trace's instructions, and simplifies the instructions if possible. Whenever it encounters a constant address it checks whether this address is contained in a known immutable region, or it is one of the trace's constant addresses. If it is an immutable address, DynamoRIO simplifies the original instruction by replacing the dereference to the address with a constant value, which is obtained by reading the content of the address. If the address is a trace constant address, then its content is known, and DynamoRIO simply replaces the dereference of the address with its value.

A similar approach is taken when dealing with stack-allocated addresses that are trace-head constants. The difficulty in this case is to compute the offset of the address in the stack frame. DynamoRIO uses the layout of the instructions that correspond to the call in order to obtain this information. During the building of traces, when the call is executed, the constant values are stored. The offset is retrieved by inspecting the instructions during the optimization stage. Here is the expected layout of the instructions:

```
mov    offset(ebp) -> reg
push   reg
lea    offset(ebp) -> reg
push   reg
call   set_trace_cnstant_stack_address
```

DynamoRIO detects calls to set_trace_constant_stack_address while inspecting the code during constant propagation. Then, DynamoRIO looks on the previous instructions for the expected pattern presented above. If the pattern is found, the offset is obtained, and the instructions can be simplified or eliminated. The compiler might reschedule other instructions in this sequence of instructions, but using a more aggressive search that skips instructions until it finds the pattern, will eventually obtain the right offset. After obtaining the offset and eliminating the call, the sequence of instructions includes only the stack manipulation (instead of the *push* instructions).

Stack-allocated addresses can be used only in the scope in which the declarative call was made. The way DynamoRIO keeps track of scopes is as follows: At the beginning of a trace the scope is set to zero. Each time an *enter* instruction or a "push ebp" (*mov esp* $\rightarrow$ *ebp*) instruction is encountered the scope is increased by one. Whenever a *leave* instruction or a *pop ebp* instruction is encountered the scope is decreases by one. When DynamoRIO detects a call to set_trace_constant_stack, it fills the information regarding the addresses in a structure used for constant propagation. The information in this structure includes the value of address (as was stored while building the trace), the offset from the stack frame (as was computed), and the current scope. DynamoRIO only folds dereferences of stack-allocated addresses to constants when the control is within the stack frame of their corresponding call to *set_trace_constant_stack*. Whenever an instruction that writes to the *ebp* register is detected all stack-allocated addresses are invalidated (i.e. cannot be used for constant propagation), because the scope tracking is violated.

In addition to the trace-head constant addresses, DynamoRIO provides interfaces to declare addresses that are unaliased, but their value is not constant in respect to the trace. DynamoRIO provides the **unaliased_stack_address(addr)** API function for global memory addresses and **unaliased_address(addr)** API function for stack-allocated addresses. These interfaces indicate to DynamoRIO that the addresses are unaliased, and thus some constant propagation operations can be performed. DynamoRIO treats these addresses in the same manner it treats their corresponding interfaces, in which the value is provided. An example to the need for such instrumentation is taken from Kaffe, where the stack pointer may change each time the trace starts (thus, it is not a trace-head constant). However, since it is unaliased, it can be used during constant propagation to remove redundant loads.

The information regarding trace-head constant addresses, with known or unknown values, can be further used in the dead code elimination as we describe in the next section.

5.3 Dead Code Elimination

Constant propagation produces dead code, such as from storing temporary values that are no longer needed, which is removed using dead code elimination. However, if addresses are known to be unaliased (global memory addresses and stack-allocated addresses), more aggressive dead code elimination can be performed. Recall that unaliased addresses were used in constant propagation, with or without known values. However, in dead code elimination we are concerned only with the addresses.

While the information about constant global addresses is valid throughout the trace, the information regarding stack-allocated addresses is valid only in the scopes they were declared at. This fact imposes two difficulties on the dead code elimination process. First, since the trace's instructions are scanned backward, there is no notion of the scope with respect to the beginning of the trace. Second, the information regarding the stack-allocation addresses is local to the constant propagation and needs to be passed to the dead code elimination. The way constant propagation passes this information is by annotating some instructions in

instructions	Scope	CP-List	action
call stack_addr(x)	0		add (x,0) to list
... ...	0	(x,0)	
push			advance current scope
call stack_addr(y)	1		add (x,0) to list
... ...	1	(y,1),(x,0)	
pop			decrease current scope; annotate instruction (y,1)
... ...	0	(x,0)	
last_instr	0		annotate instruction (x,0)

instructions	Scope	DC-List	action
last_instr			add (x,0) to list
... ...	0	(x,0)	
pop			add (y,1) to list increases current scope
... ...	1	(y,1),(x,0)	
push			remove (y,1) from list decrease current scope
... ...	0	(x,0)	

Figure 5-4: The left table contains constant propagating data. The right table is dead code elimination data.

the trace. We use the example in Figure 5-4 to explain this annotation process.

In the left table of Figure 5-4 we show the sequence of events in the constant propagation process. The first column gives the sequence of instructions from the beginning of the trace. The second one is the relative scope. The third column presents the list maintained by constant propagation to hold the stack-allocated addresses (note that we eliminate the actual values from this presentation). The last column shows special actions taken by the constant propagation. The right table presents similar data, in respect to the dead code elimination. Note however that the sequence of instructions is backward, i.e., it starts at the last instruction of the trace.

In order to pass the scope at the end of the trace to the dead code elimination, constant propagation annotates the last instruction in the trace with its scope. Thus, when dead code elimination starts, it first set the current scope. In addition, it receives the set of addresses and their associated scopes that are valid at this point.

Whenever a *leave* or a *pop* instruction is encountered during the constant propagation, the instruction is marked to pass the addresses that are constant at the previous scope. (In our example this would add offset y and scope 1.) When dead code elimination encounters this instruction it updates its list. An *enter* instruction or a “push” (*mov esp* → *ebp*) instruction indicates to the dead code elimination to remove the addresses of the previous scope from the list. (In our example this would remove offset y and scope 1.)

5.4 Call-Return Matching

DynamoRIO performs expensive checks when a return is encountered, because returns are indirect branches. If the static analysis of the trace reveals that the return matches a call site in the trace, the return overhead can be eliminated. This is a very powerful optimization, especially in DynamoRIO, as we show in the Results chapter. Though, we did not modify its implementation from the basic DynamoRIO infrastructure, it is very beneficial for interpreters, because the traces are long and contains many nested function calls. Thus, there are many return instructions that can be eliminated.

5.5 Avoiding Too Long Traces

On one hand, it is beneficial to have long traces, because we can perform more traditional optimizations (across basic-blocks). However, when the total traces' size is too big to fit in the cache, we encounter a degradation in performance, as we will see in the Results chapter.

There are a number of reasons for having longer traces in DynamoRIO-with-LogPC than in DynamoRIO. First, the traces in the DynamoRIO-with-LogPC represent a sequence of bytecodes. Each bytecode corresponds to a long sequence of native instructions. For example, if function calls are included in the bytecode implementation, they are inlined in the trace, expanding its length. In addition, traces include the inlined interpretive overhead (which is eliminated by the traditional optimization techniques).

The second main reason is that native loops are also inlined in the trace. This happens if one bytecode in the trace is implemented with a *for* or a *while* instruction. Then the trace's length might grow substantially. The loop is unrolled in the trace, and for each optional exit, DynamoRIO creates an exit stub. These exit stubs have low potential to be taken, however they occupy large space in the trace. An example of an unrolled loop is the lookup for a field in a table. If the lookup is implemented by scanning the elements of the table until reaching the required field, the whole loop would be inlined.

There are two ways to avoid the problem of too long traces. The first, is to restrict the length of the traces to a maximum number of basic-blocks, using a runtime option. This solution is easy to implement and use. However, choosing the best maximum number of basic-blocks is not trivial, especially because basic-blocks might have different sizes. We experimented with the choice of maximum basic-blocks in a trace. When the maximum of basic-blocks was too small, there was no profit from building, optimizing, and caching traces. At some number, we got the best results. Increasing the number of basic-blocks further degrades the performance.

The second way to avoid unrolled native loops is to instrument the interpreter, such that it would not inline functions with loops. This information can be provided to DynamoRIO using the instrumentation: **set_call_no_inline(func_addr)**. This indicates DynamoRIO not to inline the function at address *func_addr*. In TinyVM we identified *pop, make_array, dump_val, popargs, decr_refs* as functions that contain loops, and instructed DynamoRIO-with-LogPC not to inline them. By doing so, we substantially improved performance. In the Results chapter we see that by not-inlining only two functions in Kaffe, we could speedup performance by more than 30% compared to the run without it (see Table 6.4).

The drawback of this method, is the difficulty of identifying these functions, as there may be many functions with loops (especially in large interpreters such as Kaffe). Also, by not-inlining many of them, we do not take advantage of caching and optimizing this code. This problem is one of the major open problems in DynamoRIO-with-LogPC. We further discuss this problem and present possible fixes to it in the Summary chapter.

5.6 Functional Memorization

The functional memorization optimization technique is based on the classical concept of partial evaluation, as described in section 2.2. DynamoRIO-with-LogPC inlines calls to functions, and these inlined functions may later be a part of traces. Some of the inlined functions are called with the same parameters each time the trace is executed and result in the same returned value. We can take advantage of this behavior by storing the returned values and carefully replacing function calls with these values. This saves both time and space. Time is saved because we do not perform the function call; and space is saved because we reduce the size of traces (although this could also be done by not inlining the function call). Here are the conditions that must be satisfied so that we can perform functional memorization on a function:

1. The function's parameters should be constants with respect to the trace.
2. The returned value should be the same for each execution of the function with the same parameters.
3. The function should not have side effects, such as memory changes or system calls. This is important because the function call will be removed; thus, this changes would not be able to be performed.

DynamoRIO provides the interpreter writer with a **part_eval_func(return_value)** interface to identify functions for functional memorization. The *return_value* is the value returned from the function. This instrumentation should immediately follow the function call.

We demonstrate this instrumentation based on Kaffe: We identified the function *Field* lookupClassField(class, poolInfo, isStatic, errorInfo)* as a good candidate for functional memorization. This function is called from each of the bytecodes that deal with static and non-static fields (i.e., *getfield, getstatic, putfield, putstatic*). The function's parameters depend on the class (class and constant pool information), which in turn depend on the method, and on the bytecode that is executed (isStatic). Since the bytecode is inlined, and the method is part of the trace's tag, this information is trace-head constant address. Because, the method is the same each time the trace is run, this function results in the same field pointer each time. Moreover, this function performs a search without changing any of the program's structures or addresses. Thus, it satisfies all of the conditions required for it to be functionally memorized. The part of the code that includes the instrumentation of this function is:

```
field = lookupClassField(class,WORD2UTF(pool->data[NAMEANDTYPE_NAME(ni, pool)]),
                            isStatic, einfo);
part_eval_func((int)field);
```

In order to perform the replacement correctly, DynamoRIO requires that the function would be defined as "not to be inlined," as described in section 5.5. For the example above we instrumented Kaffe by adding the following annotation into its main function:

```
set_call_no_inline(&lookupClassField);
```

The not-inlining instrumentation allows us to identify calls during functional memorization and to replace them with an equivalent constant. For performing functional memorization it is essential that the function would be declared as "not to be inlined." Otherwise, the boundary of the function's body could not be distinguished, and thus, the function could not be removed.

The way functional memorization is implemented in DynamoRIO is as follows: While building traces, DynamoRIO stores the returned values (when processing the part_eval_func calls). After recording traces (during the optimization stage) these traces are inspected. When a part_eval_func call is encountered, DynamoRIO looks for its preceding call instruction. This call is assumed to be the function's call. Based on x86 calling convention the returned value resides in the *eax* register. Thus, we replace the call to the function with an instruction that moves the constant value to the *eax* register. DynamoRIO is unable to remove the push and pop instructions that deal with the functions' parameters, because there may be different number of parameters for these functions. Yet, it cleans the call to part_eval_func, so it would not be executed again. Figure 5-5 gives the sequence of instructions that includes the instrumentation before and after the functional memorization.

Note, that our optimizations are not capable to track trace-head constants through function calls (because the use of registers for the calling convention). Thus, even with enough annotations of the immutable regions and trace-head constants, the current techniques would not succeed removing the inlined loop.

5.7 Discussion

In this chapter we presented various optimization techniques that can be applied to generated traces. We described the annotations required and demonstrated their use in the context of TinyVM and Kaffe.

```
before functional memorization:
        push args
        call lookupClassField
        pop args
        ...
        push return_value
        call part_eval_func
        pop return_value

after functional memorization:
        push args
        mov return_value->eax
        pop args
       ...
```

Figure 5-5: The sequence of instructions, including the functional memorization instrumentation, before and after functional memorization.

In the first four sections we described "traditional" techniques, with the appropriate modifications and extensions so they better suit interpreted programs. In the first section we described the notion of "immutable regions." For our purpose, we presented the concept of trace-head constant addresses. These addresses can be folded to constants during constant-propagation. We also explained the way the values are retrieved for both global addresses and stack-allocated addresses. Section 5.2 described constant-propagation. It provided a detailed explanation of handling immutable addresses, and in particular stack-allocated addresses. That section also introduced the instrumentations of unaliased addresses (both global and stack-allocated), and explained their use in constant-propagation.

Both the immutable addresses and the unaliased addresses can also be used in dead-code-elimination, as described in section 5.3. We also described the design of the interface between constant-propagation and dead-code-elimination to pass this information. Section 5.4 briefly described call-return-matching technique, as it was previously used in DynamoRIO and did not need any modifications to better fit it to interpreters.

While the traditional optimizations successfully remove the interpreter dispatch loop, the resulting traces may still be too long, as demonstrated in the Results chapter. Thus, in section 5.5 we showed techniques that avoid too long traces. We gave one solution that truncates the long trace in a “brute-force” way, based on the number of basic-blocks in the trace. Another solution that we presented eliminates inlining of unrolled native loops by not inlining the function that contains the loop. We discussed the pros and cons of each of these solutions and conclude that neither is perfect.

In section 5.6 we provided the functional memorization technique. It was motivated by the not-inline solution used to shorten traces. We noticed that many functions, especially for search and lookup, would be called from the trace with the same parameters (which are a function of the trace tag), and would result in the same value, each time it is called, without changing any other memory address. These functions usually contain loops, and thus, by performing functional memorization on them, we reduce the trace length, and saves execution time.

Next chapter gives results of using DynamoRIO-with-LogPC, and shows the efficiency of the optimization techniques discussed here.

Chapter 6

Experimental Results

In this chapter we present results of different tests that measure the efficiency of DynamoRIO-with-LogPC. We experiment with both TinyVM and Kaffe VMs. Our results show that the DynamoRIO-with-LogPC and optimizations can improve performance of interpreters and VMs.

The following is a list of the benchmarks used for TinyVM and their experimental parameters:

fib2_ref This program finds all the Fibonacci numbers up to a given number. In the experiments we used 36 as this number.

fib2 This program generates the Fibonacci numbers up to a given number repeatedly (in a loop). In our experiments we used 40 as the upper number for the Fibonacci and 600,000 as the number of loops.

bubble This program performs a bubble search in an array of a given size, set to the worse case configuration. In our experiments we used an array size of 4000.

sieve2 This program calculates the sieve of Eratosthenes, i.e. finding all the prime numbers smaller than a given number. In our tests we used 5,000,000 as this number.

sieve_loop This program executes the Sieve of Eratosthenes, a given number of times (in a loop). In our tests we used 1000 for the number of executions of the sieve, and 1000 for the sieve upper limit number.

matrix, matrix2 - These programs execute matrix multiplication of two square matrices. The multiplication is performed a number of times. In **matrix** the size of each matrix is 100x100 and the number of executions is 2. In **matrix2** the size of each matrix is 24x24 and the number of executions is 1000.

For Kaffe we used three of the SpecJVM98 benchmarks [46]: compress, db and mpeg, with a problem size of 100. We emphasize that the results reported here are not comparable with a SpecJVM98 metric.

We ran our tests on a Pentium-3 i868 machine, using Linux kernel, version #1 release 2.4.7-10, with no other load. In all tests with DynamoRIO and DynamoRIO-with-LogPC we used a hot_threshold of 50.

This chapter begins with a summary of the performance results for TinyVM and Kaffe, for DynamoRIO, DynamoRIO-with-LogPC, and DynamoRIO-with-LogPC and optimizations in section 6.1. We present the overall performance in comparison to the native execution of the program. Then, we investigate these results in different aspects. Section 6.2 focuses on the differences between DynamoRIO and DynamoRIO-with-LogPC. We compare them in two aspects: trace utilization, and trace length; and we explain why DynamoRIO performs so poorly on interpreters.

In section 6.3 we examine the optimizations used, and investigate each one of them. We first look at the traditional optimization techniques (i.e. constant-propagation, dead-code-elimination and call-return-matching). We investigate the behavior of the TinyVM benchmarks in terms of the performance and the trace length. We show that the best performances (improvement of 38% on average) are due to the use of all of the traditional optimizations together, and that the total traces' size is shortened by an average of 66% using these optimizations. While these techniques improve performance, they do not sufficiently reduce trace size. Thus, we further investigate the long traces generated by DynamoRIO-with-LogPC. We analyze the effects of using trace reduction methods by comparing the different methods in traces length, cache misses and overall performance. These tests confirm our assumption that too long traces harm performance, and that by reducing traces' size we can substantially improve performance. The last part of section 6.3, presents the results

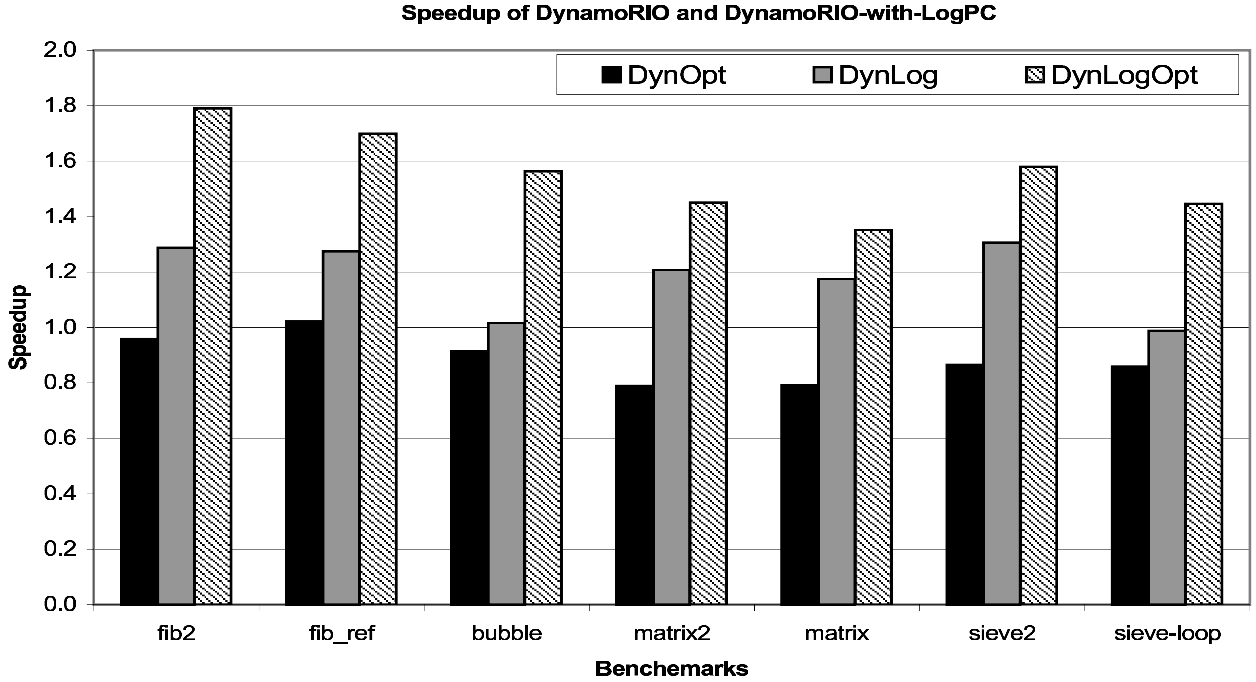

Figure 6-1: Performance in three configurations, normalized against native execution, for TinyVM benchmarks. The configurations are: DynamoRIO with optimization (DynOpt), DynamoRIO-with-LogPC (DynLog), and DynamoRIO-with-LogPC and optimizations (DynLogOpt).

for the functional memorization optimization technique in Kaffe, and shows that it further improves performances by 30-50%. Finally, section 6.4 summarizes the experimental results of this thesis.

6.1 Performance Summary

Figure 6-1 displays the performance (native run time divided by DynamoRIO's run time) on TinyVM benchmarks (described above), of the DynamoRIO with optimizations, DynamoRIO-with-LogPC, and DynamoRIO-with-LogPC with optimizations. These results are normalized against the execution time of the native programs. The optimization techniques used with both DynamoRIO and DynamoRIO-with-LogPC are the traditional ones (constant-propagation, dead-code-elimination and call-return-matching). In addition, DynamoRIO-with-LogPC also does not inline the *pop* function, as an unrolled loop avoidance method. We note that for TinyVM, functional memorization was not used, since neither of its functions satisfies the requirements for applying functional memorization.

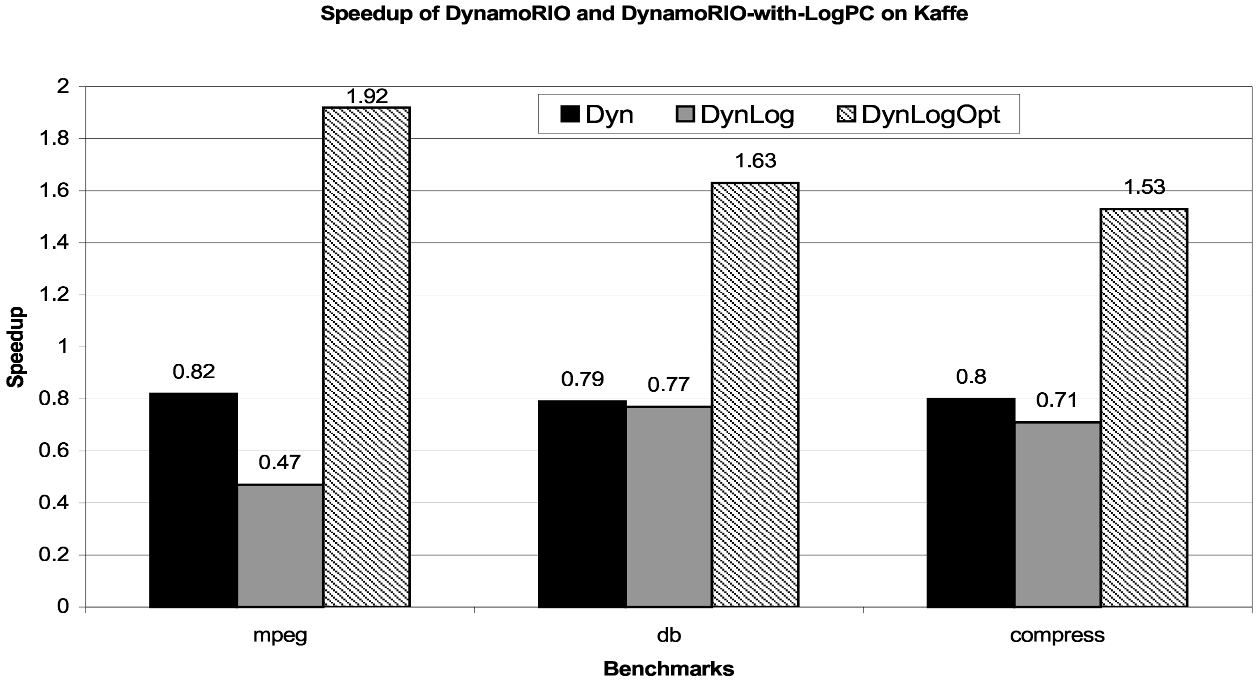

Figure 6-2: Running times in three configurations, normalized against native execution, for Kaffe. The configurations are: DynamoRIO (Dyn), DynamoRIO-with-LogPC (DynLog), and DynamoRIO-with-LogPC and optimizations (DynLogOpt).

From the graph we see that DynamoRIO is slower by up to 20% than the native run. Using DynamoRIO-with-LogPC improves the performance by creating long traces that correspond to frequent sequences of code in the high-level program (the interpreted program). We can see that DynamoRIO-with-LogPC achieves 20-30% speedup in most cases. However, in two of the benchmarks (bubble and sieve-loop) the results are only close to the native ones. One explanation for this is that DynamoRIO-with-LogPC creates too long traces that harm its performance, as we explained in section 5.5.

The results for DynamoRIO-with-LogPC and optimizations substantially improve the performance of all the benchmarks. It outperforms the native execution with a speedup of 30-90% (where a pseedup of 100% would be half the executin time). The performance of DynamoRIO-with-LogPC can be further improved, by not-inlining additional functions, as demonstrated on sieve-loop in section 6.3.2.

A similar behavior can be seen in Figure 6-2 for SpecJVM98 benchmarks, running on Kaffe. This Figure shows that DynamoRIO-with-LogPC (without optimizations) is slower by

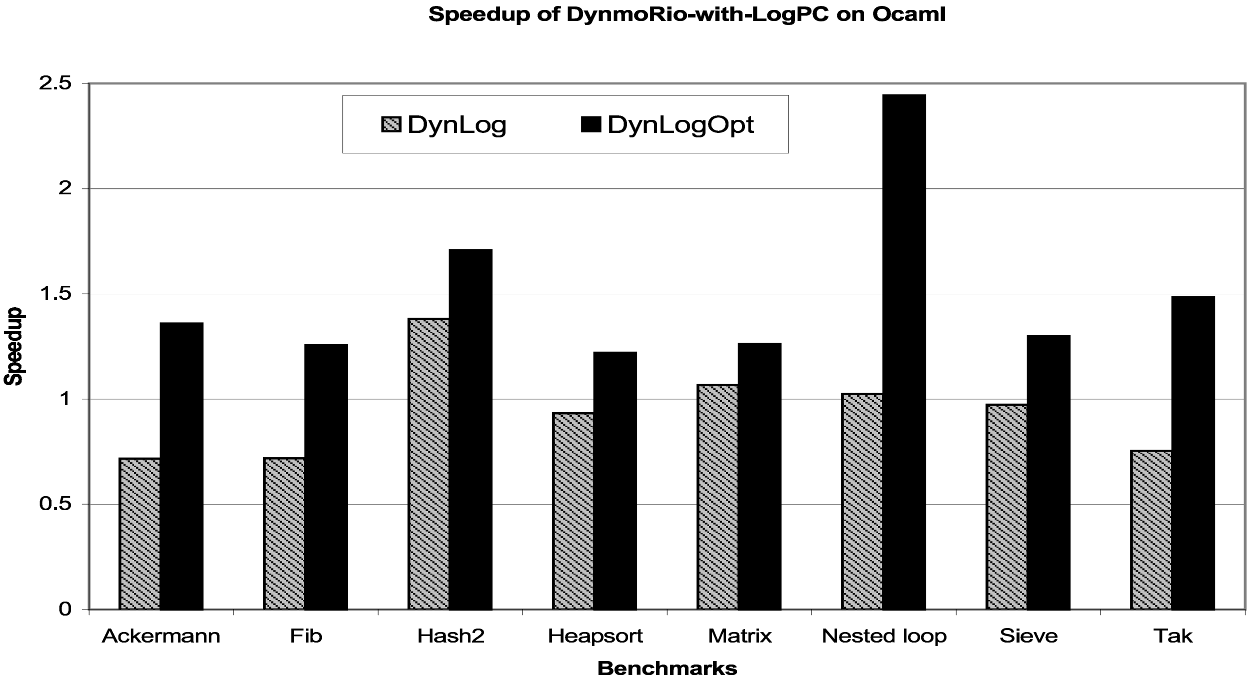

Figure 6-3: Running times in two configurations, normalized against native execution, for Ocaml. The configurations are:DynamoRIO-with-LogPC without optimization (DynLog), and DynamoRIO-with-LogPC and optimizations (DynLogOpt).

a factor of 1.4-2.5 than the native run-time. This happens due to the extremely long traces that are generated in Kaffe. DynamoRIO-with-LogPC and optimizations uses the following optimizations: the traditional optimization techniques, and 'not-inline' and functional memorization on the *lookupClassField* and *findMethodLocal* functions. Applying optimizations improves the performance of DynamoRIO-with-LogPC and presents a speedup of 1.5-2 in comparison with the native run times. In fact, optimizations reduce DynamoRIO-with-LogPC run times to less than half, and in the *mpeg* case even to less than a quarter. These improvements show that even when the performances of DynamoRIO-with-LogPC are poor, applying the right optimization techniques can still substantially shorten the run time of programs.

In addition to Kaffe and TinyVM interpreters, we also experimented with Ocaml interpreter. Figure 6-3 shows the performance of these tests. The results are consistent with the ones obtain from TinyVM and Kaffe. While DynamoRIO-with-LogPC and no optimizations is slower than the native in some cases, the performance of DynamoRIO-with-LogPC and

trace	DynamoRIO					DynamoRIO-with-LogPC				
	size	% in trace	Exit			size	% in trace	Exit		
			stub#	%of trace	%time			stub#	%of trace	%time
1	956	17.5	1	21.8	94.3	13884	48.6	34	14.6	1.3
			10	98.5	5.7			242	100	98.7
2	745	16.5	5	71.3	19.9	26163	24.4	347	100	100
			8	98	80.1					
3	1279	6.1	7	47.6	1.4	52582	13.6	161	23.3	93
			16	99.1	98.6			492	70.5	3.3
								700	100	3.7
4	1219	6.1	15	99	100	7479	6.2	97	100	100

Table 6.1: Hottest traces' exit statistics and sizes, for Sieve2 benchmark on TinyVM using DynamoRIO and DynamoRIO-with-LogPC.

optimizations is by far better than the native run time.

6.2 Comparison between DynamoRIO-with-LogPC and DynamoRIO

In this section we discuss the results of DynamoRIO-with-LogPC without optimization, in comparison with the ones of the DynamoRIO. The two main problems that cause DynamoRIO to be slow with interpreters are that its traces are short and that the traces are poorly utilized. In this section we demonstrate this behavior using the sieve2 benchmark.

6.2.1 Trace Utilization

In Table 6.1 we present the four hottest traces and their sizes and exit statistics, when executing the sieve2 benchmark on TinyVM using DynamoRIO and DynamoRIO-with-LogPC. We see that for DynamoRIO the hottest trace, which corresponds to the interpreter's dispatch loop, is executed 17.5% of the program run time. Moreover, 94.3% of the times, the program exits the trace on its first exit (executing only 21.8% of the trace). This exit stub corresponds to the *switch* instruction in the interpreter program, which is an indirect jump. As we explained in chapter 3 an indirect jump is a time consuming operation in DynamoRIO. This indirect jump is required since the program takes different paths when it exits the trace

	DynamoRIO					DynamoRIO-with-LogPC				
benchmark	number	mean	max	min	total	number	mean	max	min	total
fib2	38	330	1551	57	12564	3	22153	43859	1739	66458
fib_ref	22	514	1386	49	11321	6	3564	4705	1050	21383
bubble	68	302	1270	49	20600	5	27727	51905	5784	138634
matrix2	112	233	1020	52	26141	17	25144	52742	5066	427441
matrix	113	233	1020	52	26424	17	25158	44987	8950	427694
sieve2	70	316	1270	55	22117	6	16717	47452	372	100302
sieve-loop	114	259	1136	49	29488	11	133490	902130	372	1468393

Table 6.2: Trace characteristics for DynamoRIO vs. DynamoRIO-with-LogPC (no optimizations) on TinyVM's benchmarks. The first column gives the number of traces; columns 2,3,4 show the mean, max and min length of traces, respectively (in bytes); last column gives the total memory space for all traces (in bytes).

at this exit stub.

For DynamoRIO-with-LogPC traces are better utilized than in the DynamoRIO's case. For example, the two hottest traces are almost always completely used. Moreover, these traces are also longer than in the DynamoRIO case.

6.2.2 Trace Length

Table 6.2 presents traces characteristics for DynamoRIO and DynamoRIO-with-LogPC without optimization. The leftmost column is the benchmark name. For each of configuration, the first column gives the number of traces; columns 2,3,4 are the mean, maximum and minimum length of the traces (in bytes); and the last column gives the total memory space for all traces (in bytes). From the table we can see that in DynamoRIO we have many short traces, while in DynamoRIO-with-LogPC we have a smaller number of relatively long traces.

Note that the basic-blocks' characteristics are the same for DynamoRIO and DynamoRIO-with-LogPC, because the basic-block infrastructure is the same. The only basic-blocks that differ are those that contain logical CTIs.

6.2.3 Discussion

The qualitative results, described in this section for the sieve2 benchmarks are similar to the ones for other benchmarks on TinyVM and Kaffe. As we anticipated, DynamoRIO-with-

LogPC generates long traces that are well utilized. However, some traces get extremely long, especially when they include native loops. In Kaffe benchmarks, for example, we get huge traces. The total memory space for the traces (after applying traditional optimizations) is large: 1.24Mb, 3.83Mb, 1.71Mb for the compress, mpeg, and db benchmarks, respectively. Two factors contribute to these results: Kaffe is a more complex VM than TinyVM, and the SpecJVM98 benchmarks are longer programs than TinyVM's benchmarks.

On one hand, it is beneficial to have frequently accessed long traces. On the other hand, when the total trace size is too large to fit in the cache, the benchmark execution time of DynamoRIO-with-LogPC is even longer than their execution time under DynamoRIO. This degrading performance can be seen in Figure 6-2 when running DynamoRIO-with-LogPC on Kaffe. However, as we discussed in chapter 5, an additional advantage of the traces generated in DynamoRIO-with-LogPC is that they can be optimized. Thus, in the next section we present different optimization techniques and their impact on both execution time and memory space required.

6.3 DynamoRIO-with-LogPC and Optimizations

In this section we demonstrate the effects of the different optimization techniques, presented in chapter 5. The first subsection deals with traditional optimizations: constant-propagation, dead-code-elimination, and call-return-matching. The second subsection investigates the behavior of two optimization techniques that prevent generation of long traces: "max basic blocks" and "not-inline." We show memory characteristics that demonstrate the efficiency of these techniques. In the third subsection we show the results of implementing functional memorization on Kaffe.

6.3.1 Traditional Optimization Techniques

This subsection analyzes the contribution of each of the traditional optimization techniques, and investigates the behavior of DynamoRIO-with-LogPC with these optimizations.

Figure 6-4 presents the breakdown of the contribution of each optimization technique,

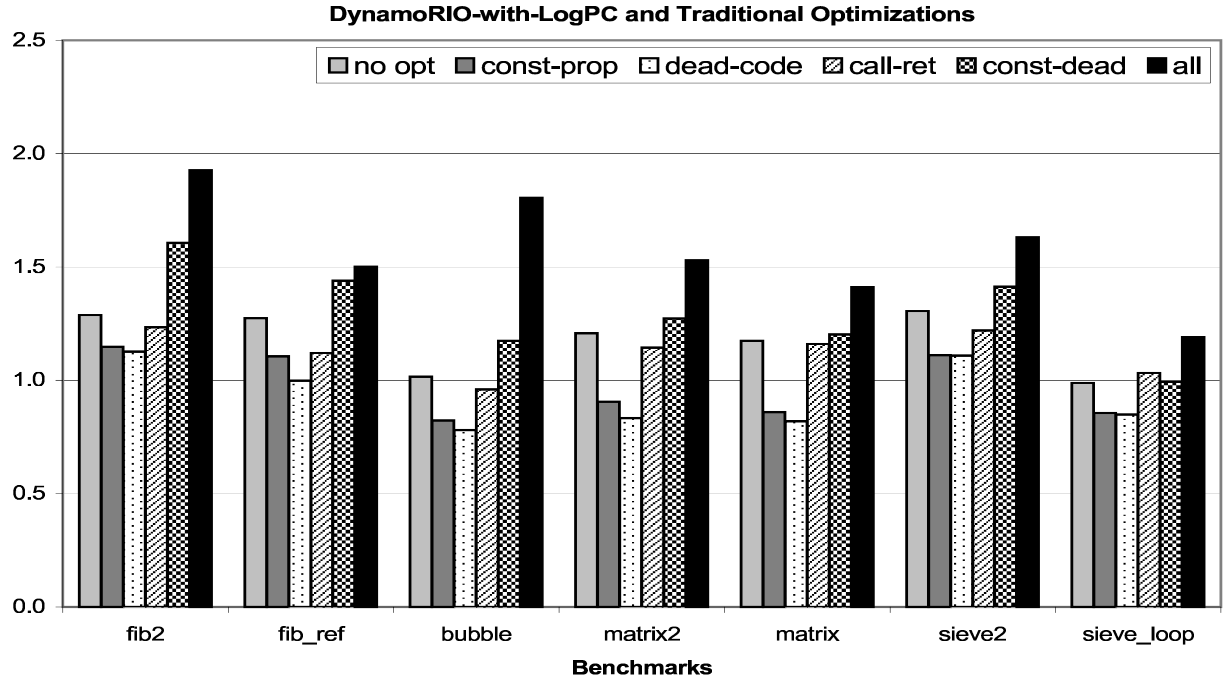

Figure 6-4: Contributions of separate optimizations, normalized against native execution. The configurations for DynamoRIO-with-LogPC are with: no optimizations (no opt), constant-propagation (const-prop), dead-code-elimination (dead-code), call-return-matching (call-ret), constant-propagation and dead-code-elimination (const-dead), and finally all of these optimizations together (all).

normalized to the native execution time. We see that each of the *constant-propagation, dead-code-elimination* and *call-return-matching* by itself does not necessary improve performance. Call-return-matching is the optimization technique that most significantly improves performance when running alone. Although this characterization is program dependent, in all programs the most substantial improvement is from using *all* of the optimizations.

There are several reasons for this improvement. Each optimization improves performance at the low level (instruction level): Constant-propagation removes unnecessary memory loads, dead-code-elimination removes unnecessary memory stores, and call-return-matching removes unnecessary indirect lookup.

At the high level (trace level), the combined optimizations remove the interpretive overhead (including the switch lookup), and thus reduce trace size. It turns out that reducing traces' size substantially improves execution time. Table 6.3 shows the trace characteristics for DynamoRIO-with-LogPC with traditional optimization, for each of the TinyVM bench-

benchmark	#	mean (Kb)	max (Kb)	min (Kb)	total (Kb)	size improv.(%)	performance improv.(%)
fib2	3	7.056	13.89	0.6	21.17	68	49
fib_ref	6	1.115	1.45	0.44	6.69	69	42
bubble	5	9.062	16.95	1.84	45.31	67	46
matrix2	17	8.172	16.96	1.58	138.92	68	37
matrix	17	8.926	17.69	2.86	151.74	65	31
sieve2	6	5.432	15.3	0.17	32.59	68	42
sieve-loop	11	59.467	394.82	0.17	654.14	55	18

Table 6.3: Trace characteristics for DynamoRIO-with-LogPC with traditional optimizations only, on TinyVM benchmarks. The first column is the benchmark's name; second column is the number of traces; columns 3,4,5 are the mean, max and min of the traces, respectively; column 6 is the total memory space for all traces; columns 7 and 8 are the improvements in the total memory space required and in the performance achieved, respectively. These improvements are one minus the division of the results from DynamoRIO-with-LogPC with traditional optimizations by those from the DynamoRIO-with-LogPC with no optimizations case.

marks. The last two columns present the improvements (in percentage) in the total space required for traces and performance, in comparison with the DynamoRIO-with-LogPC with no optimizations, respectively. We see that the optimizations reduced memory space for traces by 66% on average and improve performance by 41% on average (on all the benchmarks other than the sieve-loop where perfromance improved by only 18%).

Although these traditional optimizations reduce traces' length, some traces are still very long. In the next subsection we investigate this issue and measure memory characteristics, in order to show that it harms performance.

6.3.2 Avoiding too Long Traces

As discussed in section 6.2.2, DynamoRIO-with-LogPC might create extremely long traces, which might harm performance. One of the major reasons for having long traces is the native unrolled loops. We used the “max basic blocks” and “not-inline” techniques (described in section 5.5) to avoid such loops in traces. Figure 6-5 presents, on a logarithmic scale, the total traces' size for the TinyVM Benchmarks, with the following configurations: DynamoRIO-with-LogPC without optimization (DynLog), with trace reduction optimizations only (DynLogRed), with traditional optimizations only (DynLogOpt), and with all

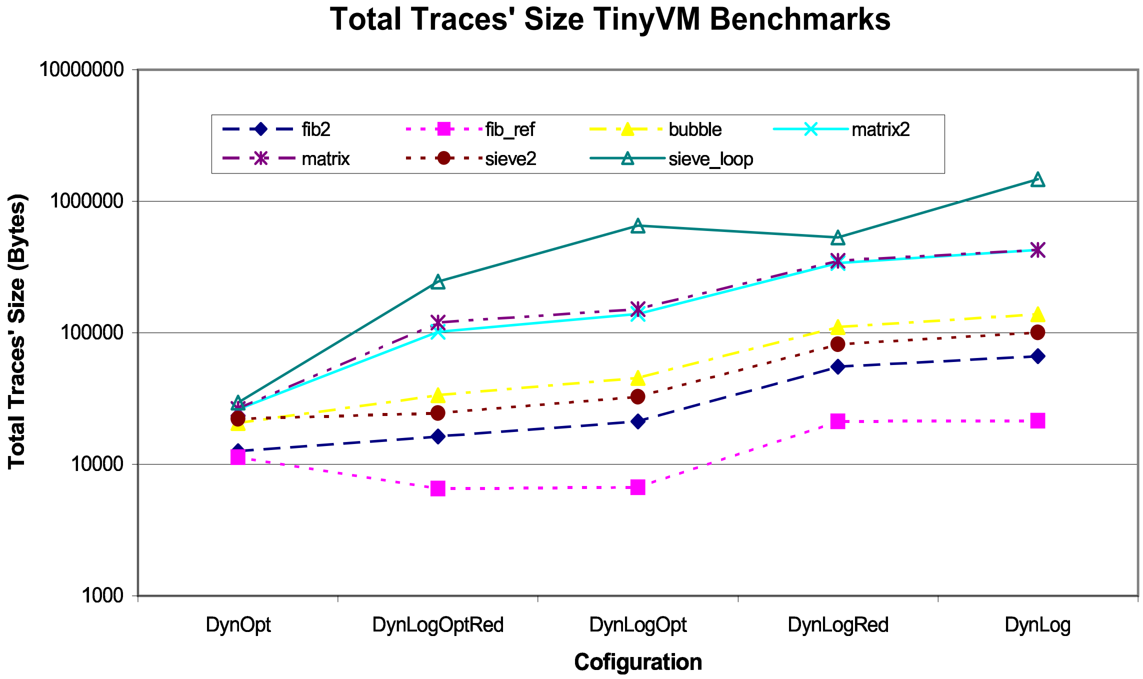

Figure 6-5: Total trace sizes of DynamoRIO-with-LogPC for the TinyVM benchmarks. We compare four different configurations of DynamoRIO-with-LogPC: with traditional optimizations and trace reduction (DynLogRed), only with traditional optimizations (DynLogOpt), only with trace reduction (DynLogRed), and without any optimizations (DynLog). Note that the y axis is with a logarithmic scale.

optimizations together (DynLogOptRed). The main point of this graph is that the sizes of total traces created by DynamoRIO-with-LogPC can be substantially reduced when using the optimization techniques. In this section we examine the traces created and memory usage. We also measure the efficiency of each of the trace reduction solutions, in both total trace size and performance.

Table 6.4 demonstrates the trace reduction methods for the sieve-loop in TinyVM. The table gives running time and lengths of the two longest traces. (Other traces are an order of magnitude shorter and therefore we ignore them.) In addition, it shows the number of cache misses for L1 and L2 cache. These numbers were obtained by using PAPI [40]. In our system, L1 cache contains 8KB for data cache and 8KB of instruction cache. The L2 cache's size is 256KB.

The first row in Table 6.4 presents the results of a run of DynamoRIO-with-LogPC and

	Time (sec)	Trace1(Kb)	Trace2(Kb)	L1 miscaches	L2 miscaches
With native loops	13.84	206.41	394.82	7,099,507	5,851,072
Trace size<1000BB	10.35	47.51	33.15	2,191,292	986,356
Pop not inlined	11.3	203.88	0.40	4,424,834	3,268,424
All not inlined	10.31	8.24	0.40	1,678,004	524,205

Table 6.4: DynamoRIO-with-LogPC and traditional optimizations, sieve-loop, avoiding too long traces. Traces' length, execution time and cache misses for different configurations.

optimizations (without any of the unrolled-loop avoidance techniques). The second row gives results from a similar run, in which traces are confined to be shorter than 1000 basic-blocks. (In our examples the performance was similar for a range of 500-1000 maximum basic-blocks in a trace). The third row gives the results without inlining the *pop* function of TinyVM. And the last row of the table shows results when we did not inline all the identified TinyVM functions that contain loops.

From the table we see that constraining the maximum basic-blocks in a trace shorten traces by an order of magnitude, and also improved performance by 25%. This supports the idea that too long traces harm performnaces. The not-inlining of the *pop* function substantially shortened one trace, and improved performance by 18%. And not inlining all the functions with loops led to shorter traces than with the max basic-blocks, and improved performance even further.

From the miscache analysis we see that there is a direct correlation between the sizes of the traces, the cache misses, and the performance time. Too long traces cause a higher number of cache misses, which increase the execution time. This is true in general, but it is more evident in cases where a trace is as big as the total cache size (such as the long traces in the sieve-loop example).

Table 6.5 presents trace characteristics for all TinyVM's benchmarks with traditional optimizations and not-inlining the *pop* function. On average this reduces memory space required for all traces by 79% (compare to 66% on average for the results of DynamoRIO-with-LogPC and traditional optimizations only). However, only the performance of sieve-loop was improved from not inlining the *pop* function. Moreover, from a detailed inspection[1]

[1]In sieve-loop, for example, using traditional optimizations in Table 6.3 the maximum trace is 394.82Kb,

benchmark	#	mean (Kb)	max (Kb)	min (Kb)	total (Kb)	size improv.(%)	performance improv.(%)
fib2	3	5.433	10.60	0.43	16.3	75	44
fib_ref	6	1.09	1.45	0.36	6.54	69	41
bubble	5	6.698	12.48	1.49	33.49	76	36
matrix2	17	5.985	12.38	1.40	101.74	72	31
matrix	17	7.032	15.59	2.19	119.55	76	26
sieve2	6	4.092	11.5	0.17	24.55	83	37
sieve-loop	11	22.234	203.88	0.17	244.57	79	31

Table 6.5: Trace characteristics for DynamoRIO-with-LogPC with traditional optimizations and "not-inline" the *pop* function, on TinyVM benchmarks. The first column is the benchmark name; second column is the number of traces; columns 3,4,5 are the mean, max and min of the traces, respectively; column 6 is the total memory space for all traces; columns 7 and 8 are the improvements in the total memory space required and in the performance achieved, respectively. These improvements are one minus the division of the results from DynamoRIO-with-LogPC with traditional optimizations and "not-inline" the *pop* function by those from the DynamoRIO-with-LogPC with no optimizations case.

we saw that the not-inlining reduced the total memory space for traces mainly by shortening the long traces.

Table 6.6 presents the total memory usage of traces for SpecJVM98 benchmarks under Kaffe. The table displays the DynamoRIO-with-LogPC and traditional optimizations, with and without not-inlining as the method for avoiding unrolled native loops. The functions we did not inline are *lookupClassField* and *findMethodLocal*. We also use these functions for functional memorization optimization, as shown in the next subsection. The results in this table support the results of Table 6.4. The total size of memory usage when not-inlining the functions is on average 64% shorter than when inlining these functions.

From the results of Tables 6.4 and 6.6 we see that too long traces harm performance. Both "max basic blocks" and "not-inline" can be used to prevent too long traces from occurring; and therefore they improve the performance of DynamoRIO-with-LogPC. The advantage of constraining the number of basic-blocks is that it is simple and therefore can be easily implemented for any VM. However, it is less elegant than the not-inlining method.

and the total traces space is 654.14Kb. In Table 6.5, after adding not-inlining of the *pop* function, the total size is 244.57Kb. From Table 6.4, we see that not-inlining the *pop* function reduces the length of Trace2 by 394.42Kb from 394.82Kb to 0.4Kb. This improvement causes most of the difference for the total traces space.

benchmark	without "not-inline" optimization (Kb)	with "not-inline" optimization (Kb)	size improv. (%)	performance improv. (%)
compress	1,242.5	801.7	64.5	35
mpeg	3,833.8	2,147.2	56	58
db	1,746.5	1,150.3	65.9	11

Table 6.6: Total memory space required for traces, for DynamoRIO-with-LogPC and traditional optimizations, with and without not-inline optimization. The last two columns present the total memory space and performance from without 'not-inline' to with 'not-inline' (in %), respectively. These improvments are one minus the division of the with 'not-inline' results by the ones from the wihtou 'not-inline'.

We attribute the fact that not-inlining harmed the performance of some of the TinyVM benchmarks to the fact that TinyVM and its benchmarks are "toy examples." Thus, we suspect that, for real interpreters and programs, not-inlining of native loops should be beneficial (as demonstrated by Kaffe's benchmarks).

6.3.3 Functional Memorization

We used functional memorization technique in Kaffe, in addition to the not-inline instrumentations. As described in section 5.6, this technique reduces trace length, and saves time by eliminating part of the execution code.

In Figure 6-6 we present the execution time with functional memorization, when possible, for the DynamoRIO-with-LogPC with and without not-inline instrumentations. Using part_eval improves performance by 30-50%, despite the fact that total traces' length for this case is almost identical to the length in the case of "not-inline", as shown in Table 6.6.

6.4 Discussion

In this chapter we presented experimental results from Kaffe and TinyVM for both execution time and memory usage (trace length). The two chosen VMs are very different in their functionality. TinyVM is a simple stack VM; and Kaffe is used for a dynamic language (Java), and thus includes more dynamic overhead (such as method dispatch upon invocation, array check etc). We show performance improvements for both small programs as the TinyVM

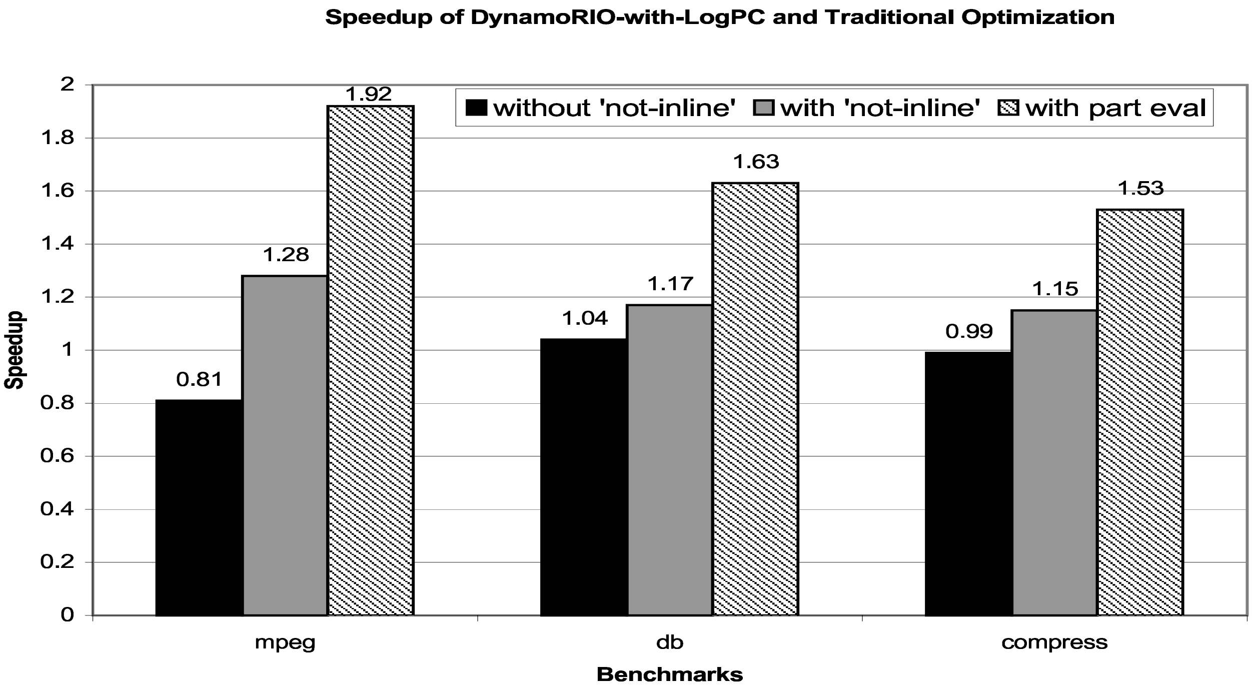

Figure 6-6: Running Times of DynamoRIO-with-LogPC and traditional optimizations, with and without not-inline optimization, and with part-eval optimization, normalized against native Execution.

benchmarks, where native execution times are less than a minute, and for very large programs as the Kaffe benchmarks, where native execution times are tens of minutes.

We started with presenting results of using DynamoRIO-with-LogPC without optimizations. From these results, we saw that the DynamoRIO-with LogPC generates, as we hoped, long traces which are heavily utilized.

We then demonstrated the performance improvement obtained by using optimization techniques. From experimenting with traditional optimizations, we learned that each one of the traditional optimizations techniques might slow the program run time; however, using all of them together improves performance of DynamoRIO-with-LogPC.

We discover that the generation of too long traces is a main problem in the DynamoRIO-with-LogPC (especially on the Kaffe VM). Therefore, we supplied two possible solutions to solve this problem, and demonstrated their efficiency. We emphasize that the objective of the solutions presented here was mainly to show that shortening of huge traces can be done and it substantially improves performance. We consider possibilities to improve performance of the DynamoRIO-with-LogPC by shortening traces as a main research direction in the future (we shortly discuss this in the Summary chapter).

We have also shown that the use of functional memorization (in Kaffe) reduces trace length and saves time by eliminating part of the execution code.

The techniques described in this thesis are implemented by adding simple instrumentations to the interpreter. These instrumentations require a minimal understanding of the interpreter's design. Unlike JITs, they do not require changing the code or adding new code, both of which complicate the original program. Admittedly, JITs are better suited for dynamic languages, but they are platform and VM dependent. However, since DynamoRIO-with-LogPC is based on the DynamoRIO infrastructure, it controls all the program execution; and, thus allows for implementing many optimizations that are used by JITs, such as register-allocation.

Finally, we saw that DynamoRIO-with-LogPC obtained on average a speedup of 55% on TinyVM's benchmarks and of 70% on Kaffe running some SpecJVM98 benchmarks. Based on these results we claim that there is high potential to dynamically optimize interpreters using DynamoRIO-with-LogPC.

Chapter 7

Summary

7.1 Thesis Overview

In this thesis, we provided a proof-of-concept for applying dynamic optimization in software for interpreters and VMs. We presented DynamoRIO-with-LogPC, which is based on the DynamoRIO infrastructure, as the dynamic optimizer. The experimental results showed that this technique improves the performance of the two interpreters investigated: TinyVM and Kaffe.

Chapter 1 introduces the notions of compilers and interpreters and explains why interpreters are typically slower than compiled programs. It also presents this thesis' objectives and outlines the road map to achieve them. In chapter 2 we surveyed different approaches to improve interpreters' performance. We presented an overview of run-time code generation, dynamic compilation, JITs and other techniques. One additional optimization technique is dynamic optimization. It differs from the previous techniques, as it optimizes the executable code during runtime, rather than compiling or translating the code. Therefore, chapter 3 discussed the dynamic optimization technique, and gave an overview of DynamoRIO as the dynamic optimizer that we use. DynamoRIO identifies and collects traces, which are native sequences of frequently executed code, during runtime. Then it optimizes these traces, and emits them to the cache. This process improves the performance of dynamic applications; however, it is not effective for interpreters and VMs, because it generates traces for the interpreters rather than for the applications (the interpreted programs). These traces are typically short and not well utilized.

In chapter 4, we explained how to modify DynamoRIO such that it would support optimization of interpreters and VMs. We designed DynamoRIO-with-LogPC to target the interpreted application instead of the interpreter. The use of DynamoRIO-with-LogPC requires simple annotations to the interpreters. These annotations provide DynamoRIO-with-LogPC with information regarding the high-level (interpreted) program's representation. Thus, DynamoRIO-with-LogPC creates traces that correspond to frequently used code sequences in the application. These traces are longer and better utilized than the ones generated by DynamoRIO.

In chapter 5, we expanded the optimization capability of DynamoRIO-with-LogPC. We outlined three classes of optimizations: "traditional" (used by DynamoRIO), avoiding too long traces, and functional memorization. Some of these optimizations may benefit from adding optional annotations to the interpreters. The first class is enhancing the traditional optimization techniques: call-return-matching, dead-code-elimination, and constant-propagation. To improve the efficiency of constant-propagation and dead-code-elimination we used immutable regions and newly defined trace-head constants (for both global and stack-allocated addresses).

For the second class we first explained the need to avoid too long traces. Then, we proposed two techniques to shorten traces. The first one is to constrain the number of basic-blocks in a trace to be smaller than a given maximum. However, neither one of them is perfect. The third class, functional memorization, was motivated by the not-inline technique. For specific types of functions, we can use a more aggressive optimization: we can completely eliminate the function call and replace it with its returned value.

In chapter 6 we presented extensive experimental results from Kaffe and TinyVM interpreters, for both execution time and memory usage (trace length).

We compared the results of DynamoRIO-with-LogPC with and without optimizations and of DynamoRIO to the native run time. For TinyVM, this comparison showed that while DynamoRIO is typically slower than native (by 20%), the DynamoRIO-with-LogPC is faster than the native execution time (by $20\% - 30\%$). DynamoRIO-with-LogPC with optimizations is by $30\% - 80\%$ faster than the native run time. For Kaffe, DynamoRIO and DynamoRIO-with-LogPC are slower than the native run time; however, DynamoRIO-with-LogPC and optimizations is by an average of 70% faster than the native run time.

These results show that performance improvement can be obtained with DynamoRIO-with-LogPC for both small programs as the TinyVM benchmarks, where native execution times are less than a minute, and large programs as the SpecJVM98 benchmarks, where native execution times are tens of minutes.

Finally, in the next section we outline possible directions for future research. Such extensions are required in order to further simplify the instrumentations provided in this thesis and to further improve performance of DynamoRIO-with-LogPC.

7.2 Future Work

DynamoRIO-with-LogPC achieves good results. However, there is room for additional improvement: in the optimizations it performs, in the core implementation of the system, and in the instrumentations it provides to the interpreter.

7.2.1 Additional Optimization Techniques

DynamoRIO-with-LogPC does not handle long loops within single bytecodes or functions. These loops are unrolled and inlined in the trace, increasing its size. In this thesis we provided two methods to avoid long traces: maximum basic-blocks and not-inline. Forcing a trace to be shorter than a maximum number of basic-blocks loses a big portion of the trace which could be optimized. In addition, it does not use the end of the trace, where traces can be linked in a direct logical CTI. The second method is more suitable to DynamoRIO's approach, because it allows optimization to be performed on most of the trace (besides the specific function). However, this method is not general enough. For each VM, it requires to identify and annotate functions with loops. In addition, it adds an overhead to DynamoRIO-with-LogPC, when verifying whether a function call should not be inlined. The conclusion is that neither of these methods is sufficient. We would rather have a tool that identifies loops

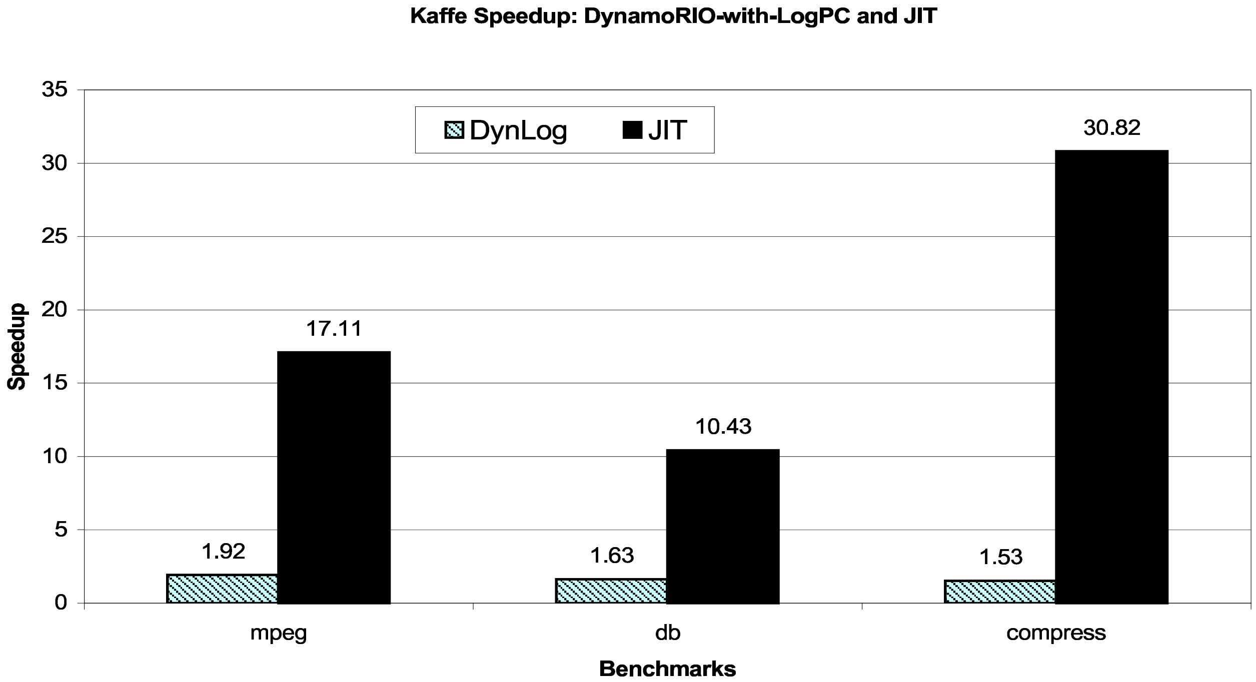

Figure 7-1: Performance comparison between DynamoRIO-with-LogPC and optimizations on Kaffe's interpreter and Kaffe's JIT, normalized to the native execution time of Kaffe's interpreter.

in the trace and handles them in such a way that would keep traces short without "losing" code. After identifying such loops, we can replace back-jumps of the loops with regular jumps; Moreover, implementing this solution would violate the current assertion that traces have one entry and multiple exits.

Dynamic dispatch, for example, adds a level of indirection to the method invocation. This dynamic dispatch is represented in our system by an indirect logical CTI. Functional memorization was found to be a beneficial tool in the optimization process. Its implementation has the disadvantage that it is interpreter-implementation dependent. We can annotate function calls to be replaced by the constant return value; However, if the relevant code does not reside in a single function, or if the function performs additional tasks that change memory and data structures, we cannot perform this optimization. Thus, future research is needed to generalize the use of functional memorization in traces.

Figure 7-1 compares the performance of DynamoRIO-with-LogPC and optimizations on the Kaffe interpreter to the Kaffe JIT's results. Although DynamoRIO-with-LogPC achieves a speedup of almost two in comparison to the native run time, it is still slower than the JIT by 10-15 times. Similar results hold for the Ocaml JIT. In order to obtain better performance on Java VMs, we should adopt some of the optimizations used in JITs. From the results presented in [1] for the Intel JIT, we learned that the register allocation has the largest impact on the overall performance.

Register allocation of local variables within methods can be implemented in DynamoRIO as part of the trace optimization techniques. While this is an expensive optimization technique, a good analysis should produce very good results on Java VMs. In Kaffe for example, the stack pointer and locals pointer are read and stored for almost each bytecode. It would be very efficient to allocate these addresses to registers. However, there is a trade-off between the analysis time and the performance gain. The simplest analysis, would be to find the top number of variables with the highest static reference counts (the number corresponds to the number of registers to allocate). This method is cheap, but it can allocate only a limited number of variables, even if they have non-overlapping life ranges. More sophisticated algorithms for register allocation are described in the literature and in JIT implementations [1], [29] and include priority metrics to determine which variables to allocate to registers. Copy-propagation can also be applied to reduce the number of memory accesses.

Another issue that needs to be addressed is the integration of DynamoRIO-with-LogPC with various types of garbage collection (GC), because some of them move live objects to new locations in memory. For these GCs DynamoRIO-with-LogPC needs to track these locations if it uses the objects addresses for constant-propagation or functional memorization, and make it a general optimization tool for interpreters.

Lastly, we can use DynamoRIO-with-LogPC for stack management of the VM's stack. It can use the native stack instead of the VM stack. This will improve the time for loading (storing) information from (on) the stack, which are the basic operations of a stack VM.

7.2.2 Core Improvements

There are a number of straightforward engineering improvements that should improve the performance of the system. When DynamoRIO-with-LogPC ends a trace on an indirect logical CTI, it returns to the basic-block execution, looks up for the next *Abstract PC*, and checks if it is a trace. DynamoRIO, in contrast, adds the lookup code to the trace, when encountering an indirect branch in the trace. Thus, it can check if the target is an existing trace in the cache, and jump to it directly, avoiding the time consuming context-switch. We can adopt a similar approach in ending traces with an indirect logical CTI. An additional improvement would be to inline (in the trace) the first number of trace targets, as an if-else statement. Only if the target is not one of the inlined targets, DynamoRIO-with-LogPC would perform a lookup.

An additional improvement can be done to logical CTIs at the end of traces where there is no linking. If the target is not a trace, we can directly jump to the right basic-block; instead of returning to basic-blocks execution mode, and then look up the target.

DynamoRIO-with-LogPC introduced high overhead into DynamoRIO, because it maintains a database keyed by the *Abstract PC*, and not just the *PC*; it also needs to obtain and track changes to the *Abstract PC* and to update the tables accordantly. In addition, we saw that optimizations are needed to be applied to traces in order to yield good performance. These tasks of monitoring and optimizing can be offloaded to other processors in a Symmetric Multi-Processing (SMP) system. Performing optimizations on a different thread would allow for more aggressive optimizations.

7.2.3 Simplifying the Instrumentations

In this work, we rely on the interpreter annotations as a tool to communicate high-level information to DynamoRIO-with-LogPC. The more information is provided, the more efficient is the optimization that can be performed. But adding annotations is not a trivial task; it requires some basic understanding of the interpreter. Finding immutable regions, for instance, can improve the results of the constant-propagation optimization and in some cases it is crucial for removing the interpreter's dispatch loop overhead from the trace. However, finding all of the immutable addresses is challenging. Declaring unaliased addresses has a similar problem. DynamoRIO could use profiling information by instrumenting all loads and stores, and studying the memory behavior of the program. This would simplify the interpreter annotation burden and would automate much of this process.

7.3 Results

Interpreters are widely used in domain-specific, dynamic, and scripting languages. While their implementation is relatively simple, their performance is poor. Optimizing interpreters is a difficult task, because the information regarding the interpreted program is not known until the interpreter is run. Thus, replacing the interpreter implementation with a simple compiler is not feasible. Moreover, many of the languages that are implemented with interpreters or VMs include dynamic features such as dynamically dispatched calls, polymorphism and user-defined control structures that make it difficult or impossible for a compiler to perform static optimizations.

Efficient implementations of interpreters (such as JITs) require a runtime system that would perform optimization dynamically, using program specification. These solutions incur high engineering cost, which in most cases offsets the reasons for using the interpreted language in the first place (speed and ease of software development). Furthermore, such implementations are usually language-specific and hardware-dependent, and are rarely portable or reusable. Thus, the motivation for this research is the need to optimize interpreters and VMs, without incurring the costs of developing an efficient implementation.

We created a single software dynamic optimizer that can be applied to different interpreters. Our system, DynamoRIO-with-LogPC is based on DynamoRIO's infrastructure with the appropriate changes to fit interpreters. DynamoRIO is a dynamic optimization tool, jointly developed at HP Labs and MIT. While it is very efficient in a broad range of applications, typical interpreter implementations foil DynamoRIO's strategy of trace collection. In addition, DynamoRIO's low-level optimization is unable to extract sufficient high-level semantics of the interpreted program to perform many of the necessary optimizations.

We designed and implemented DynamoRIO-with-LogPC in three phases. In the first one, we modified DynamoRIO such that it would target the interpreted program rather than the interpreter. In the second phase, we provided an interface for communicating high-level application information from the interpreter to DynamoRIO-with-LogPC. In the third phase, we implemented additional optimization techniques in the new system.

We demonstrated the use of DynamoRIO-with-LogPC by improving the execution times of benchmarks on two very different interpreters: Kaffe and TinyVM. Our experimental results are encouraging. They suggest that DynamoRIO-with-LogPC has a high potential for improving performance of interpreters in general. Using simple instrumentations that require only a little familiarity with the interpreter, we succeeded in improving the native run times of the interpreters by 50-90%. This provides a proof-of-concept for applying dynamic optimization in software to interpreters. While these results are not comparable with the ones of a hand-crafted native compiler, we believe that this system provides an appealing point on the language implementation spectrum.

Appendix A

List of Instrumentation API

In this appendix we summarize all of the instrumentations presented in this thesis. Some of the instrumentations are mandatory for the correctiveness of DynamoRIO-with-LogPC (as presented in chapter 4); the others are optional, and are used for the trace optimization techniques (as described in chapter 5). We represented each function by an entry that includes: Name- The function's name; Arguments - The set of parameters that need to be passed to the function; Mandatory - A flag that indicates if the function is mandatory or optional; Description - A description of the function's role and operation; Constraints - Any constraints or conditions of use; and Example - An example for the function's use.

The functions are presented in alphabetical order and each one begins on a new page. Table A.1 summarizes the API functions based on their use.

Used for	Name	Mandatory
Trace Creation	dynamorio_set_logical_pc	Yes
	dynamorio_logical_direct_jump	Yes
	dynamorio_logical_indirect_jump	Yes
Constant Propagation Dead Code Elimination	dynamorio_set_region_immutable	No
	dynamorio_add_trace_constant_address	No
	dynamorio_set_trace_constant_stack_address	No
	dynamorio_unaliased_address	No
	dynamorio_unaliased_stack_address	No
Reduce Trace Size	dynamorio_set_call_no_inline	No
Functional Memorization	dynamorio_part_eval_func	No

Table A.1: Instrumentation API divided to categories of use.

dynamorio_add_trace_constant_address

Argument void *addr

Mandatory No

Description This function indicates to DynamoRIO-with-LogPC that *addr* is an address, which its content is a constant with respect to the trace head from which the address is accessed (i.e., at the beginning (head) of the trace, this address would have a fixed known value). This address is often a function of the Logical PC.

Constraints Trace-head constant addresses must not be aliased.

Example Assuming *pc* is a global address and is a trace-head constant address. Then:

dynamorio_set_logical_pc(pc);
dynamorio_logical_direct_jump();
dynamorio_add_trace_constant_address(&pc);
goto loop;

dynamorio_logical_direct_jump

Argument None

Mandatory Yes

Description This function notifies DynamoRIO-with-LogPC about a control flow change in the interpreted application. It should be used in cases where the control flow is direct. This means, that each time the control flow reaches this call with a specific logical PC, it would go to the same target logical PC.

Constraints Should be placed immediately after the dynamorio_set_logical_pc function.

Example Assuming *pc* is the logical pc and that the change of control flow is direct. Then:

case JMPOP:
...
pc = arg;
dynamorio_set_logical_pc(pc);
dynamorio_logical_direct_jump();
goto loop;

dynamorio_logical_indirect_jump

Argument None

Mandatory Yes

Description This function notifies DynamoRIO-with-LogPC about a control flow change in the interpreted application. It should be used in cases where the control flow is indirect. This means, that each time the control flow reaches this call with a specific logical PC it might take different targets.

Constraints Should be placed immediately after the dynamorio_set_logical_pc function.

Example Assuming *pc* is the logical pc and that the change of control flow is direct. Then:

```
case RETOP:
...
dynamorio_set_logical_pc(pc);
dynamorio_logical_indirect_jump();
goto loop;
```

dynamorio_part_eval_func

Argument int return_value

Mandatory No

Description This function indicates DynamoRIO-with-LogPC to replace the function call in a trace with its return-value. The call should be located immediately after the function call, and the function should be annotated as "not-inline" using the dynamorio_set_call_no_inline.

Constraints The function should obey the following rules:

1) The function's parameters are constants with respect to the trace.

2) The returned value is the same for each execution of the function with the same parameters.

3) The function has no side effects.

Example Assume function *lookupClassField* obeys all the constraints, and thus, can be annotated for functional memorization. Then:

```
...
field = lookupClassField(class, ...,...);
dynamorio_part_eval_func((int)field);
...
```

dynamorio_set_call_no_inline

Argument byte *func_address

Mandatory No

Description This function instruct DynamoRIO-with-LogPC not to inline a specific function (in order to avoid loops or for functional memorization). The function is passed by its address.

Constraints None

Example Assume *pop* is a function that contains a loop, and thus we want to not inline it. Then:

VM initialization code ...
dynamorio_set_call_no_inline(&pop);

dynamorio_set_logical_pc

Argument int logical_pc

Mandatory Yes

Description This function passes to DynamoRIO-with-LogPC the value of a unique identifier into the control flow structure of the interpreted application (the "Logical PC"). It should be called immediately after a non-sequential change to the Logical PC.

Constraints None

Example Assuming *pc* is the "Logical PC". Then:

```
case CALLOP:
...
pc = arg;
dynamorio_set_logical_pc(pc);
...
goto loop;
```

dynamorio_set_region_immutable

Argument int start, int end

Mandatory No

Description This function indicates to DynamoRIO-with-LogPC that the region of memory, delimited by *start* and *end* (inclusive), is immutable (i.e., the content of the addresses in that region do not change, after the function call). This information takes effect immediately after the call.

Constraints None

Example Assuming that the array of bytecodes of the program *insts*, is immutable. Then:

VM initialization code ...
*dynamorio_set_region_immutable(insts,(insts+nInst*sizeof(ByteCode)-1));*
dynamorio_set_region_immutable(&insts,(&instss+sizeof(ByteCode)-1));*
...

dynamorio_set_trace_constant_stack_address

Argument void* addr, int value

Mandatory No

Description This function indicates to DynamoRIO-with-LogPC that *addr* is a stack-allocated address, which its content (*value*) is a constant with respect to the trace head from which the address is accessed (i.e., at the beginning, head, of the trace, this address would have a value of *value*). This address is often a function of the Logical PC.

Constraints Trace-head constant addresses must not be aliased.

Example Assuming *pc* is a stack-allocated address and its a trace-head constant address. Then:

```
case CALLOP:
...
pc = arg;
dynamorio_set_logical_pc(pc);
dynamorio_logical_direct_jump();
dynamorio_set_trace_constant_stack_address(&pc, pc);
goto loop;
```

dynamorio_unaliased_address

Argument void *addr

Mandatory No

Description This function indicates that the address *addr* is an unaliased memory address. This information is used by DynamoRIO-with-LogPC to perform constant propagation and dead code elimination on this address.

Constraints None

Example Assuming *arg* is an unaliased memory address. Then:

dynamorio_set_logical_pc(pc);
dynamorio_logical_direct_jump();
dynamorio_unaliased_address(&arg);
goto loop;

dynamorio_unaliased_stack_address

Argument void *addr

Mandatory No

Description This function indicates that the address *addr* is a stack-allocated unaliased memory address. This information is used by DynamoRIO-with-LogPC to perform constant propagation and dead code elimination on this address.

Constraints None

Example Assuming *arg and op* are both unaliased stack-allocated memory addresses. Then:

```
case CALLOP:
...
pc = arg;
dynamorio_set_logical_pc(pc);
dynamorio_logical_direct_jump();
dynamorio_unaliased_stack_address(&arg);
dynamorio_unaliased_stack_address(&op);
goto loop;
```

Appendix B

Main Functions of TinyVM

```
/*  tinyvm.c

Tinyvm -- A simple stack-based VM.

Bytecodes:   arguments (n) are integers.
CALL    n         -- jump to nth instr; pushes old FP, sets up new frame, FP
RETURN  n         -- return   - restores FP, cleans up stack assuming n arg.s
JUMP    n         -- jump to nth instr
BEQ     n         -- tests top of stack, jumps to nth instr if 0 ; pops SP
DUPE    n         -- duplicate SP-n on top of stack
DUPEARG n         -- duplicate FP - (2 + n) on top of stack
DUPELOC n         -- duplicate FP + n + 1 on top of stack
DUPEDEF n         -- duplicate stack_bottom + n + 1 on top of stack
POP     n         -- pop n values from stack
POPARGS n         -- pop n values from under the top (value) of the stack
INT     n         -- pushes integer n on stack
PRIMCALL   n      -- calls primitive number n; answer val returned on stack
-- primop is responsible for popping argument values.
SET     n         -- sets nth elt below SP to value on top of stack; pops SP
SETARG  n         -- sets (n + 2)nd elt below FP to value on top of stack; pops SP
SETLOC  n         -- sets FP + n to value on top of stack; pops SP
SETDEF  n         -- sets stack_bottom + n to value on top of stack; pops SP
EXIT    n         -- exit the VM with status n

Stack starts at 0; push increments SP.
Execution starts at 0th instruction.
Data is tagged:  rightmost bit is 1 if data is immediate (int),
0 if a pointer (into C heap).
- allocated data is managed with reference counting -- freed when
refs drop to 0.  (note that circular structures are never collected).
- allocated data has header with reference count and array size.

Calling sequence for call f(a, b), PC starts at p:
[...]               SP = n,     FP = m (m < n), PC = p
p:   push b  => [..., b]            SP = n+1,   FP = m,         PC = p+1
p+1: push a  => [..., b, a]         SP = n+2,   FP = m,         PC = p+2
p+2: call f  => [..., b, a, m, p+3] SP = n+4,   FP = n+4,       PC = f's offset
<body-of-f> => [..., b, a, m, p+3, ..., val] SP = n+5, FP = n+4, PC = f+<sz-of-f>
f+?: return 2  => [..., val]        SP = n+1,   FP = m,         PC = p+3

- note that frame offsets (from base of stack) are what's actually stored in frame,
because stack may be relocated as it grows.
- *fp = return pc, *(fp - 1) = frame offset for caller fp,
*(fp - 2) = 1st arg., etc.

*/
```

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stddef.h>
#include <unistd.h>
#include <time.h>

#include "tinyvm.h"

#ifdef DYNAMORIO
#include "dynamorio.h"
#endif

#define INITIAL_STACK_SIZE 10

    ByteCode* instrs; /* global array of bytecodes */
int num_instrs; /* number of bytecodes in instrs */

label *labels = NULL; /* name/offset info - for debugging */
int num_labels = 0;
char *primop_names = NULL; /* primop_names[n] is name for primop n */
int num_primops = 0;

value eval () {
     int pc = 0;
     short op;
     int arg;
     div_t q;
     value val, old_val, ref_val; /* values from stack */
     int n, fo, next_pc, saved_fo;
     value *next_fp, *sp1;

     debug(DBG_EVAL, (fprintf(stderr, "eval: "), dump_stack()));

loop:
     if (pc >= num_instrs) {
          debug(DBG_EVAL, (fprintf(stderr, "eval - eof\n")));
          return(*sp);
     }
     op = instrs[pc].op;
     arg = instrs[pc].arg;
     switch (op) {
     case CALLOP:
          debug(DBG_EVAL, fprintf(stderr, "eval call %d (%s)\n", arg, find_label(arg)));
          fo = fp - stack; /* convert fp to offset */
          push_raw(fo); /* and save on stack */
          push_raw(pc + 1); /* push return PC on stack */
          fp = sp; /* set new FP (points at return PC) */
          pc = arg; /* go to start of function body */

#ifdef DYNAMORIO_LOGPC
          dynamorio_set_logical_pc(pc);
          dynamorio_logical_direct_jump();
#ifdef DYNAMORIO_ANNOT
          dynamorio_set_trace_constant_stack_address(&pc, pc);
          dynamorio_unaliased_stack_address(&arg);
          dynamorio_unaliased_stack_address(&op);
#endif
#endif

  goto loop;
     case RETOP:
          debug(DBG_EVAL, fprintf(stderr, "eval return\n"));
          if (fp > stack) { /* if there is a call stack, */
               val = *sp; /* get returned value */
               sp--;   /* don't pop that value (don't want to free it) */
               pop(sp - fp); /* pop down to fp (pop(1) while sp > fp) */
```

```
            pc = pop_raw(); /* pop the return PC */
            fo = pop_raw(); /* pop caller's frame offset */
            debug(DBG_EVAL, fprintf(stderr, "  pc=%d, fo=%d\n", pc, fo));
            fp = stack + fo;
            *(++sp) = val; /* put return value back on stack */
```

#ifdef DYNAMORIO_LOGPC
dynamorio_set_logical_pc(pc);
dynamorio_logical_indirect_jump();
#ifdef DYNAMORIO_ANNOT
dynamorio_set_trace_constant_stack_address(&pc, pc);
dynamorio_unaliased_stack_address(&arg);
dynamorio_unaliased_stack_address(&op);
#endif
#endif

```
            goto loop;
        } else {  /* here FO is 0 -- we're done with program */
            return *sp;
        }
    case BEQOP:
        debug(DBG_EVAL, (fprintf(stderr, "eval beq %d (%s) top=", arg, find_label(arg)),
              dump_value(*sp), fprintf(stderr, "\n")));
        n = pop_int(); /* top must be an int */
        if (n) { /* not equal 0 */
            pc++; /* just continue */
        } else { /* equal 0 */
            pc = arg; /* do the branch */
```

#ifdef DYNAMORIO_LOGPC
dynamorio_set_logical_pc(pc);
dynamorio_logical_direct_jump();
#ifdef DYNAMORIO_ANNOT
dynamorio_set_trace_constant_stack_address(&pc, pc);
dynamorio_unaliased_stack_address(&arg);
dynamorio_unaliased_stack_address(&op);
#endif
#endif

```
        }
        goto loop;
    case JMPOP:
        debug(DBG_EVAL, fprintf(stderr, "eval jmp %d (%s)\n", arg, find_label(arg)));
        pc = arg;
```

#ifdef DYNAMORIO_LOGPC
dynamorio_set_logical_pc(pc);
dynamorio_logical_direct_jump();
#ifdef DYNAMORIO_ANNOT
dynamorio_set_trace_constant_stack_address(&pc, pc);
dynamorio_unaliased_stack_address(&arg);
dynamorio_unaliased_stack_address(&op);
#endif
#endif

```
        goto loop;
    case DUPEOP:
        assert((sp - arg) >= stack);
        val = *(sp - arg);
        debug(DBG_EVAL, (fprintf(stderr, "eval dupe %d, val= ", arg),
              dump_value(val), fprintf(stderr, "\n")));
        push(val); /* handles incr'ing refcounts */
        pc++;
        goto loop;
    case DUPEARGOP:
```

```
        assert((fp - 2 - arg) >= stack);
        val = *(fp - 2 - arg);
        debug(DBG_EVAL, (fprintf(stderr, "eval dupearg %d, val= ", arg),
              dump_value(val), fprintf(stderr, "\n")));
        push(val); /* handles incr'ing refcounts */
        pc++;
        goto loop;
    case DUPELOCOP:
        assert((fp + arg + 1) <= last_sp);
        val = *(fp + arg + 1);
        debug(DBG_EVAL, (fprintf(stderr, "eval dupeloc %d, val= ", arg),
              dump_value(val), fprintf(stderr, "\n")));
        push(val); /* handles incr'ing refcounts */
        pc++;
        goto loop;
    case DUPEDEFOP:
        assert((stack + arg + 1) >= stack);
        val = *(stack + arg + 1); /* a dummy value is at bottom of stack */
        debug(DBG_EVAL, (fprintf(stderr, "eval dupedef %d, val= ", arg),
              dump_value(val), fprintf(stderr, "\n")));
        push(val); /* handles incr'ing refcounts */
        pc++;
        goto loop;
    case POPOP:
        debug(DBG_EVAL, (fprintf(stderr, "eval pop %d, top=", arg),
              dump_value(*sp), fprintf(stderr, "\n")));
        pop(arg); /* handles decr'ing refs */
        pc++;
        goto loop;
    case POPARGSOP:
        debug(DBG_EVAL, (fprintf(stderr, "eval popargs %d, top=", arg),
              dump_value(*sp), fprintf(stderr, "\n")));
        popargs(arg); /* handles decr'ing refs */
        pc++;
        goto loop;
    case INTOP:
        debug(DBG_EVAL, fprintf(stderr, "eval int %d\n", arg));
        push(make_int(arg));
        pc++;
        goto loop;
    case PRIMCALLOP:
        debug(DBG_EVAL, (fprintf(stderr, "eval primcall %d (%s), top= ", arg,
              primop_name(arg)), dump_value(*sp), fprintf(stderr, "\n")));
        (*(primops[arg].funp))(); /* do the call */
        pc++;
        goto loop;
    case SETOP: /* [new_v, <n-1 elts>, old_v, ...] => [<n-1 elts>, new_v, ...] */
        val = *sp; /* new value */
        sp1 = sp - arg; /* lhs address */
        old_val = *sp1;
        if (is_ref(old_val)) {
            decr_refs(old_val);
        }
        if (is_ref(val)) {
            incr_refs(val);
        }
        debug(DBG_EVAL, (fprintf(stderr, "eval set %d, top= ", arg),
              dump_value(*sp), fprintf(stderr, "\n")));
        *sp1 = val;      /* do update */
        pop(1);          /* will decr refs of new val, if an addr */
        pc++;
        goto loop;
    case SETARGOP:
        assert((fp - 2 - arg) > stack);
        sp1 = fp - 2 - arg; /* lhs address */
        val = *sp;
        old_val = *sp1;
        if (is_ref(old_val)) {
            decr_refs(old_val);
        }
```

```
            if (is_ref(val)) {
                incr_refs(val);
            }
            debug(DBG_EVAL, (fprintf(stderr, "eval setarg %d, top= ", arg),
                  dump_value(*sp), fprintf(stderr, "\n")));
            *sp1 = val; /* do the update */
            pop(1);
            pc++;
            goto loop;
        case SETLOCOP:
            assert((fp + arg + 1) <= last_sp);
            sp1 = fp + arg + 1; /* lhs address */
            val = *sp; /* new val */
            old_val = *sp1;
            if (is_ref(old_val)) {
                decr_refs(old_val);
            }
            if (is_ref(val)) {
                incr_refs(val);
            }
            debug(DBG_EVAL, (fprintf(stderr, "eval setloc %d, top= ", arg),
                  dump_value(*sp), fprintf(stderr, "\n")));
            *sp1 = val; /* do the update */
            pop(1);
            pc++;
            goto loop;
        case SETDEFOP:
            assert((stack + arg + 1) <= last_sp);
            sp1 = stack + arg + 1; /* there's a dummy value on bottom of stack */
            val = *sp;
            old_val = *sp1;
            if (is_ref(old_val)) {
                decr_refs(old_val);
            }
            if (is_ref(val)) {
                incr_refs(val);
            }
            debug(DBG_EVAL, (fprintf(stderr, "eval setdef %d, top= ", arg),
                  dump_value(*sp), fprintf(stderr, "\n")));
            *sp1 = val; /* do the update */
            pop(1);
            pc++;
            goto loop;
        case EXITOP:
            // currently, exit code is ignored.
            debug(DBG_EVAL, (fprintf(stderr, "eval exit %d top=", arg),
                  dump_value(*sp), fprintf(stderr, "\n")));
            return(*sp);
        default:
            fprintf(stderr, "[eval] Invalid bytecode op '%d'.\nExiting...\n", op);
            exit(1);
        } /* switch (op) */
}

void print_usage(char** argv) {
    fprintf(stderr, "Usage: %s options source\n", argv[0]);
    fprintf(stderr, "    where source is a filename\n"
    "Options: \n"
    "    -? : print this message.\n"
    "    -a : input file is tiny assembler (default is binary)\n"
    "    -d filename : dump binary to filename\n");
}

int main (int argc, char *argv[])
{
    value val;
    int aflag = 0;
    int dflag = 0;
    char* dump_filename;
    char* in_filename;
```

```
    int c, i;
    clock_t start, end;

#ifdef DYNAMORIO
    dynamorio_app_init();
    dynamorio_app_start();
#endif

    opterr = 0;

    while ((c = getopt (argc, argv, "ad:")) != -1) {
        switch (c)
        {
        case 'a':
            aflag = 1;
            break;
        case 'd':
            dflag = 1;
            dump_filename = optarg;
            break;
        case '?':
            if (isprint (optopt))
                fprintf (stderr, "Unknown option `-%c'.\n", optopt);
            else
                fprintf (stderr, "Unknown option character `\\x%x'.\n", optopt);
            print_usage(argv);
            exit(1);
        default:
            abort();
        }
    }

    // one and only non-optional argument is name of input file -- load it
    if (optind < argc) {
        in_filename = argv[optind];
    } else {
        fprintf(stderr, "Missing source.\n");
        print_usage(argv);
        exit(1);
    }

    init_primops(); /* initialize primops */
    init_debug();

    if (aflag) {
        load_txt(in_filename); /* allocs & fills instrs, labels */
    } else {
        load_bin(in_filename);
    }

    if (dflag) {
        dump_bin(dump_filename);
    }

    // initialize stack
    stack = sp = fp = malloc(INITIAL_STACK_SIZE * sizeof(value));
    last_sp = stack + INITIAL_STACK_SIZE - 1;
    *sp = make_int(1);    /* stack starts with something on it */
```

#ifdef DYNAMORIO_LOGPC
#ifdef DYNAMORIO_ANNOT

// loops blow up traces
dynamorio_set_call_no_inline(&pop);

// bytecode (instrs) and number of
dynamorio_set_region_immutable(&num_instrs,((int)&num_instrs+sizeof(int)-1));
dynamorio_set_region_immutable(instrs,((int)instrs+num_instrs*sizeof(ByteCode)-1));
dynamorio_set_region_immutable(&instrs,((int)&instrs+sizeof(ByteCode*)-1));

```
    //primops and number of
    dynamorio_set_region_immutable(&num_primops, ((int)&num_primops+sizeof(int)-1));
    dynamorio_set_region_immutable(primops,((int)primops+num_primops*sizeof(primop)-1));
    dynamorio_set_region_immutable(&primops,((int)&primops+sizeof(primop*)-1));

    //debug flags
    dynamorio_set_region_immutable(&debug_flags, ((int)&debug_flags+sizeof(int)-1));

#endif
#endif

    start = clock();
    val = eval();
    end = clock();
    dump_value(val);
    printf("\ncpu time = %f\n", ((double) (end - start)) / CLOCKS_PER_SEC);

#ifdef DYNAMORIO
    dynamorio_app_stop();
    dynamorio_app_exit();
#endif

    exit(0);
}

/*  $Id: tinyvm.c,v 1.11 2003/05/21 20:37:52 gregs-ai Exp $ */
```

Bibliography

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time java compiler.

[2] L. O. Anderson. Program analysis and specialization for the c programming langauge. DIKU report No. 94/19, DIKU Deaprtment of Computer Science, University of Copenhagen, 1994.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, Oct. 2000.

[4] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–159, 1996.

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

[6] A. Bondorf. *Similix 5.0 Manual.* DIKU, University of Copenhagen, Denmark, May 1993. Included in Similix distribution, 82 pages.

[7] Portable, fast direct threaded code.

[8] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001.

[9] D. D. Chamberlin, M. M. Astrahan, W. F. K. III, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. G. Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. Support for repetitive transactions and ad hoc queries in system r. *TODS*, 6(1):70–94, 1981.

[10] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming laguage. In *In Proceedings of the SIGPLAN '89 Conference on Programming Languages Design and Implementation*, pages 140–160, Portland, OR, June 1989.

[11] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000.

[12] C. Consel. A tour of schism: A partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 145–154. New York: ACM, 1993.

[13] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. Tempo: specializing systems applications and beyond. *ACM Computing Surveys*, 30(3es), 1998.

[14] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Restone: An on-line program specializer. In *Proceedings of Hot Chips 11*, Aug. 1999.

[15] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *ACM Symposium on Principles of Programming Languages (POPL '84)*, Jan. 1984.

[16] D. Engler and T. Proebsting. vcode: a retargetable, extensible, very fast dynamic ocde generation system.

[17] D. R. Engler, W. C. Hsieh, and M. F. 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.

[18] D. R. Engler and T. 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.

[19] M. A. Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.

[20] R. Glück, R. Nakashige, and R. Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.

[21] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[22] W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Software - Practice and Experience*, 21(4):375–390, 1991.

[23] D. Inglass, T. Kaehler, S. W. John Maloney, and A. Kay. Back to the future: the story of squeak, a usable smalltalk written in itself. In *OOPSLA '97 conference proceedings*, pages 318–326, 1997.

[24] N. D. Jones. An introduction to partial evaluation. ACM Computing Surveys, sep 1996.

[25] N. D. Jones, C. Gomrade, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[26] N. D. Jones, P. Sestoft, and H. Sondergaard. Mix: A self-applicable partial evaluator fro experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[27] J. Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, 1991.

[28] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pages 258–268, Jan. 1992.

[29] I. Kazuaki. Design, implementation and evaluation of optimizations in a just-in-time compiler.

[30] D. Keppel, S. J. Eggers, and H. RObert R. A case for runtime code generation. Technical Report 91-11-04, University of Washington Department of Computer Science and Engineering, 1991.

[31] H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicable language: A theory and implementation in the case of prolog. In Ninth ACM Symposium POPL, Albuquerque, New-Mexico, 1982.

[32] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8), Mar. 2001.

[33] M. Leone and P. Lee. Lightweight Run-Time Code Generation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.

[34] X. Leroy. The objective caml system release 1.05, 1997.

[35] C. May. Mimic: A fast s/370 simulator. In *Proceedings of the 1987 ACM SIGPLAN Symposium on Interpreters and Interperetive Techniques*, pages 1–13. SIGPLAN Notices, 22(6), June 1987.

[36] S. microsystems. The java hotspot (tm) virtual machine, v1.4.1, d2, Sept 2002.

[37] E. Miranda. Brouhaha - a portable smalltalk interpreter. In *OOPSLA '87 conference proceedings, ACM Press*, pages 354–365, oct 1987.

[38] M. Mock, C. Chambers, and S. J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *International Symposium on Microarchitecture*, pages 291–302, 2000.

[39] C. H. Moore and G. C. Leach. Forth – a language for interactive computing. Technical report, Mohasco Industries, Inc., Amsterdam, NY, 1970.

[40] P erformance api papi. University of Tennessee.
`http://www.icl.cs.utk.edu/projects/papi/`.

[41] I. Piumarta and F. Riccardi. Optimizing direct-threaded code by selective inlining. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[42] M. Poletto, D. Engler, and M. Kaashoek. tcc: A template-based compiler for ‘c, 1995.

[43] L. L. Pollock and M. L. Soffa. Incremental compilation of locally optimized code. In *In Proceedings of the ACM SIGPLAN '84 Conference on Programming Language Design and Implementation*, pages 152–164, 1984.

[44] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture (MICRO '96)*, Dec. 1996.

[45] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.

[46] SPEC JVM Client98 benchmark. Standard Performance Evaluation Corporation.
`http://www.spec.org/osg/jvm98`.

[47] G. Sullivan, D. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. ACM Workshop on Interpreters, Virtual Machines and Emulators (IVME-03), san Diego, California, june 2003.