NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE PROGRAM
COMPUTER SYSTEMS TECHNOLOGY

MASTERS THESIS

Optimizing Dynamic Traces Using Symbolic Execution

Efthymios Chr. Hadjimichael

Supervisors: **Yannis Smaragdakis,** Professor UoA
George Fourtounis, Dr. Electrical & Computer Engineer

ATHENS

OCTOBER 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΕΧΝΟΛΟΓΙΑ ΣΥΣΤΗΜΑΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βελτιστοποίηση σε Δυναμικά Ίχνη με Συμβολική Εκτέλεση

Ευθύμιος Χρ. Χατζημιχαήλ

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Δρ. Ηλεκτρολόγος Μηχανικός & Μηχανικός Η/Υ

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2016

MASTERS THESIS

Optimizing Dynamic Traces Using Symbolic Execution

Efthymios Chr. Hadjimichael
R.N.: M1369

SUPERVISORS: **Yannis Smaragdakis,** Professor UoA
George Fourtounis, Dr. Electrical & Computer Engineer

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βελτιστοποίηση σε Δυναμικά Ίχνη με Συμβολική Εκτέλεση

Ευθύμιος Χρ. Χατζημιχαήλ
Α.Μ.: Μ1369

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Δρ. Ηλεκτρολόγος Μηχανικός & Μηχανικός Η/Υ

ABSTRACT

In this thesis we examine and optimize execution traces of binaries. We focus on the potential for optimization of machine code by making assumptions about memory accesses and control flow. Based on our assumptions, we use symbolic execution to (a) find optimization opportunities in the trace scope and (b) perform these optimizations. We show that optimization opportunities exist in traces found in real programs, and we suggest how our assumptions can be adapted to the needs of a dynamic environment.

SUBJECT AREA: Dynamic Symbolic Execution

KEYWORDS: symbolic execution, x86 architecture, dynamorio, optimization, common subexpression elimination

ΠΕΡΙΛΗΨΗ

Σε αυτή τη διπλωματική εργασία εξετάζουμε ίχνη εκτέλεσης διεργασιών και εκτελούμε βελτιστοποιήσεις πάνω σε αυτά. Εστιάζουμε στην πιθανότητα βελτιστοποίησης του κώδικα έχοντας κάνει υποθέσεις για την πρόσβαση στη μνήμη και τον έλεγχο ροής. Με βάση τις υποθέσεις μας, χρησιμοποιούμε συμβολική εκτέλεση ώστε (α) να βρούμε ευκαιρίες βελτιστοποίησης στο εύρος του ίχνους και (β) να εκτελέσουμε τις βελτιστοποιήσεις αυτές. Δείχνουμε ότι ευκαιρίες βελτιστοποίησης υπάρχουν σε ίχνη πραγματικών προγραμμάτων και θεωρούμε τρόπους για το πώς οι υποθέσεις μας μπορούν να προσαρμοστούν στις προϋποθέσεις ενός δυναμικού περιβάλλοντος.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Δυναμική Συμβολική Εκτέλεση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: συμβολική εκτέλεση, αρχιτεκτονική x86, dynamorio, βελτιστοποίηση, αφαίρεση κοινών υποεκφράσεων

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis supervisor, Prof. Yannis Smaragdakis, for giving me the opportunity to work on this incredibly interesting and educational subject, for his availability, his invaluable advice, and for his guidance throughout the research and development of the thesis.

I would also like to thank my thesis supervisor, Dr. George Fourtounis, for his willingness to discuss issues pertaining to the thesis at any time, and whose insight and input helped shape this thesis.

October 2016

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The work for this thesis started in September 2014 in Athens and finished in October 2016. It was developed on an Intel Haswell architecture CPU and tested in various AMD64 machines running 64-bit operating systems. The outcome was a library for machine code manipulation, which is available for 64-bit versions of Linux, for the AMD64/Intel64 architecture.

1. INTRODUCTION

Virtual machines are programs that abstract the underlying operating system from client programs they execute. Language virtual machines are virtual machines that execute programs written in specific programming languages, or a language-specific byte representation of them [1]. Dynamically compiled languages such as Java [2] have arisen and matured in Computer Science. In the last two decades, programs written in these dynamically compiled languages are getting increasingly closer in performance to statically compiled alternatives. These languages provide unique features, such as dynamic loading of code during runtime and eliminating the need for memory management for the programmer. The user of these programs may, depending on the language implementation, enjoy cross-system compatibility since the target architecture for the virtual machines of these languages is independent of the language specification.

The reduction in the performance gap is achieved in part by dynamically compiling the, otherwise interpreted, code of a running program when a piece of code exceeds an execution-frequency threshold — the selection of the piece of code is dependent on implementation. One of the two methods of selecting code for compilation is the method/function scope: whenever an internal procedure with some inputs and some outputs surpasses an execution-frequency threshold, it is replaced by a compiled version of itself. The other method is by discovering frequently used execution traces: specific execution paths taken by the program through the control flow. The trace scope is unhindered by limitations presented by the control flow and function calls. The control flow constructs limit the range of some optimizations to the basic block — in layman's terms the code between control flow.

1.1 Goals

The goal of this thesis is to create a basis for a dynamic tracing reoptimizer for machine code. The reoptimizer will select an execution trace and discover optimization opportunities within it, while guarding the validity of the trace with several mechanisms. As the software will run dynamically, several sub-goals arise:

1. *Performance*: The software should have a linear complexity in regards to the amount of instructions. A viable optimizing solution will need to recognize traces consisting of large amounts of instructions.
2. *Modularity*: The software should be able to handle corner cases during development without introducing extra strain to the entire execution. A developer of the software should also be able to replace key parts of it without having to reinvent its core engine.
3. *Expandability*: As it aims to provide a basis for a completely dynamic tool, the soft-

ware must be designed in a way that allows for further development of the dynamic engine and any future goals.

4. *Simplicity*: The software should adhere to a design that can handle instructions while preferably not having to delve into the specifics of the target architecture, x86, which is a CISC architecture with more than 1000 instructions.

In the thesis we attempt and achieve to satisfy all the sub-goals.

1.2 libreopt.so

The software has been developed as a dynamic library for DynamoRIO [3], a runtime code manipulation system. DynamoRIO's input is a client program, which is executed on top of it, and a library, which contains executable code for instrumenting and monitoring client programs. DynamoRIO provides an API for instrumentation of the executing code, allowing for observation of the execution and injection of code in the executing code. This is accomplished by executing various hook functions whenever specific criteria are met. Such criteria can be the creation of a basic block (or better, what DynamoRIO considers a basic block), or an execution trace. The library takes advantage of the hooks to read and process instructions as they are executed. DynamoRIO will be further discussed in Chapter 2.

The library is contains two implementations, differing on trace selection:

- **General.** This implementation expects a user input of a starting instruction, and a span on the amount of instructions to process.
- **Guided.** The guided implementation recognizes marks in the execution of the process. These marks are be inserted in the source code of the client program, in the form of a pair of specific assembly instructions which define the boundaries of instruction processing. Thus, the main input of the library is a set amount of instructions, and its output is those instructions in a assembly-like form, annotated as *removed* or *modified*, along with a few added instructions necessary to preserve the proper execution of the trace.

The library employs a *flyweight object* module developed specifically for its operation – but generic enough to be usable in any other project. This module is a custom implementation of the flyweight object pattern [4]. It provides the feature of matching intricate immutable objects with the same structure and data in $O(logn)$ time complexity (in practice a constant cost). Objects of the same structure and data are manifested uniquely while the program is running. This reduces the equality checks between these flyweight objects to simple pointer equality checks. The module will also be further discussed in Chapter 2.

Whichever of the two implementations of the library the user decides to execute, the processing starts at some point during execution and stops at some other point during exe-

cution. The library processes one instruction in one *step*. By "step" we define the required restructuring of the library state to include that instruction as part of the library state. The processing of an instruction is independent of the actual execution. In fact, the granularity under which the library processes instructions lies at the level of the basic block.

The *state* of execution is defined as the set of **symbolic expression→symbolic expression** mappings, along with some required metadata. As far as the mappings are concerned, the key (left-hand side) defines a register or location in memory while the value (right-hand side) defines the symbolic value of that register or memory location. The value of the instruction pointer[1] is not recorded. The metadata bind instructions to symbolic expressions. We will discuss the Symbolic Execution engine in Chapter 3.

During each step, libreopt.so processes an instruction by manipulating a symbolic execution state. After a step is completed, the set of mappings describes the state of the machine from the point when the processing started to the point after the instruction. Libreopt.so follows the execution through any kind of control flow (i.e. backward or forward jumps, direct or indirect calls). It should be stressed that this thesis focuses on optimizing large traces of straight-line machine code, and not on providing a general solution to dynamic reoptimization.

After the processing is finished, the library iterates over the mappings. All instructions that are irrelevant to the final state are removed. All repeated calculations with the same data are merged. Their results are stored in special memory and loaded to the appropriate locations (registers or memory) instead of being recalculated.

Finally, the amount of the resulting modifications is printed in a separate file. The symbolic expressions may optionally be printed in *dot* notation and processed by visualization tools that understand this script, such as *graphviz* [5], resulting in an image with a visual representation of the symbolic expressions. The modifications may also be printed as annotations over a printed form of the instructions.

1.3 Contributions

This work makes the contribution of offering insight as to how traces of instructions of real programs can be optimized by symbolically executing them.

[1] The instruction pointer is the register that holds the position in memory of the next instruction to be executed.

2. TOOLS USED FOR DEVELOPMENT

This section describes the tools that we used in the development of our symbolic execution engine and optimization framework. These are the DynamoRIO framework for manipulating binaries (Section 2.1), a custom hash-consing mechanism for efficient representation of symbolic expressions (Section 2.2), and a visualization front-end (Section 2.3).

2.1 Code Manipulation Engine

We based our prototype on DynamoRIO [3], a state-of-the-art runtime code manipulator for binary code. DynamoRIO offers utilities such as a mature disassembler for the x86-64 architecture and an efficient instruction representation engine with modification and injection capabilities. DynamoRIO is the successor of the Dynamo [6] project developed at the Hewlett Packard Labs, was developed as part of Derek L. Bruening's PhD thesis at MIT, and is an active project at the time of writing of this thesis. Its main purpose is to enable instrumentation and monitoring of a process at runtime, while preserving performance close to the performance of a native execution of the binary.

DynamoRIO intercepts the entry and every exit (i.e. system call) of the instrumented application to maintain control of its execution. Every basic block of the client program requested for execution is passed through its code cache, which manipulates its branch instructions to valid cache or pre-existing basic block targets. The code cache keeps counters on all basic block targets of backwards jumps (referred to as *trace heads*). Whenever an execution amount threshold is surpassed on a trace head it traces the next path taken from that trace head to the first trace head encountered. Its trace algorithm is the Next Execution Tail algorithm [7], with the modification that backwards indirect branch targets are not considered trace heads; this reduces the amount of trace heads, resulting in a bigger mean size of traces. DynamoRIO provides runtime hooks for its client libraries which it calls at various points during its execution.

For the purposes of our project, we take advantage of the hook DynamoRIO provides when a new block is formed in its code cache. We copy the disassembled list of instructions in the block to a new space in memory. We instrument the basic block to call an entry point to the library, parameterized with the fresh copy of instructions. Thus, whenever this basic block is executed it calls the entry point function, and processes the backup list of instructions anew, based on the state at that point of the execution.

Our implementation also exploits the instruction manipulation interface of DynamoRIO. Such features as determining the instruction, determining an operand type and identity, and retrieving the flags usage of an instruction are crucial to the functionality of our library. They enable it to retrieve information about the instructions without having to execute per-

opcode behavior, other than some corner-cases.[2]

Programming Language. The available programming language spectrum was definitely narrow, as DynamoRIO provides its API for the C programming language only. Hence, the available options were C and C++. We chose C++ as its object-oriented features can express the symbolic execution nodes as classes, offering good balance between performance and memory use for our prototype. Various features of C++ were employed for the development, such as single and multiple inheritance, templated types, and object-oriented programming patterns (the Visitor, Factory Method and Curiously Recurring Template patterns).

2.2 FlyWeight Module

When performing symbolic execution and during each step, the project must check if the newly produced symbolic expression(s) are structurally equal to any other already existing symbolic expression. In the context of this project we define two objects as structurally equal if their respective primitive-type data are equal and their respective sub-objects are structurally equal. The objects may also define *shadow data*, including some sub-objects. Shadow data are allowed to differ between the two objects, i.e. they have no impact on determining structural equality.

We created a custom module based loosely on the flyweight pattern [4] to take advantage of these characteristics. There are two invariants that prompted the creation of the module:

1. The symbolic expressions themselves are not mutable, but can only be used as part of another symbolic expression after their creation.
2. Differing data for symbolic expressions of otherwise the same structural form need to be taken in account for all such symbolic expressions, i.e. the instructions for which the expressions were formed.

Suppose objects were checked for structural equality to any other existing object in the program *on creation*, and then only one was kept. In that case we could enforce that two objects are structurally equal if and only if the pointers that point to them have the same value, i.e. they are the same object.

The flyweight module has all objects created through a type-specific pseudo-factory using the factory pattern. The factory class maintains a hash table that maps all discrete objects. The production of the hashes is type-specific and must be implemented by classes that want to be supported by the flyweight functionality. Shadow data must not be included in

[2] An example of a corner-case is the move instruction (`mov`), which loads, stores, or copies ("moves") data between registers and memory. The program should recognize that this instruction is not an operation, since it simply copies data, without any other modifications or secondary destinations such as the rflags register.

the calculation of the hash; only data that take part in the structural equality check should.

Following an initial construction, where shadow data may be inserted, the object's hash is calculated and the hash table is checked for an equal-hash object. If one is found, structural equality is checked. At this point, if the outcome is positive, shadow data for the two objects are possibly merged.

The hash-cons feature of some functional language implementations is in essence a very similar technique [8, 9].

2.3 Visualization

The symbolic expressions have two out-of-memory forms. The first one is a simple textual representation aiming at the representation of simple expressions. The second one is the representation of the symbolic expressions DAG in dot notation, as input to the graphviz tool. The tool outputs graphical images as described in its input. This visual representation facilitates the comprehension of simple symbolic expressions.

3. METHODOLOGY

This Chapter provides an in-depth view on the mechanisms and decisions taken behind the implementation of the library, along with relevant tools and data. We present our symbolic execution engine for machine code (Section 3.1) and how it works with our memory model (Section 3.2). We then show how to combine our symbolic information with instrumentation to guide the dynamic analysis along certain execution paths (Section 3.3). We finish by demonstrating two optimizations possible in our framework: dead code elimination and common sub-expression elimination (Section 3.4).

3.1 Symbolic Execution

As was mentioned before, the input of the library is an off-line trace of instructions that a program executed. Therefore, these instructions include the control flow of the program, but following any branch instruction, the trace only includes the path taken by the execution. The hypothetical state of the machine, i.e. registers and memory, before any of the instructions in the off-line trace would execute will be referred to as the *initial state*. Accordingly, the phrase *final state* will be used to refer to the state of the machine after all the instructions in the off-line trace would have been executed.

The scope of the trace trascends function calls and control flow. Optimizations infeasible by the static compiler, e.g. due to segmentation of the optimization window by function calls, or instructions in a rare path of a branch disabling the compiler from optimizing the other path, are opportunities that the library should take advantage of.

The library forms relationships between the final state of the machine and the inputs of the initial state, aiming to discover recurring patterns. To achieve this, it symbolically executes the program [10] to form the final state of registers and memory as functions over the inputs. Since any instructions that don't contribute to the final state are not included in the symbolic expressions, dead code can automatically be eliminated. Thanks to the flyweight module described in Section 2.2, all different symbolic expressions appear in the memory of the library just once.

The implementation of the flyweight module in the library takes significant liberties as far as the similarity of the objects goes. It disregards the size of operands [3], and conceals instructions of the `mov` family along with other shadow data.

To better understand the concept of symbolically executing machine code, let's briefly examine the assembly code in Figure 1.

[3] Disregarding size of operands is a significant assumption. We assume that the full values of the destination registers/memory are fully determined by the instruction writing it, regardless of the destination size being possibly smaller. I.e. there are no previous instructions determining part of its value afterwards.

```
1 movq   %rax, %rbx
2 addq   %rcx, %rbx
3 addq   $256, %rcx
```

Figure 1: Sample x86 instructions in AT&T Assembly syntax.

The instructions in this figure would make the following changes:

1. Copy the value of the *RAX* register to the *RBX* register.
2. Add the value of the *RCX* and *RBX* registers, and then store the result in *RBX*.
3. Subtract 256 from the value of the *RCX* register, and then store the result in *RCX*.

To symbolically execute the instructions of Figure 1 would result in the symbolic expressions in Figure 2. Root nodes containing *RBX* and *RCX* symbolize registers and memory references that have been modified during execution. Root nodes containing *xF* symbolize bits of the flags register. These nodes will always have an edge to another symbolic expression, which will signify their value at the final state. Each one of those symbolic expressions has the instructions it executes as intermediate operation nodes. The direct descendants of a node represent the symbolic expressions they execute on. The leaf nodes can only be immediate values, or the initial values of registers, flags bits, or memory locations.

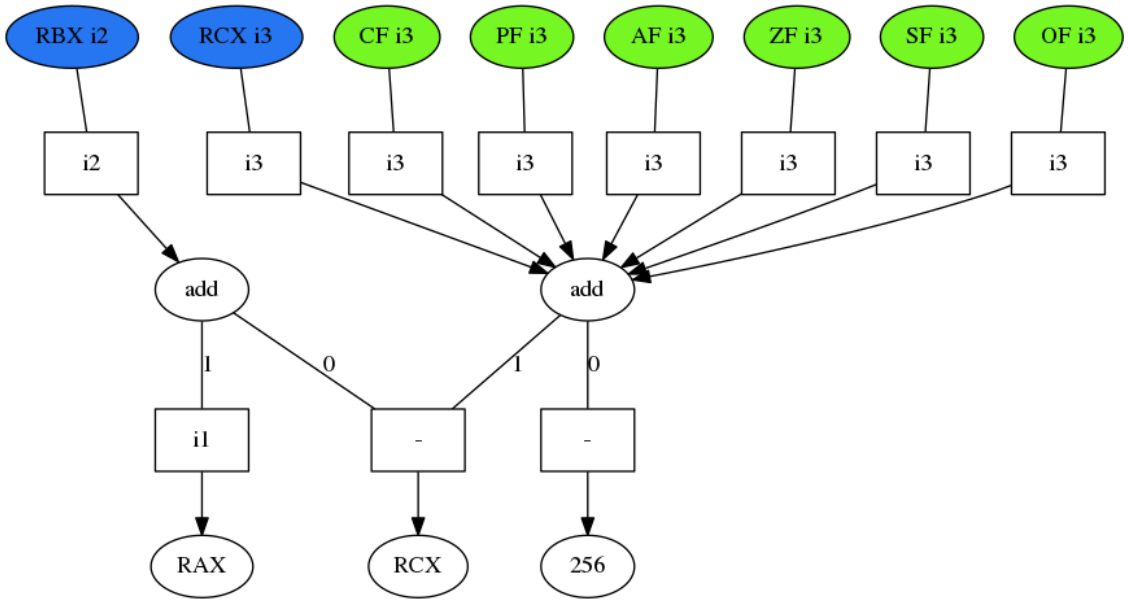

Figure 2: Symbolic expressions in visual form expressing the symbolic execution of the instructions in Figure 1.

The rectangle nodes between the symbolic execution nodes are part of the edges. They correspond to serial numbers of the off-line trace instructions, beginning with `1` for the first instruction in the trace. These nodes indicate that the link between two symbolic nodes is yielded after the corresponding instruction executes. We added them as part of the optimization engine — they do not take part in the symbolic execution.

One of the main features of the x86 architecture is the **rflags** register. This register holds a number of bits — these bits may have significance to some instructions depending on whether their preceding instructions set them or not. Table 1 lists the flags descriptions based on data provided by Intel in their Intel® 64 and IA-32 Architectures Software Developer's Manual [11].

CF	Carry flag	Set iff an arithmetic operation generates a carry or borrow out of the most-significant bit of the result.
PF	Parity flag	Set iff the least significant byte of the result contains an even amount of set bits.
AF	Adjust/Auxiliary flag	Set iff an arithmetic operation generates a carry or borrow out of bit 3 of the result; used in BCD arithmetic.
ZF	Zero flag	Set iff the result is zero.
SF	Sign flag	Set equal to the most-significant bit of the result.
DF	Direction flag	Controls the processing direction of string processing instructions (movs, cmps, scas, lods, stos).
OF	Overflow flag	Set iff the result cannot be expressed in the destination operand due to size limitations.
TF	Trap flag	Enables single-step mode for debugging when set.
IF	Interrupt enable flag	Controls the response of the processor to maskable interrupt requests.
NT	Nested task flag	If set, the current task is linked to the previously executed task.
RF	Resume flag	Controls the processor's response to debug exceptions.

Table 1: List of flags registers the library takes into account.

Different instructions set different flags of Table 1, clear others, while the value of some flags may also be undefined after an instruction executes. If an instruction does not leave a flag as is, the flag is set to be produced by the instruction. In Figure 2 an add operation produces the flags *CF*, *PF*, *AF*, *ZF*, *SF*, and *OF*. This add is executed as third and final instruction of Figure 1.

The Trap, Interrupt enable, Nested Task and Resume flags are system flags — they are not modified by application programs and therefore should keep their initial values in the scope of a trace.

```
1 movq   %rax, %rbx
2 movq   $0, %rdx
3 addq   %rbx, %rcx
4 adcq   $256, %rdx
```

Figure 3: Sample x86 instructions with flags manipulation.

A similar example in Figure 3 has one extra instruction: adc — add with carry. The adc instruction executes after the add instruction, reading the carry flag the add produced. The

result of the symbolic execution of the instructions is shown in 4.

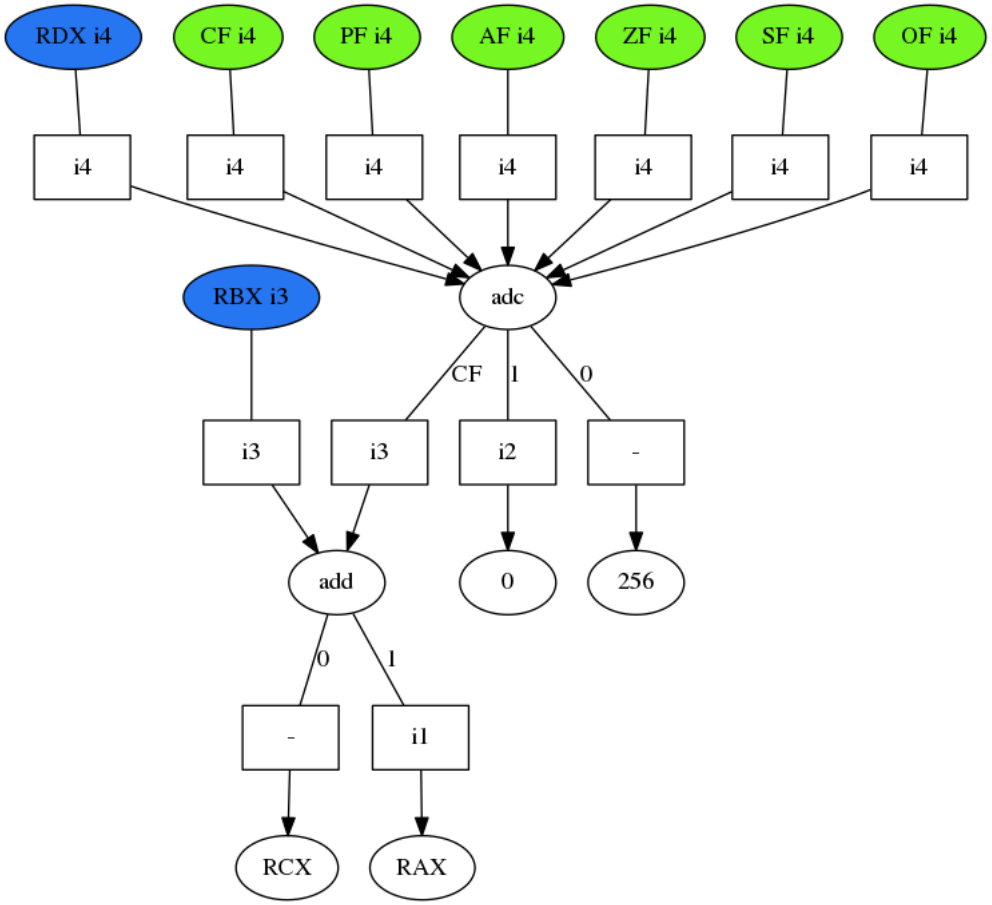

Figure 4: Symbolic expressions in visual form expressing the symbolic execution of the instructions in Figure 3.

The symbolic representation shows the read of CF by adc as an extra source operand to the adc operation; this source operaand is a destination of the add instruction that executed just before. Also take note that all the flags in this example are produced by the adc instruction. This happens because after add executes and sets them, adc executes, reads *CF* and sets the same flags. Hence their value is derived by adc.

3.2 Memory

Memory use is a fundamental feature of x86 machine code: programs frequently compute pointers to memory and use them to read and write memory contents, while most instructions can directly access memory. Thus, symbolic execution cannot find many interesting facts about a machine code fragment without having a specific model of memory.

This section describes two aspects of how our symbolic execution views memory: how much memory aliasing is permitted in our model (Section 3.2.1) and how to use dynamic memory allocation information (Section 3.2.2)

3.2.1 Memory Aliasing

Memory references in x86 are formed in three ways: (a) as absolute addresses, (b) as offsets relative to the program counter, or (c) as offsets calculated using the value of register sub-operands and immediates — the base register(B_{reg}), the index register(I_{reg}), the displacement(D) and the scale(S):

$B_{reg} + D + S * I_{reg}$

This presents a significant complication to the evaluation of the symbolic expressions. While PC-relative addresses are decoded and handled as absolute offsets — DynamoRIO provides them as such —, memory references with register sub-operands will correlate to symbolic expressions. This means that we are not able to provide a unique representation of memory addresses. We instead have to reference them using symbolic expressions. Referencing memory in this manner introduces memory aliasing.

Memory aliasing occurs when one memory location can have multiple representations — as opposed to an instruction set where each memory location may be referenced solely as a direct address. This situation is present in any pragmatic instruction set. Not enabling indirectly addressed memory means that a most basic language structure, the *pointer*, would not be implementable. Memory aliasing voids the correctness of the symbolic execution, since multiple symbolic expressions describing memory may evaluate to the same memory location. One approach to this would be the usage of a special path-branch node (called an “If-Then-Else” or “ITE” node). The most naive solution would have the ITE node inserted on all existing memory-referencing expressions each time a new memory reference is written to as follows:

Value at `existing` = *IF* `existing` aliases `new` *THEN* `symbolic expression for new` *ELSE* `symbolic expression for existing`

Something like this is not viable — the amount of paths with even a small amount of memory writes gets out of hand quickly. Furthermore, the x86 instruction set supports writing different sizes of data in memory. The check would have to be for overlaps of memory locations.

To give an example of how this can grow out of proportion quite rapidly, let’s suppose a point during symbolic execution of the instructions where the state has five discrete written memory locations. For this to have happened, the symbolic execution has produced ITE nodes on the final symbols to cover the possibility of one or more aliased memory locations, in order of execution: For each final symbol/node corresponding to a memory write, the creation of a new final memory write symbol must wrap it under an ITE node. The ITE node asserts evaluated equality between the symbolic expression for the new memory write node and the current symbolic expression calculating the address of that node. After the fifth memory write is symbolically executed the first node will have a depth of four ITE nodes, the second three, and so on. Each time a read with a different repre-

sentation happens after these symbolic expressions will have been formed, the symbolic expression to be chosen as a source will have to compare to all the symbolic expressions for the writes through ITE nodes.

To mitigate the aliasing problem, we introduced a single assumption the symbolic execution would adhere to: all memory references refer to unique addresses. The trace will then be optimized under this assumption. This means that in a supposed dynamic environment and before the trace executes, a series of guards will assert inequality between all memory addresses with more intelligent structures. If aliases are found then the trace does not execute, since its output will most probably be incorrect.

3.2.2 Custom Memory Management

When allocating memory from the heap using the *alloc family of functions, the C/C++ programmer will request sizes that either fit one object of a type of the program, or multiple objects of a specific type. These objects are then referenced by either their start address, or, if they are objects of composite types, then both their start address and offsets in them.

Ideally, an engine which attempts to assert inequality dynamically will have information on these types and then be able to discern aliasing based on where the variables in the program refer to. This is not something that is feasible at the machine code level. At this level, only addresses are available. However, if the instrumenting library could intercept the memory allocation call site and replace the target with a memory allocator of its own, it would gain information on discrete objects and arrays.

As part of a proof of concept for the realization of an on-line trace, we designed and implemented a memory allocator. This memory allocator segregates allocations based on the allocation size of requests. It uses internal structures which assign sizes to pages. It exposes the structures to functions which, by reading an address inside an object, can derive the start of the object and the size of the object. Since $O(logn)$ structures (hash tables) can be used to map the addresses to pages, and the pages to sizes, the functions can derive aliases and potential aliases very quickly. The memory allocator itself is not part of this thesis.

3.3 Instrumentation

The library instruments the client program at the basic block level. As described in Chapter 2, DynamoRIO provides the ability to instrument, i.e. modify the execution, when processing the basic block. Whenever a basic block is the target of a jump by the client program, DynamoRIO passes it through its code cache and performs callbacks the library may have inserted. The library gains access to the instruction list of the basic block, and the ability to modify it before it's released in the pool of processed executable code.

The library only modifies the executable code when it is active. This can be set to skip a

specific amount of instructions from the start of the program execution and then instrument for a number of instructions. It can alternatively be set to start when encountering a specific instruction expected by the library, and stop when encountering another such specific instruction. For this purpose we selected the multibyte version of the *nop* instruction. This version of the encoding has a different opcode. Rather than having no operands like its single-byte counterpart, the multi-byte *nop* accepts an address operand. The operand remains unused, yet it enables us to recognize marks in the source code — assuming no other nop instruction with the specific operand exists somewhere else in the program. For the purposes of our library we used the two instructions shown in Figure 5. When the first instruction is encountered, the instrumentation begins. When the second one is encountered, the instrumentation ceases.

```
1 nopl  $0x11111111
2 nopl  $0x22222222
```

Figure 5: *Nop* instructions used for enabling and disabling instrumentation.

These instructions must pre-exist: they must have been inserted in the source code of the program to be executed. The template for insertion of these instructions in a small C program is shown in Figure 6.

```
1  #include <stdio.h>
2
3  int main(void) {
4      //code outside the instrumentation scope of the library
5
6      asm volatile ("nopl $0x11111111\n\t");
7
8      //instrumentable code
9
10     asm volatile ("nopl $0x22222222\n\t");
11
12     //code outside the instrumentation scope of the library
13 }
```

Figure 6: Template for using instrumentation enabling and disabling instructions in source code.

At initialization, the library creates some basic internal structures. These structures are then set to be pointed to by DynamoRIO's thread local storage through its API. This enables the structures to be accessed at any point during execution, given any executing context — be it DynamoRIO control flow or program control flow.

As soon as instrumentation is enabled, the library instruments newly created basic blocks:

- At the beginning of each basic block, the instructions are modified so that the register state is saved in special memory provided by DynamoRIO; this memory is inaccessible by the client program.

- The original instruction list is cloned to another identical list.
- A function call is placed after the register state save. This function call jumps to library code. The parameters for this function call are the cloned list and the DynamoRIO *dr_context*, an opaque structure that can be used to extract the library state (i.e., its internal structures) and use them in the function.
- The instructions for the new basic block are passed on to the client program, which executes it whenever it is encountered, along with the instrumented code.

3.3.1 General Symbolic Execution Step Processing

The library function called by the client program retrieves the library environment through the DynamoRIO API and calls the symbolic execution function over each of the instructions. The symbolic execution function uses the DynamoRIO API to discern the amount and type of source and destination operands, reading and writing of flags and opcode for the instruction. It then produces a symbolic execution node, which is linked to its destination operands (registers, memory locations, and flags bits). If an instruction has zero destination operands it is not linked and is therefore discarded. Exceptions to the rule are described in Section 3.3.2.

Textual Representation for the Symbolic Execution. The text form for Symbolic Execution is a series of statements. The statements are of the form $X := Y$, where X is either a register or a Symbolic Expression for a memory address, and Y is the Symbolic Expression for the value of said register or the memory with address evaluated by Symbolic Expression X.

The syntax of Symbolic Expressions *SE* and Memory Addresses *ME* follows:

SE	::=	$OPER(SE_1, \ldots, SE_n), n \geq 1$	operator application
	|	$\$C$	constant
	|	*REG*	register
	|	*FL*	flag
	|	*ME*	memory address
	|	$\texttt{MEM}ME$	dereference
ME	::=	$[C], C \geq 0$	absolute address
	|	$[SE_b + C + SE_i * D], D \in \{1, 2, 4, 8\}$	general address

where *C*, *REG*, and *FL* range over integer constants, register names, and flags bit names respectively. *OPER* ranges over the operation names — e.g. add, adc, lea, sub. General memory addresses have the form $(base + constant\ displacement + index * scale)$. (cf. Section 3.2.1). The other form, absolute addressing, is listed separately. Alternatively, absolute addresses could also be represented by the general address form, when $SE_b = SE_i = \$0$.

In Figure 7 we see the textual representation for the result of the symbolic execution in Figure 2. Expressions involving the flags have not been printed for the sake of clarity. The meaning of the first statement in Figure 7 would be: “The value of the register *RBX* in the final state is the result of performing the add operation to the initial values of *RAX* and *RCX*.” The meaning of the second statement would be: “The value of the register *RCX* in the final state is the result of performing the add operation on the constant 256 and the initial value of *RCX*.”

```
1 RBX := add(RAX, RCX)
2 RCX := add($256, RCX)
```

Figure 7: Textual representation for the result in Figure 2.

Consider a modification to the previous example, with the results in Figure 8.

```
1 RBX := add(RAX, RCX)
2 [RBX] := add($256, [RBX])
```

Figure 8: Memory address example for the textual representation of Symbolic Expressions.

In the example of Figure 8 a symbolic expression replaces the *RCX* register. The second statement now reads: “The value in the final state of the memory location pointed to by **the initial value** of *RBX* is the result of performing the add operation on the constant 256 and the initial value of the memory location pointed to by the initial value of *RBX*.”

Symbolic Expressions for memory locations are wrapped in brackets (`[ ]`). For example, `[add(1, 2)]` is the symbolic value of the memory at the location `add(1, 2)`, i.e. 3. In statements, final memory location symbolic expressions (*X*) are always bracketed.

3.3.2 Exceptions to the General Symbolic Execution Step Processing

Six total instructions require unique handling. The first four instructions are: *push*, *pop*, *add*, and *sub*. The first two are heavily evident in results from symbolic executions as references to memory; by modifying the stack pointer they create constant (immediate) offsets from a base memory location. We merge codependent instructions of integer additions and subtractions of constant values in one, assuming arbitrary sizes of operands, practically ignoring the overflowing of values.

The *add* and *sub* instructions are otherwise handled as the general case for step processing.

The *push* instruction of the x86 architecture performs two operations and yields values in two destinations; it first increases the size of the stack by decrementing the stack pointer according to the size of its single source operand. It then stores the value of the source operand at the new top of the stack: the address of the new stack pointer.

The *pop* instruction of the x86 architecture also performs two operations. It loads the value on the top of the stack to its single destination operand and then increments the stack pointer by the size of the operand.

The final two instructions that require special handling are *call* and *ret*. These instructions set the *RIP* register, save or load the *RIP* register to or from the stack, but more importantly change the value of the stack pointer to account for the size of the *RSP* register pushed or popped. While *RIP* values are discarded as part of the control flow, the change of the *RSP* register must be represented by symbolic expressions.

```
1  #include <stdio.h>
2
3  int fn(int a) {
4      return 0;
5  }
6
7  int main(void) {
8
9      asm volatile ("nopl 0x11111111\n\t");
10     int x = 6;
11     fn(4);
12     x = 3;
13     asm volatile ("nopl 0x22222222\n\t");
14
15     return 0;
16 }
```

Figure 9: C program from which the instructions in Figure 10 derive.

To understand the need for the transformation of the operations of these instructions, let's briefly consider Figure 10. This is executed code, disassembled by DynamoRIO in some notation closely matching the AT&T assembly. Figure 9 presents the corresponding C code, i.e. the code between the multi-byte *nop* instructions.

```
1  movl   $0x00000006, 0xfffffffc(%rbp)   # set x = 6
2  mov    $0x00000004, %edi               # arg1 for function-call to fn
3  call   $0x0000000000400756             # perform the function-call
4  push   %rbp                            # create the stack frame
5  mov    %rsp, %rbp
6  mov    %edi, 0xfffffffc(%rbp)
7  mov    $0x00000000, %eax               # set the return value
8  pop    %rbp                            # destroy the stack frame
9  ret                                    # return to caller
10 movl   $0x00000003, 0xfffffffc(%rbp)   # set x = 3
```

Figure 10: DynamoRIO output assembly for symbolic execution between function calls.

The **x** variable corresponds to `-4` from the base pointer (`0xfffffffc` is the two's complement representation of `-4` for a four-byte operand). It is written to twice: once before the function call, and once after the function call.

```
1 RAX := 0
2 RDI := 4
3 RSP := RSP
4 RBP := RBP
5 [RBP + $-4] := 3
6 [RSP + $-8] := call([40000])           # RIP value at call site
7 [RSP + $-16] := RBP
8 [RSP + $-20] := 4
```

Figure 11: Result of symbolically executing the instructions of Figure 10: named registers are followed by memory locations (in brackets).

Any omitted symbols in Figure 11 remain unchanged between the initial and the final state.

On the other hand, the symbolic execution algorithm without special handling of the instructions, calculates the result presented in Figure 12 (captured by the debug output mode of our library).

```
1 RAX := $0
2 RDI := $4
3 [RBP + $-4] := $6
4 [RSP + $-8] := call([400756], RSP)
5 RBP := pop(push(RBP, call([400756], RSP)), [push(RBP, call([400756], RSP))])
6 RSP := ret(pop(push(RBP, call([400756], RSP)), [push(RBP, call([400756], RSP))
     ]), [pop(push(RBP, call([400756], RSP)), [push(RBP, call([400756], RSP))])
     ])
7 [push(RBP, call([400756], RSP)) + $-4] := $4
8 [call([400756], RSP) + $-8] := push(RBP, call([400756], RSP))
9 [pop(push(RBP, call([400756], RSP)), [push(RBP, call([400756], RSP))]) + $-4]
     := $3
```

Figure 12: Debug output of library symbolically executing the instructions of Figure 10. The library is set to omit handling of the push and pop instructions, instead treating them as black boxes.

Consider these two observations:

1. The symbolic value of symbolic expressions containing pop, push, or call is not apparent intuitively. The destination operands are saved in the memory representation of the symbolic expressions, therefore the information to evaluate those is correct, even though it is not being printed in this debug output.
2. The size of the operands is ignored in the symbolic operation. Since it is saved in memory, it is assumed that the operation is defined by the size of its operands and acts accordingly to set the value of the destination. In practice, it is irrelevant.[4]

Memory guards. The debug result of Figure 12 itself is not correct. One specific example would be

```
[pop(push(RBP, call([400756], RSP)),
```

[4] This is expanded upon in Section 3.4.

```
[push(RBP, call([400756], RSP))]) + $-4]
```

and

```
[RBP + $-4].
```

Both describe the same offset from the *RBP* register's initial value, yet the algorithm cannot know that.

Thus immediate values must be merged using some form of partially evaluated symbolic expressions. Inevitably, some final symbols with different representations will end up having the same concrete value under some cases. In those specific cases the benefit of being able to assert inequality between used memory addresses comes into play, since it functions as a set of guards against aliasing to the optimized trace. This set of guards protects the validity of the trace by verifying on-line the assumption that all symbolic expressions write to different final memory locations — as well as the possibility that aliased intermediate memory locations are not falsely considered different.

This is where having a custom memory allocator comes into play, as mentioned in Section 3.2.2. The memory allocator can assert these cases and answer whether the to-be-executed trace is valid or not.

3.4 Optimization

Along with selecting a trace of code and symbolically executing it, we perform the following book-keeping:

1. We save the instructions constituting this initial trace.
2. We number the instructions uniquely in increasing order (order of appearance in the trace).
3. We assign the instruction numbers to symbolic execution nodes. The reasoning for this is that a symbolic execution node is assigned number **X** if the expression this node forms with its subtrees is formed after instruction numbered **X** is executed.

The data is stored in such form so that the instruction number itself is linked to the instruction numbers of its source operands, in the same manner as the symbolic execution nodes are.

After having picked a trace and symbolically executed it, the library performs two optimizations over the symbolic expressions: dead code elimination (Section 3.4.1) and common sub-expression elimination (Section 3.4.2).

3.4.1 Dead Code Elimination

Our algorithm traverses the roots of symbolic expressions that are directly linked to final symbols; it traverses both symbolic expressions used to calculate addresses for final

symbols that are memory locations, along with symbolic expressions which correspond to their value at the final state. When encountering a symbolic expression with more than one corresponding instructions it links all numbers to the number of the first instruction.

Having done that, the algorithm traverses the final symbols in the same manner a second time. At this point the algorithm marks the instructions that correspond to expressions reachable from final symbols; this automatically performs dead code elimination.

3.4.2 Common Sub-Expression Elimination

When marking alive nodes, our algorithm also checks if more than one instructions correspond to a specific node. In that case, the algorithm marks the first corresponding instruction as *saved*, and the other instructions with a link to the first one. A “saved” instruction is an instruction whose destination operands may be needed to be stored in special memory (inaccessible by the original program). The operands are loaded replacing recurring calculations. If the rest of the usages are unneeded to form the final state, the instruction results will not be stored.

To finalize common sub-expression elimination the algorithm traverses the **instructions** and replaces re-evaluated subexpressions with loads to registers from a specific offset from the special memory. The special memory is written after the first calculation occurrence in the execution trace, using the common sub-expression as a guide. This happens under the condition that the number of instructions removed after performing the modification is above a given threshold [5].

We should note here that the amount of instructions removed can be fewer than the amount of instructions needed for the calculation of a symbolic expression. This is the case when a subset of those instructions may be required for calculating the value of another final symbol.

An example of this can be seen in Section 4, where in Figure 16 and Figure 17 the instructions involving the value of the *RAX* register are not removed, since they determine the value of *RAX* in the final state.

[5] At the time of writing the threshold is a numeric constant.

4. EXPERIMENTAL RESULTS

This chapter shows how our prototype finds code that can be optimized away in a set of actual programs.

The experimental results are split in two categories: The first set contains the results of running the library in a controlled environment (Section 4.1). This set, the test cases set, presents the reduction in code the library can perform over a trace of execution. The second set contains iterations on loops of real-world programs, for increasing iterations of the loops (Section 4.2).

4.1 Test Cases

The test cases presented are part of the test suite used to verify the validity of the symbolic expressions and the produced instruction modifications for the library.

We consider the instructions produced as the results of the test cases; these instructions are modifications over the source trace. The modifications are per instruction and can be one of the modifications described in Table 2. A pedantic reader will realise that no information about the source trace is lost and can be recreated for a juxtaposition with the final trace.

none	Source instruction executed as is. It remains in the final trace.
−−	Source instruction removed. This instruction belonged in the source trace, but is not needed for valid execution of the final trace.
++	Final trace pseudo-instruction. This instruction can either be a load or a save to a safe pseudo-memory operand, used exclusively by the library for spilling and restoring data to the state of the machine.
+−	Source instruction replaced by pseudo-instruction. As above, but this pseudo-instruction has replaced an instruction of the source trace. The removed instruction is listed after in a comment (signified by #).

Table 2: List of possible instruction modifications and how they appear in the library output.

```
1  -- mov    $0x00000000, %rax
2  -- mov    $0x00000001, %rbx
3  -- mov    $0x00000002, %rcx
4  -- mov    $0x00000003, %rdx
5  -- add    $0x04, %rax
6  -- add    $0x08, %rbx
7  -- add    $0x0c, %rcx
8  -- add    $0x10, %rcx
9  -- add    %rax, %rcx
10 -- add    %rbx, %rcx
11 -- add    %rcx, %rdx
12 -- add    %rcx, %rax
13    mov    $0x00000000, %rax
14    mov    $0x00000001, %rbx
15    mov    $0x00000002, %rcx
16    mov    $0x00000003, %rdx
17    add    $0x04, %rax
18    add    $0x08, %rbx
19    add    $0x0c, %rcx
20    add    $0x10, %rcx
21    add    %rax, %rcx
22    add    %rbx, %rcx
23    add    %rcx, %rdx
24    add    %rcx, %rax
```

Figure 13: Execution results of the `ts_doubled` test-case program.

Figure 13 presents a very simple case, where repeating code occurs. The first half of the instructions are found to not contribute to the final state by the library and are removed.

```
1     push   %rax
2  -- mov    %rbx, %rax
3     pop    %rax
```

Figure 14: Execution results of the `ts_pushpop` test-case program.

The very simple test case in Figure 14 shows an example of the special handling of the push and pop instructions. The `mov` instruction is removed because the value of rax is overwritten by the next instruction. More importantly, the push and pop instructions are not removed since they form a symbolic expression for the memory location where the stack pointer points. This symbolic expression contributes to the final state.

```
1     add    $0x04, %rdi
2  -- mov    (%rdi), %ecx
3  -- mov    0x00000308(%rax), %r8
4  -- mov    %ecx, %edx
5  -- lea    (%r8,%rdx,4), %r8
6  -- lea    (%rdx,%rdx,2), %rdx
7  -- lea    (%rbx,%rdx,8), %rdx
8  -- movzx  0x06(%rdx), %r10d
9  -- movzx  0x04(%rdx), %r9d
10 -- and    $0x0f, %r9d
11 -- mov    0x08(%rdx), %r11
12 -- mov    %r11, %r9
13 -- add    (%rax), %r9
14 -- add    $0x04, %r8
15 -- add    $0x01, %ecx
16    add    $0x04, %rdi
17    mov    (%rdi), %ecx
18    mov    0x00000308(%rax), %r8
19    mov    %ecx, %edx
20    lea    (%r8,%rdx,4), %r8
21    lea    (%rdx,%rdx,2), %rdx
22    lea    (%rbx,%rdx,8), %rdx
23    movzx  0x06(%rdx), %r10d
24 -- movzx  0x04(%rdx), %r9d
25 -- and    $0x0f, %r9d
26    mov    0x08(%rdx), %r11
27    mov    %r11, %r9
28    add    (%rax), %r9
29    add    $0x04, %r8
30    add    $0x01, %ecx
31    mov    %ecx, %edx
```

Figure 15: Execution results of the `ts_leatest` test-case program.

Figure 15 is again a case for dead code elimination, yet slightly more complex. The `lea` instruction (Load Effective Address) stores the value calculated as an address by the first operand to the second operand, without accessing the memory at that location. The other instruction used, `movzx` moves the first, smaller in size, operand to the second operand by zero-extending the value to cover the second operand's size.

In this case we can observe that where the values of registers are needed as sub-operands the library preserves the dependency and does not remove contributing instructions.

```
 1    mov    $0x00000000, %r11
 2    mov    $0x00000001, %r12
 3    mov    $0x00000002, %r13
 4    mov    $0x00000003, %r14
 5    add    $0x04, %r11
 6    add    $0x08, %r12
 7    add    $0x0c, %r13
 8    add    $0x10, %r13
 9    add    %r11, %r13
10    add    %r12, %r13
11 ++ mov    %r13, $((temp10 + 0))
12    add    %r13, %r14
13    add    %r13, %r11
14    mov    $0x00000000, %rdx
15    mov    $0x00000001, %rax
16 -- mov    $0x00000002, %rbx
17    mov    $0x00000003, %rcx
18    add    $0x04, %rdx
19    add    $0x08, %rax
20 -- add    $0x0c, %rbx
21 -- add    $0x10, %rbx
22 -- add    %rdx, %rbx
23 +- mov    $((temp10 + 0)), %rbx          # --  add    %rax, %rbx
24    add    %rbx, %rcx
25    add    %rbx, %rdx
```

Figure 16: Execution results of the `ts_sameexpr_ad` test-case program.

Figure 16 shows the removal of common sub-expressions. The instruction threshold for common subexpressions for this execution was set to 3: By loading the value of a calculation the algorithm may remove a certain amount of instructions. The threshold makes sure that the algorithm will not perform common sub-expression elimination for a specific expression if less than 3 instructions can be removed.

In the case of Figure 16, the final value of the *RBX* register is a recalculation of the value of the *R13* register after instruction `10`. Hence, the library inserts a store to the pseudo-memory location and uses this value at instruction `23` to replace the final instruction for the calculation of the value for *RBX*. Notice that the previous 3 instructions, now irrelevant to the final state, are also removed. No other recalculations are handled in this manner, since the instructions removed would be fewer than the threshold.

```
 1    mov    $0x00000000, %r11
 2    mov    $0x00000001, %r12
 3    mov    $0x00000002, %r13
 4 -- mov    $0x00000003, %rdx
 5    add    $0x04, %r11
 6    add    $0x08, %r12
 7    add    $0x0c, %r13
 8    add    $0x10, %r13
 9    add    %r11, %r13
10    add    %r12, %r13
11 ++ mov    %r13, $((temp10 + 0))
12 -- add    %r13, %rdx
13    add    %r13, %r11
14    mov    $0x00000000, %rdx
15    mov    $0x00000001, %rax
16 -- mov    $0x00000002, %rbx
17    mov    $0x00000003, %rcx
18    add    $0x04, %rdx
19    add    $0x08, %rax
20 -- add    $0x0c, %rbx
21 -- add    $0x10, %rbx
22 -- add    %rdx, %rbx
23 +- mov    $((temp10 + 0)), %rbx          # --  add    %rax, %rbx
24    add    %rbx, %rcx
25    add    %rbx, %rdx
```

Figure 17: Execution results of the `ts_sameexpr_dr` test-case program.

Figure 17 is a copy of Figure 16 where the *R14* register is replaced by the *RDX* register. A case for dead code elimination in the context of common sub-expression elimination.

```
 1    sub    $0x08, %rsp
 2    mov    %rdx, (%rsp)
 3    mov    (%rsp), %rax
 4    add    $0x08, %rax
 5    add    $0x09, %rax
 6 ++ mov    %rax, $((temp5 + 0))
 7 -- mov    %rdx, %rbx
 8 -- add    $0x08, %rbx
 9 +- mov    $((temp5 + 0)), %rbx        # -- add    $0x09, %rbx
10    add    $0x08, %rbp
```

Figure 18: Execution results of the `ts_simplemem` test-case program.

Concluding with the test cases, we present in Figure 18 the possible need for linear instruction passes on the code. While the library recognizes and handles the removal of repeated calculations, it is not able to recognize that all instructions between the store and the load have been removed. A better solution would be replacing both instructions numbered `6` and `9` with `mov %rax, %rbx`.

4.2 Real-world Benchmarks

Running the library over arbitrary programs at arbitrary points of execution is supported but would not give productive results. Since the library simply parses the execution path and considers it as a trace of code, rather than optimizing actual hot code, the results would vary depending on the point at which the library would have started the parsing. This point could change depending on environment factors, such as the libraries loaded by the program or the amount of instructions executed before the `main()` function is called. Rather than presenting results skewed by such a volatile environment we present results for the execution of programs with large controlled loops, where the source code has been marked for the library as described in Section 3.3. These programs are a BASIC interpreter (Section 4.2.1), the Lua language interpreter (Section 4.2.2), and a custom mini interpreter for a very basic language created by us for this purpose (Section 4.2.3).

To facilitate the understanding of the results since the output instructions would be much greater in size than the test-cases, we show the amount of symbolic nodes created per trace.

The benchmarks are visualized on 2D-plane graphs, where the X-axis signifies the number of times the program passed through the checkpoint marker before stopping. Five sets of data are visualized. These sets of data, as shown on the legends of the graphs, are:

1. *Instrs*: The amount of instructions executed in total.
2. *OperExpr*: The amount of Symbolic Expression Nodes that signify operations produced.
3. *AccMov*: The amount of reachable `mov`-family instructions (loads and stores) where these instructions are links between the instructions forming Symbolic Nodes and are hence required by the Symbolic Expressions.
4. *OperExprBr*: The amount of Symbolic Expression Nodes that signify operations produced, considering branch targets as actual memory destinations.
5. *AccMovBr*: The amount of reachable `mov`-family instructions required by the Symbolic Expressions, when considering branch targets as actual memory destinations.

Reiterating, each point on the X-axis signifies an execution of x iterations, where x is the offset from zero on the X-axis. The word iterations in this context means the number of times the process executed the checkpoint instruction. The points on the Y-axis signify the amount of instructions executed in total, the amount of Symbolic Expressions, `mov`-family instructions, Symbolic Expressions including control flow destinations and `mov`-family instructions reached when including control flow destinations, for that specific execution with x instructions.

4.2.1 BASIC Interpreter

For the BASIC interpreter, we used an implementation in C by Adam Dunkels, uBASIC [12]. We modified this version by adding an initialization and checkpoint marker at the function parsing and executing each line.

We used two BASIC programs as benchmarks, a program computing the factorial of 20 (Figure 19) and a simple program that loops forever involving just a constant value (Figure 21).

```
10 x = 1
20 for i = 2 to 20
30       y = x
40       for j = 2 to i
50           x = x + y
60       next j
70 next i
80 print x
90 end
```

Figure 19: Factorial source code used in benchmarks for the BASIC interpreter.

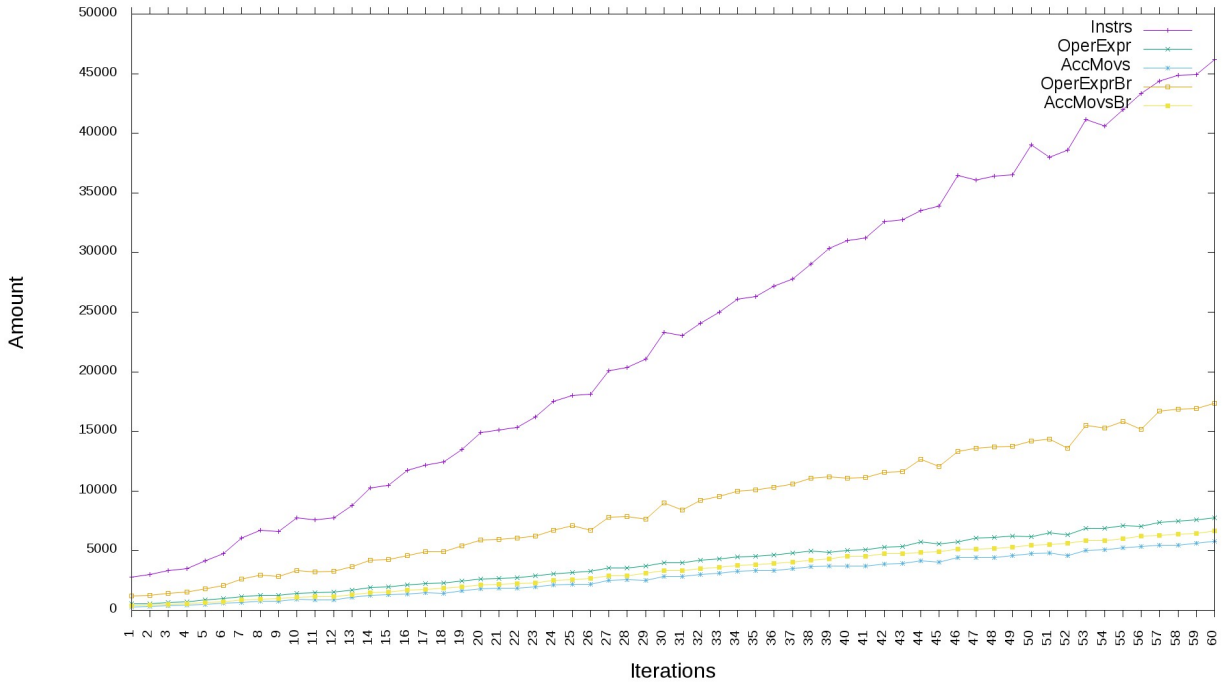

Figure 20: uBASIC interpreter results for the factorial program.

Starting with the factorial program, we can see the measurements from its execution in Figure 20. We can observe the following:

- The `mov`-family instructions attached to the operations amount to a significant percentage of the total operations. This is consistent with the findings of Huang and Peng [13] for DOS and Windows 95 applications.

- When branch targets are considered memory state, the operations required to describe the final state are more than doubled.
- Interestingly, the slope between lower amounts of iterations to higher amounts of iterations is negative between some. After carefully examining the execution traces between executions for the same amount of iterations, we found that the control flow differs. As the instructions executed are often not present in the binaries of the programs tested, they must appear in the execution trace as the result of library calls, and the non-determinism of the execution stems from code during the execution of library functions.

Yet, a result of operations increasing linearly is not something we would expect. What we would expect is a decline in the deltas between points as the amount of iterations increased. This would be consistant with the library recognizing repeated interpreter code.

```
1 10 x = 1
2 20 goto 10
```

Figure 21: Simple program used in the benchmarks for the BASIC interpreter.

The unexpected results from the previous program can be further investigated by observing the symbolic execution of the simpler program of Figure 21. This program, although tiny and only using constant values and a single loop without a condition, also demonstrates too many symbolic values, as seen in Figure 22.

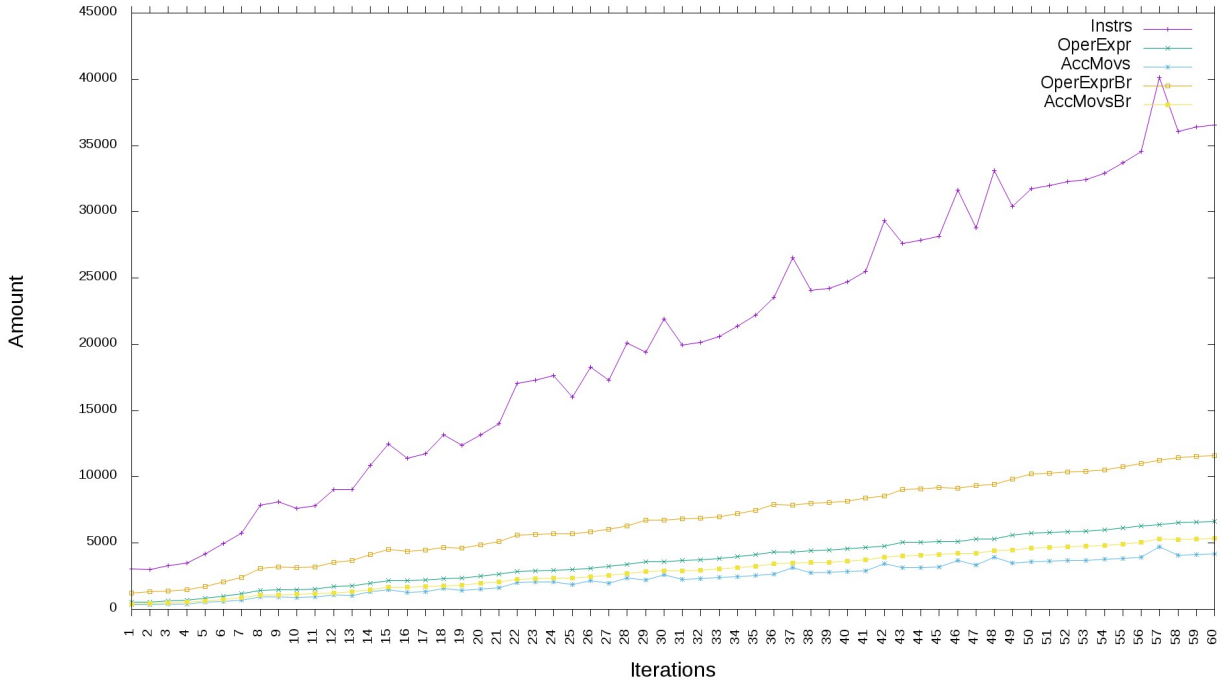

Figure 22: uBASIC interpreter results for the simple program.

Compared to the previous BASIC example, we observe minor differences in the measured

data; however, we would expect a constant number of symbolic values.

In the next section, we will investigate similar behavior of the Lua interpreter, to see how the same issues arise in a more mature interpreter.

4.2.2 Lua Interpreter

Before moving on to our own interpreter, we will compare the BASIC results with results from using the official Lua interpreter [14]. We instrumented the Lua interpreter by placing the instrumentation initialization and checkpoint marker at the beginning of the main interpreter loop. This loop processes instructions, hence the instrumentation is parallel to the one for the BASIC interpreter.

Since we used the factorial algorithm to present results for BASIC, we are using the same algorithm for the Lua interpreter. The Lua version is shown in Figure 23.

```
1 function factorial(n)
2     local x = 1
3     for i = 2, n do
4         x = x * i
5     end
6     return x
7 end
8
9 print (factorial(30))
```

Figure 23: Factorial source code used in benchmarks for the Lua interpreter.

The results, presented in Figure 24, are very similar to the uBASIC results.

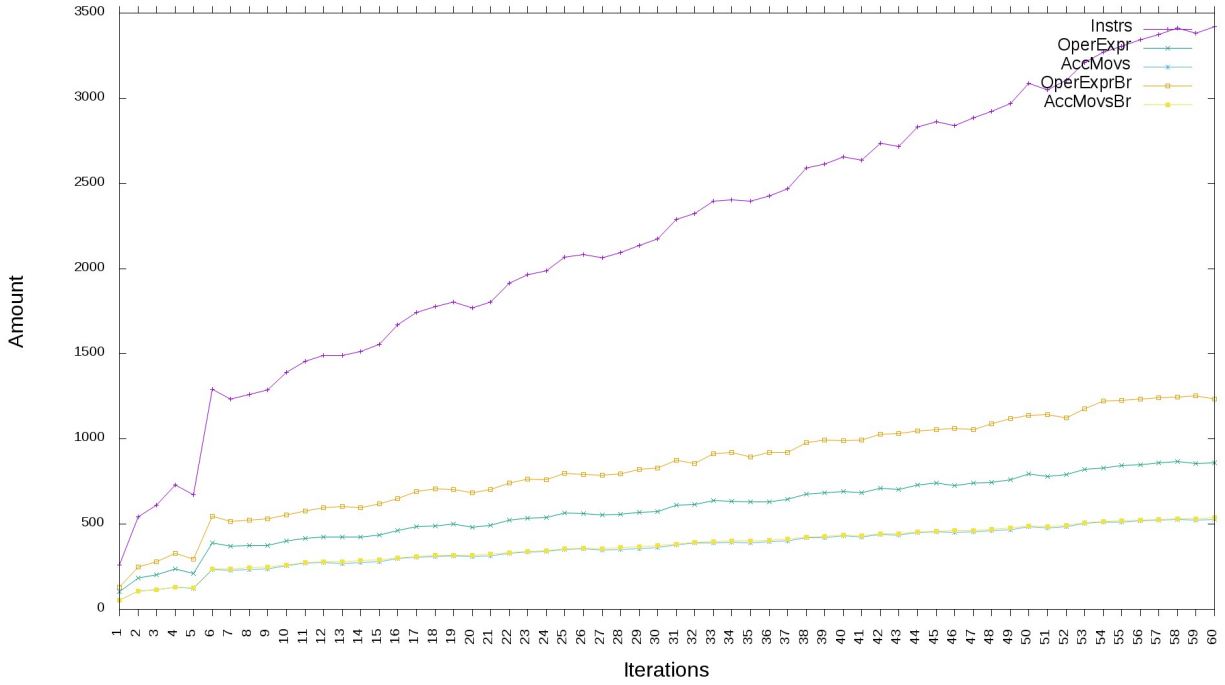

Figure 24: Lua interpreter results for the factorial program.

The results of our Lua experiments above (and the BASIC ones of Section 4.2.1) prompted us to create our own interpreter in the next section to measure the generation of symbolic values in a more controlled environment.

4.2.3 Custom Mini-Language Interpreter

Our own interpreter implements a BASIC-like language that supports (a) integer variables, (b) the assignment operator (which can perform addition, subtraction, multiplication, and division), and (c) the instructions PRINT (prints to output), EXIT (halts interpretation), GOTO (moves the program counter to a specific line number), and BZERO (moves the program counter to a specific line number when the variable argument has a value of zero).

To control the cost of variable access, only single-letter variables are permitted. Their values are stored in a 26-element array, ensuring a constant read/write cost. Each interpreter loop interprets exactly one line of source code.

We tested a very simple program to verify the problem persists on our interpreter and to explore the solution (Figure 25). Our program sets x to 1, performs the addition of x and 1, stores the result in y, and then sets the program counter of the interpreter to the first line, therefore repeating indefinitely.

```
1 x = 1
2 y = x + 1
3 GOTO 1
```

Figure 25: Source code used in benchmarks for the Mini-Language interpreter.

The results from the execution of our test program are shown in Figure 26.

As a side-effect, we observe that since our interpreter does not use external libraries, the amount of instructions per iteration is much more obvious. We can also deduce that when the difference between two consecutive points is large, one of the more complex instructions is executed in that iteration, while when it is small, the GOTO instruction is executed.

More importantly, we observe that the problem persists. Since our interpreter is very simple, and the program it executes even simpler, the problem definitely lies in memory addressing: *the library keeps producing different symbolic expressions for the same data*.

The problem is not difficult to understand. Each time the `GOTO` instruction is executed, the PC is set to the numeral conversion of the characters of the string directly after `GOTO`. We, however, know that all source code is referenced as an offset from the program counter. That means, that the symbolic value about to be assigned to the Program Counter will contain a subexpression with the previous symbolic value of the Program Counter. Each time the constant `1` of line `3` in Figure 25 is read, the Program Counter Expression will reset

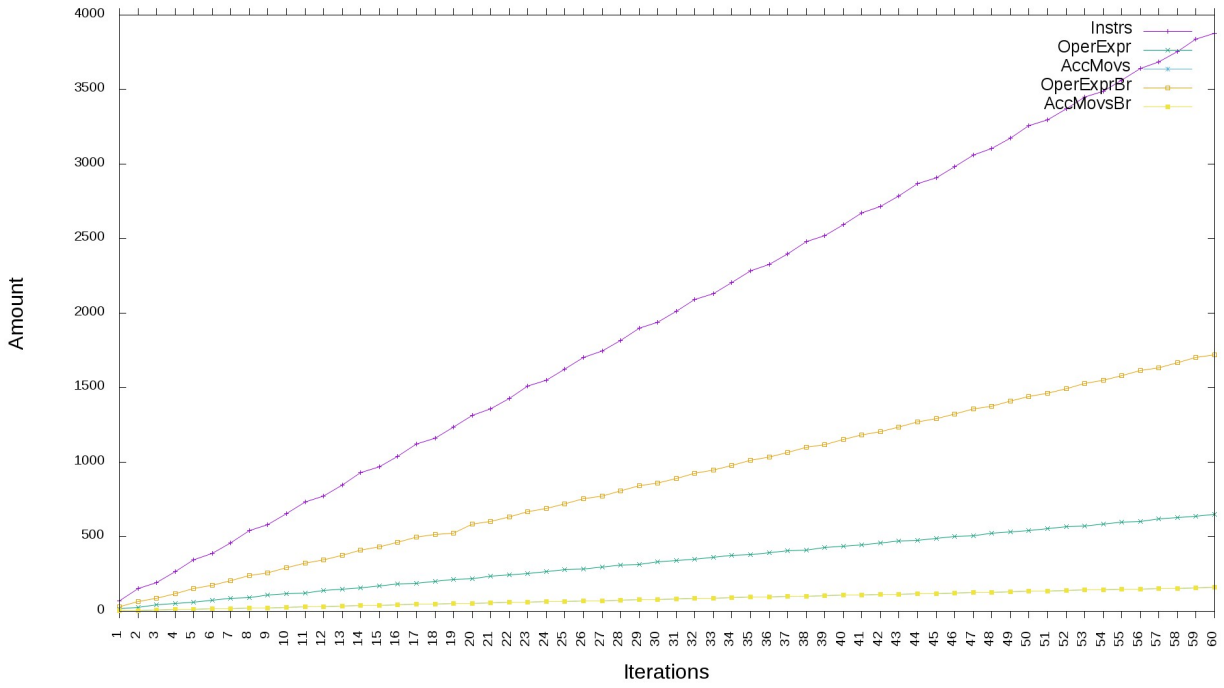

Figure 26: Mini-Language interpreter results for the simple loop program.

to a value that includes the previous symbolic value, thus never recognising a pattern. The problem thus is that `GOTO` introduces a level of indirection that makes the generated symbolic expressions for the PC to constantly grow instead of taking a finite number of different symbolic values.

We verified this assumption by creating a new command for our Mini-Language which does not have this extra level of indirection but has direct semantics: `RESET`. This command simply sets the value of the PC to 1. We modified the program of Figure 25 by replacing `GOTO 1` with `RESET`.

The results, presented in Figure 27, show a constant value of operations, with and without including control flow, since operations that decide the control flow are also referencing offsets from the PC.

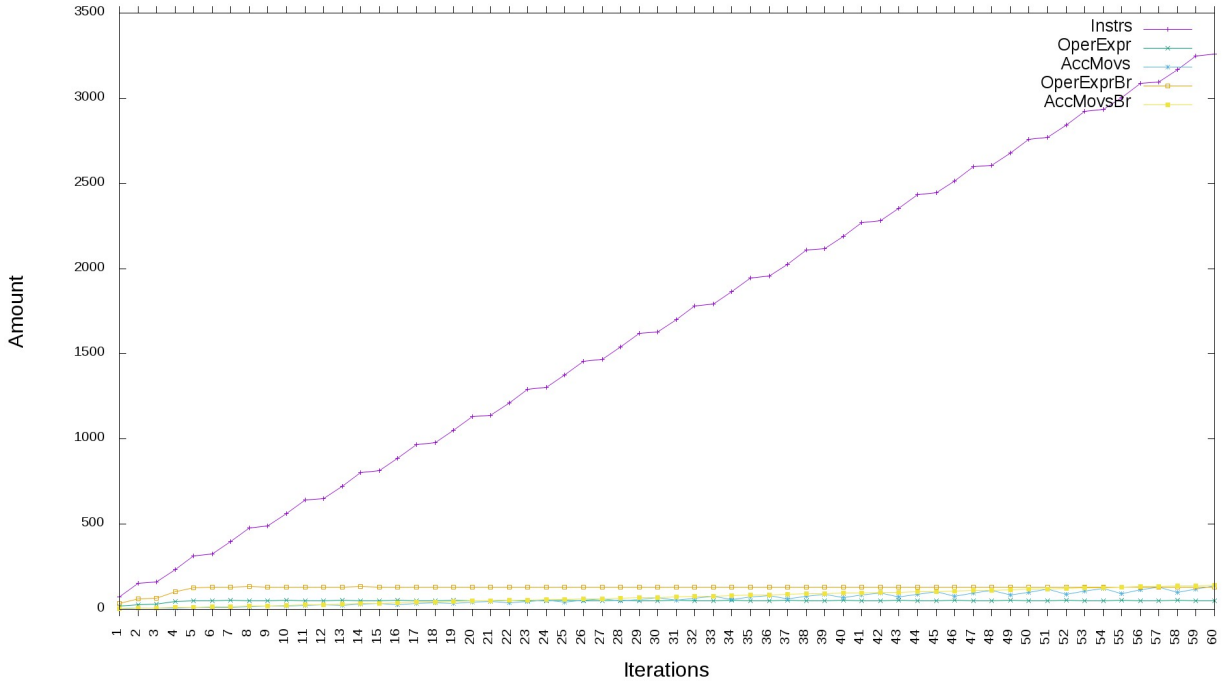

Figure 27: Mini-Language interpreter results for the simple loop program, when using RESET instead of GOTO.

A final, minor observation is that the amount of `mov`-family instructions attached to operations increases. Nodes are merged on creation, yet they retain the instructions they are linked to. Since the number of nodes remains the same, an increase of the amount of `mov`-family instructions signifies that the amount of these instructions per node has increased. So, the essence of the computation (the symbolic expressions) stays the same but each loop adds more `mov`-family instructions on the (same) symbolic nodes.

5. CONCLUSION

In this section we describe related work (Section 5.1) and conclude (Section 5.2).

5.1 Related Work

The following are related works on the dynamic tracing and optimization of interpreters:

Optimizing interpreters with DynamoRIO. Sullivan *et al.* presented a modification to DynamoRIO that allowed for passing of information regarding to the control flow of an interpreter, using API calls in the source code of the interpreter [15]. Their work also inserts API calls to pass information about the memory; that information can be immutable regions of memory, immutable targets of calls for the abstract program counter of the interpreter, and locations of stack variables. The information is exploited by the modified DynamoRIO to allow for the creation of traces based on the logical control flow and the folding of constant processing overhead to constants. The optimizations performed are constant propagation, call-return matching and dead code elimination.

This approach is relevant only to the optimization of interpreters. It presents a complete solution with dynamic tracing that, with the help of information by the interpreter creator, offers speedups of up to 2x over native execution.

Our approach of symbolically executing a trace is unaware of the physical or logical control flow and logical variables, hence it cannot perform constant propagation in the context of the interpreter. It is aware of addresses that, as a sum of expressions and constant can be found similar, if different subexpressions or combinations of additions and subtractions of different subexpressions do not produce the same results (assumes no aliases), and the constant propagation it can perform is limited to this. The approach of symbolically executing the instructions reveals patterns in the trace code allowing for common subexpression elimination where it occurs with the same inputs, while it can also be further improved in the future to recognize broader patterns.

The meta-tracing approach. When applying our prototype to interpreters (compiled to binary form), our work is also related to the meta-tracing approach of Carl Friedrich Bolz [16]. As with the previous approach, meta-tracing requires markers inserted in the sources of the interpreter. This work is similar to that of Sullivan *et al.* in regards to placing markers in the source code of the interpreter to expose information to the optimizing layer. This work provides a separate language, RPython, to the interpreter creator, and transforms the interpreter to a Just-In-Time compiler by using the markers embedded in the language.

Program slicing. Our dead code elimination via symbolic execution is similar to dead code detection via program slicing [17, §14.2.6]: our approach can be seen as an implementation of *dynamic backwards executable slicing* [18]. *Dynamic* because it happens at runtime and the trace is assumed to have followed certain execution paths. *Backwards* because it starts from the symbolic expressions in the final state. *Executable* because the slice that is produced must be an executable machine code fragment. Our approach can also be seen as *conditional slicing* using a symbolic executor, where the runtime path condition of the control flow is the input condition and the produced program is the *conditioned program* [19, 20]. Since we do not evaluate symbolically branch instructions, our symbolic trees offer information equivalent to flow dependence graphs used for program slicing [17, §14.3.1.2].

5.2 Remarks

In conclusion, we have shown that following the execution path of real world programs we can use symbolic execution to discern patterns and optimize the executed code, and in specific cases remove major redundant, repeating computations. We have shown that extra information is needed to discover patterns when memory is referenced in non-trivial ways, either via analysis or via information provided by the developer of the instrumented program.

We have also shown that the processing of the instructions can be done in $O(n)$ complexity where n is the amount of instructions, assuming $O(1)$ complexity of our mapping structures.

We have found that tracing a binary that implements an interpreter poses a special problem: the “program counter” of the interpreted language and the ways it is manipulated will have been compiled to some pieces of machine code that may be difficult to recognize. Unless the interpreter can inform us what values correspond to the program counter (what Sullivan *et al.* call the “Logical PC” [15]), we cannot know what piece of memory contains the bytecode instructions of the interpreted program. If we knew that memory, we could assume that it is read-only (few interpreters permit self-modifying bytecode) and proceed to read the instructions and their operands, which would then be guaranteed to be constants. This would permit us to eliminate the level of indirection created by the interpreter and let us work directly on traces of bytecode.

As future work, we aim to find means for retrieving data information, such as read-only sections; we aim to further optimize the processing time of instructions by simplifying the structures used; to provide a means of recognizing logical control flow in programs (control flow decided by indirect branching) and to dynamically handle the execution flow so that the traces created can actually replace the originals in real time.

ACRONYMS AND ABBREVIATIONS

JIT	Just-In-Time
CISC	Complex Instruction Set Computing
API	Application Programming Interface
DAG	Directed Acyclic Graph
AT&T	American Telephone & Telegraph Company
BCD	Byte-Coded Decimal
PC	Program Counter

REFERENCES

[1] R. Wilhelm and H. Seidl, *Compiler Design: Virtual Machines*. Springer Publishing Company, Incorporated, 1st ed., 2010.

[2] "Learn About Java Technology." `http://java.com/en/about`. [accessed 30/8/2016].

[3] D. L. Bruening, *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004. AAI0807735.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[5] "Graphviz - Graph Visualization Software." `http://www.graphviz.org/`. [accessed 25/10/2016].

[6] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, (New York, NY, USA), pp. 1–12, ACM, 2000.

[7] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: Less is more," pp. 202–211, 2000.

[8] J. Allen, *Anatomy of LISP*. New York, NY, USA: McGraw-Hill, Inc., 1978.

[9] J.-C. Filliâtre and S. Conchon, "Type-safe modular hash-consing," in *Proceedings of the 2006 Workshop on ML*, ML '06, (New York, NY, USA), pp. 12–19, ACM, 2006.

[10] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, pp. 385–394, July 1976.

[11] "Intel® 64 and IA-32 Architectures Software Developer's Manual." `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`. [accessed 19/6/2016].

[12] "The uBASIC interpreter." `http://dunkels.com/adam/ubasic/`. [accessed 24/7/2016].

[13] J. Huang and P. Tzu-Chin, "Analysis of x86 instruction set usage for DOS/Windows applications and its implication on superscalar design," *IEICE Transactions on Information and Systems*, vol. 85, no. 6, pp. 929–939, 2002.

[14] "The Programming Language Lua." `https://www.lua.org/`. [accessed 21/10/2016].

[15] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe, "Dynamic native optimization of interpreters," in *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03, (New York, NY, USA), pp. 50–57, ACM, 2003.

[16] J. Laval, A. Kellens, C. F. Bolz, and L. Tratt, “The impact of meta-tracing on VM design and implementation,” *Science of Computer Programming*, vol. 98, pp. 408 – 421, 2015.

[17] Y. N. Srikant and P. Shankar, *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. Boca Raton, FL, USA: CRC Press, Inc., 2nd ed., 2007.

[18] D. W. Binkley and K. B. Gallagher, “Program slicing,” vol. 43 of *Advances in Computers*, pp. 1 – 50, Elsevier, 1996.

[19] G. Canfora, A. Cimitile, and A. D. Lucia, “Conditioned program slicing,” *Information and Software Technology*, vol. 40, no. 11–12, pp. 595 – 607, 1998.

[20] A. D. Lucia, “Program slicing: methods and applications,” in *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 142–149, 2001.