STOCHASTIC PROGRAM OPTIMIZATION FOR x86_64 BINARIES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Eric Schkufza

August 2015

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Alex Aiken, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Pat Hanrahan

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alex Aiken) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Pat Hanrahan)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Mark Horowitz)

Approved for the Stanford University Committee on Graduate Studies

Acknowledgments

This thesis and the little bits of sanity that I still lay claim to having made it through grad school, would not be possible without the love and support of my friends and family. In keeping with the spirit of this work, the ordering of names that I've used below was chosen uniformly at random, so that none may feel slighted. Special dispensation — and the honor of being named first— is given to my advisor Alex Aiken, and to those who will never have the chance to read these words, Nancy Schkufza, Walter Lyba, and Sumit Mallik.

To everyone else, Lori Schkufza, Liz Fornero, Zach Ernst, Mark Horowitz, Jake Herczeg, Grant Toeppen, Pat Hanrahan, Zach DeVito, Paul Merrell, Kristen Bashaw, Percy Liang, Mike Spivack, Eva Lyba, Evan Cox, Aaron Apple, David Cook, Oliver Hardt, Chie Kawahara, Stefan Heule, Joe Schkufza, James Larus, Pat Hanrahan, Mike Bauer, Brion Spensieri, Trishul Chilimbi, Deian Stefan, Tara Bott, Peter Hawkins, Mike Genesereth, Niels Joubert, Matt Fornero, Sean Treichler, Berkeley Churchill, Alex Pagliere, Adam Oliner, Noah Goodman, Ben Handy, Madeline Roe, Rahul Sharma, Joana Ngo, Vladlen Koltun, Gleb Denisov, Ian Vo, thank you for everything.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis is about the aggressive optimization of high-performance code. We describe a non-traditional approach to the problem and focus on application domains where every choice of instruction and register matters. The key result of this work is that the high volume application of small random transformations is sufficient for producing very highly optimized code sequences that are capable of outperforming both the code produced by production compilers, and in many cases expert hand-written assembly.

While there is considerable value in producing the most performant code possible, the traditional structure of a compiler's optimization phase is ill-suited to this task. Factoring the optimization problem into a collection of small subproblems that can be solved independently — although suitable for generating consistently good code — in general leads to sub-optimal results. In most cases, the very best code sequences can only be obtained through the simultaneous consideration of mutually dependent issues that span multiple levels of abstraction and require cross-domain expertise. This process can require a deep understanding of a target architecture and how it interacts with machine resources, mutually dependent issues such as instruction selection and register allocation, and even the ability to leverage application-level requirements that affect how correctness and precision can be traded in favor of additional performance improvements.

The primary alternative to this approach is the use of search-based enumeration techniques. In contrast to a traditional compiler, which uses performance constraints to drive the generation of a single sequence, these systems consider multiple sequences and select the one that is best able to satisfy those constraints. An attractive feature of these approaches is completeness: if a sequence exists that meets the desired constraints then it is guaranteed to be found. However, completeness also places a severe limitation on the type of code that can considered in practice. The applicability of these techniques is either limited to sequences that are shorter than the threshold at which many interesting optimizations take place, or to code sequences written in simplified languages that correspond to a more limited class of kernels than can be formed with a general-purpose hardware instruction set.

In this thesis, we describe a method that overcomes these limitations through the use of incomplete random search. This method is presented in the abstract in Chapter 2 where we reformulate the competing constraints of correctness and performance improvement as terms in a cost function defined over the space of all code sequences, and recharacterize the optimization task as a cost minimization problem. Although the resulting search space is highly irregular and not amenable to exact optimization techniques we survey a family of random search procedures – notably Markov Chain Monte Carlo (Mcmc) sampling – and describe how they can be used to rapidly generate low-cost samples corresponding to high-quality code sequences. In Chapter 3 we describe a concrete implementation of these concepts for the `x86_64` instruction set.

Beginning with Chapter 4, we describe the application of this technique to increasingly more complex application domains. In Chapter 4 we describe the optimization of loop-free code sequences that contain entirely fixed-point computations and require the preservation of full bit-wise correctness for all possible inputs. In Chapter 5 we consider the optimization of code sequences that contain mixed fixed- and floating-point computations and show that we are able to obtain substantial performance improvements by relaxing the requirement that optimizations preserve floating-point semantics exactly as written. In Chapter 6, we examine the problem of optimizing code sequences that contain loop constructs and extend the results of the previous chapters to this new domain. Finally, in Chapter 7 we consider the optimization of code sequences with semantic requirements that transcend the traditional compiler's definition of correctness such as hash functions, random number generators, and malware attack strings.

We conclude briefly in Chapter 8 with a summary of our contributions. While much of the work in this thesis may seem crazy and beyond practical application, it is our hope that by its end the reader will be convinced that only the former is true.

Chapter 2

Formalism

In this thesis we describe the non-standard approach of using incomplete random search as a method for rapidly exploring the extremely high-dimensional space of all possible code sequences. Although the chapters that follow deal exclusively with the `x86_64` instruction, we begin with a discussion of the underlying formalism in the abstract. In addition to providing the formal background for the chapters that follow, this material has value in itself as a generalization of many of the apparently disparate ideas that appear in the area of compiler research.

We start by introducing the idea of an abstract space of code sequences, and the application of code transformations as a mechanism for navigating that space. Once we have established the idea of a search space, we introduce the idea of cost functions over code sequences. Using these definitions we examine the idea that two apparently unrelated compiler tasks – program synthesis and optimization – are simply search problems, and reformulate each in terms of the definitions that we have developed.

2.1 Code Sequences

Defining the abstract space of all possible code sequences requires that we first have an abstract definition of hardware states. This definition should be simple enough to generalize to most practical implementations, but also expressive enough to serve as an underlying data-structure for defining the semantics of code sequences in terms of the transformations that they produce. A simple choice that serves this purpose is an abstract collection of memory cells that contain sequences of bit values.

Definition 1 (Bit-vector)**.** A *bitvector* ($b \in \mathcal{B}^k$) of arity k is a sequence of k zeros or ones. We omit the k superscript when its use is clear.

Definition 2 (Hardware Location)**.** A *hardware location* ($\ell \in \mathcal{L}$) is a architecture-specific nameable machine address. We refer to the set of all possible sets of hardware location as $\mathbb{L}$.

Definition 3 (Machine State)**.** A *machine state* ($m(\cdot) : \mathcal{L} \to \mathcal{B}$) is a mapping from a set of hardware locations to bit-vectors of appropriate width. A machine state may not contain values for all possible hardware locations. Locations that do not appear in a machine state are said to be *undefined.* We refer to the set of all possible machine states as $\mathbb{M}$.

Using this abstraction we define a hardware instruction set as a collection of instructions that can both modify machine states and induce control flow by modifying a program counter. For the purpose of applying standard dataflow analyses we provide a general characterization of the effects that an instruction can produce on a machine state, and for reasoning about runtime behavior we define the dynamic sequence of transformations that a code sequence produces on machine states when executed.

Definition 4 (Instruction)**.** An *instruction* ($i(\cdot) : \mathbb{M} \to \mathbb{M} \times \mathbb{Z}$) is a statement written in a *hardware instruction set* ($\mathbb{H}$). Instructions produce both a transformation in machine state and an integer program counter displacement. We refer to the application of an instruction to a machine state as an *execution.*

Definition 5 (Code Sequence)**.** A *code sequence* ($\mathcal{S}$) is an ordered list of instructions. We refer to the j^{th} element of a code sequence i_j as $\mathcal{S}_j$.

$$\mathcal{S} = \Big\langle i_1, i_2, \dots i_n \Big\rangle \mid i_1, i_2, \dots i_n \in \mathbb{H} \tag{2.1}$$

Definition 6 (Dataflow Sets)**.** An execution may read from, write to, or undefine (remove) one or more of the locations in a machine state. We refer to these sets of locations using the *read set* ($\rho(\cdot) : \mathbb{H} \times \mathbb{M} \to \mathbb{L}$), *write set* ($\omega(\cdot) : \mathbb{H} \times \mathbb{M} \to \mathbb{L}$), and *undef set* ($\upsilon(\cdot) : \mathbb{H} \times \mathbb{M} \to \mathbb{L}$) functions.

Definition 7 (Runtime Sequences)**.** An *execution sequence* ($\mathcal{E}^{\mathcal{S},m}$) is an ordered list of instructions that results from executing a code sequence $\mathcal{S}$ beginning with its first instruction in m. A *machine sequence* ($\mathcal{M}^{\mathcal{S},m}$) is the corresponding ordered list of machine states. We refer to the j^{th} element of a dynamic sequence as $\mathcal{E}^{\mathcal{S},m}_j$ or $\mathcal{M}^{\mathcal{S},m}_j$, and the final element as $\mathcal{E}^{\mathcal{S},m}_{\text{end}}$ or $\mathcal{M}^{\mathcal{S},m}_{\text{end}}$. Both sequences may have unbounded length due to the presence of loops.

$$\begin{aligned}
\mathcal{E}^{\mathcal{S},m} &= \Big\langle \mathcal{S}_1, i_2, i_3 \dots \Big\rangle \\
\mathcal{M}^{\mathcal{S},m} &= \Big\langle m, m_2, m_3 \dots \Big\rangle \\
\text{where } (m_2, \delta_1) &= \mathcal{S}_1(m) \\
i_2 &= \mathcal{S}_{1+\delta_1} \\
(m_3, \delta_2) &= i_2(m_2) \\
i_3 &= \mathcal{S}_{1+\delta_1+\delta_2} \\
&\dots
\end{aligned} \tag{2.2}$$

```
# Length 2 Sequence

xorq rax, rax
ret
```

```
# Length 5 Sequence

PASS
xorq rax, rax
PASS
retq
PASS
```

Figure 2.1: Enforcing a fixed-length on code sequences. A length 2 code sequence (left) in the `x86_64` hardware instruction set is transformed into a length 5 code sequence (right) by inserting three copies of the PASS instruction.

Although our goal is to characterize the set of all possible code sequences, it is useful to preemptively rule out as many semantically ill-formed sequences as possible by construction. Although the notion of semantic well-formedness is difficult to define in general, and different application areas may impose additional or different constraints, we provide a definition here that will suffice for the remainder of this thesis. Our discussion is limited to code sequences of bounded length that do not perform reads from undefined hardware locations.

Definition 8 (Pass Instruction)**.** PASS is a distinguished instruction that does not appear in any hardware instruction set. It has no observable semantics with respect to machine states other than to advance the program counter and serves only to take the place of an instruction.

$$\text{PASS}(m) = \Big(m, 1\Big) \tag{2.3}$$

Definition 9 (Fixed-Length Code Sequence)**.** A *fixed-length code sequence* ($\mathcal{S}^k$) is a code sequence of arity k. Code sequences with arity less than k can be given a fixed-length by inserting one or more copies of the PASS instruction as shown in Figure 2.1. We omit the PASS instruction and the superscript k when their use is clear.

$$\mathcal{S}^k = \Big\langle i_1, i_2, \dots i_k \Big\rangle \mid i_1, i_2, \dots i_k \in \mathbb{H} \cup \text{PASS} \tag{2.4}$$

Definition 10 (Undefined Read)**.** An *undefined read* ($\text{undef}(\cdot) : \mathbb{H} \times \mathbb{M} \to \{\top, \bot\}$) is an execution of an instruction i in a machine state m that results in a read from an undefined location.

$$\text{undef}(i, m) = \exists \ell .\ \ell \in \rho(i) \wedge \ell \notin \mathit{Dom}(m) \tag{2.5}$$

Definition 11 (Compatible Machine States)**.** A machine state is *compatible* with a set of locations if it contains a mapping for each element of that set. We use the *compatibility function* ($\text{compat}(\cdot) :$

$\mathbb{L} \rightarrow \mathbb{M}$) to represent the set of machine states that are compatible with a set of locations.

$$\text{compat}(\mathcal{L}) = \Big\{ \text{m} \mid \mathcal{L} \subseteq Dom(\text{m}) \Big\} \tag{2.6}$$

Definition 12 (Well-Formed Code Sequences)**.** A *well-formed code sequence* ($\mathcal{S} \in \mathbb{S}^{k,\mathcal{L}}$) is a fixed-length code sequence for which the execution sequences defined by machine states compatible with $\mathcal{L}$ do not contain an undefined read.

$$\mathbb{S}^{k,\mathcal{L}} = \Big\{ \mathcal{S} \mid \forall m \in \text{compat}(\mathcal{L}).\ \neg\exists \text{j}.\ \text{undef}(\mathcal{E}_{\text{j}}^{\mathcal{S},\text{m}}, \mathcal{M}_{\text{j}}^{\mathcal{S},\text{m}}) \Big\} \tag{2.7}$$

Using this formal characterization of the abstract space of all code sequences we define a simple notion of connectivity in terms of transformations over elements in the space; any two elements that can be transformed into one another are considered to be adjacent. As a final bit of terminology, we introduce the notion of a target – a code sequence of interest for a particular application – and the set of all of other code sequences, which we refer to as rewrites.

Definition 13 (Target)**.** The *target* ($\mathcal{T}$) is an application-dependent distinguished code sequence.

Definition 14 (Transformation)**.** A *transformation* ($\mathcal{F}(\cdot) : \mathbb{S}^{k,\mathcal{L}} \rightarrow \mathbb{S}^{k,\mathcal{L}}$) is any mapping over well-formed code sequences. Transformations may be composed using the standard notation:

$$(\mathcal{F}^1 \circ \mathcal{F}^2)(x) \equiv \mathcal{F}^2(\mathcal{F}^1(x)) \tag{2.8}$$

Definition 15 (Rewrite)**.** A *rewrite* ($\mathcal{R}$) is any well-formed code sequence that can be produced through some repeated application of transformations to the target.

$$\mathcal{R} \in \Big\{ r \mid r = (\mathcal{F}_1 \circ \mathcal{F}_2 \circ \ldots \mathcal{F}_n)(\mathcal{T}) \Big\} \tag{2.9}$$

2.2 Cost Functions

Given the formalism developed above, it is straightforward to define a cost function over the elements in the abstract space of code sequences. Although we have considerable flexibility in how we choose to do so, for the purposes of this thesis it is sufficient that our definition simply capture the distinction between hard constraints (which must be satisfied) and soft constraints (for which better solutions are preferable, but not required). To preserve full generality, both hard and soft constraints may be defined in terms of both the target and an arbitrary code sequence.

Definition 16 (Hard Constraint)**.** A *hard constraint* ($\text{hard}(\cdot) : \mathbb{S}^{k,\mathcal{L}} \times \mathbb{S}^{k,\mathcal{L}} \rightarrow \mathbb{Z}^+$) is a non-negative function that represents an application-specific constraint that must be satisfied. A return value of zero indicates success whereas all other values indicate failure.

$$\text{hard}(\mathcal{T},\mathcal{S}) = \begin{cases} 0 & \text{constraints satisfied} \\ > 0 & \text{otherwise} \end{cases} \tag{2.10}$$

Definition 17 (Soft Constraint)**.** A *soft constraint* ($\text{soft}(\cdot) : \mathbb{S}^{k,\mathcal{L}} \times \mathbb{S}^{k,\mathcal{L}} \to \mathbb{Z}^+$) is a non-negative function that represents an application-specific constraint for which all code sequences are permissible, but some are preferable. Lower return values correspond to more preferable sequences.

Definition 18 (Cost Function)**.** A *cost function* ($\text{cost}(\cdot) : \mathbb{S}^{k,\mathcal{L}} \times \mathbb{S}^{k,\mathcal{L}} \to \mathbb{Z}^+$) is a weighted combination of hard and soft constraint terms. We refer to the hard and soft constraints of a cost function as $\text{cost}_{\text{hard}}$ and $\text{cost}_{\text{soft}}$, and the set of all cost functions as $\mathbb{C}$.

$$\text{cost}(\mathcal{T},\mathcal{S}) = w_{\text{h}} \cdot \text{hard}(\mathcal{T},\mathcal{S}) + w_{\text{s}} \cdot \text{soft}(\mathcal{T},\mathcal{S}) \tag{2.11}$$

Using this notation we define two useful relations over the abstract space of code sequences with respect to the target. One identifies the set of code sequences that satisfy the hard constraints of an application domain, and the other identifies the set of code sequences that additionally improve the soft constraints of an application domain.

Definition 19 (Sufficient Rewrites)**.** A rewrite is *sufficient* ($\text{suff}(\cdot) : \mathbb{C} \times \mathbb{S}^{k,\mathcal{L}} \times \mathbb{S}^{k,\mathcal{L}} \to \{\top,\bot\}$) if it produces a hard constraint cost of zero when evaluated with respect to the target.

$$\text{suff}(\text{cost},\mathcal{T},\mathcal{R}) = \Big(\text{cost}_{\text{hard}}(\mathcal{T},\mathcal{R}) = 0\Big) \tag{2.12}$$

Definition 20 (Preferable Rewrites)**.** A rewrite is *preferable* ($\text{pref}(\cdot) : \mathbb{C} \times \mathbb{S}^{k,\mathcal{L}} \times \mathbb{S}^{k,\mathcal{L}} \to \{\top,\bot\}$) if it is both sufficient and produces a soft constraint cost that is less than the one that is obtained when the target is evaluated with respect to itself.

$$\text{pref}(\text{cost},\mathcal{T},\mathcal{R}) = \Big(\text{suff}(\text{cost},\mathcal{T},\mathcal{R}) \wedge \Big(\text{cost}_{\text{soft}}(\mathcal{T},\mathcal{R}) \leq \text{cost}_{\text{soft}}(\mathcal{T},\mathcal{T})\Big)\Big) \tag{2.13}$$

2.3 Synthesis and Optimization

We are now ready to state the non-standard but hopefully obvious assertion that many apparently disparate tasks in the area of compiler design are instances of cost minimization problems. To do so, we must first introduce two concrete instances of cost function constraints: a hard constraint that represents equality with respect to the target, and a soft constraint that represents the performance properties of a rewrite. To aid in defining the former, we introduce terminology for describing the side effects of a code sequence.

Definition 21 (Lifted Write Set)**.** We lift the definition of the write set function to well-formed code sequences ($\omega(\cdot) : \mathbb{S}^{k,\mathcal{L}} \times \mathbb{M} \rightarrow \mathbb{L}$) as follows.

$$\begin{aligned}
&\omega(\mathcal{S}, m) = \mathrm{w}_n \\
&\text{where } n = |\mathcal{E}^{\mathcal{S},m}| \\
&\qquad \mathrm{w}_1 = \omega(\mathcal{E}_1^{\mathcal{S},m}, \mathcal{M}_1^{\mathcal{S},m}) - \upsilon(\mathcal{E}_1^{\mathcal{S},m}, \mathcal{M}_1^{\mathcal{S},m}) \\
&\qquad \mathrm{w}_2 = \Big(\mathrm{w}_1 \cup \omega(\mathcal{E}_2^{\mathcal{S},m}, \mathcal{M}_2^{\mathcal{S},m})\Big) - \upsilon(\mathcal{E}_2^{\mathcal{S},m}, \mathcal{M}_2^{\mathcal{S},m}) \\
&\qquad \ldots
\end{aligned} \tag{2.14}$$

Definition 22 (Side Effects)**.** The *side effects* ($\sigma(\cdot) : \mathbb{S}^{k,\mathcal{L}} \times \mathbb{M} \times \mathcal{L} \rightarrow \mathcal{B}$) of a code sequence are the bit-vector values that it places in the elements of its lifted write set.

$$\sigma(\mathcal{S}, m, \ell) = \begin{cases} \mathcal{M}_{\text{end}}^{\mathcal{S},m}(\ell) & \ell \in \omega(\mathcal{S}, m) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2.15}$$

Definition 23 (Live Outputs)**.** The *live outputs* ($\lambda(\cdot) : \mathbb{S}^{k,\mathcal{L}} \times \mathbb{M} \rightarrow \mathbb{L}$) of a code sequence are a distinguished subset of its lifted write set that may be dereferenced by subsequent instruction executions.

Definition 24 (Equality Term)**.** An *equality term* ($\mathrm{eq}(\cdot) : \mathbb{S}^{k,\mathcal{L}} \times \mathbb{S}^{k,\mathcal{L}} \rightarrow \mathbb{Z}^+$) is a hard constraint that compares two code sequences and returns zero if and only if they are equal. For our purposes we define equality as the property of preserving all bit-vector side effects in the live output locations defined by the target. This definition allows us to ignore functional differences in the use of temporary values that do not affect a code sequence's input output semantics.

$$\mathrm{eq}(\mathcal{S}^1, \mathcal{S}^2) = \begin{cases} 0 & \forall m.\ \forall \ell \in \lambda(\mathcal{S}^1, m).\ \sigma(\mathcal{S}^1, m, \ell) = \sigma(\mathcal{S}^2, m, \ell) \\ > 0 & \text{otherwise} \end{cases} \tag{2.16}$$

Definition 25 (Performance Term)**.** A *performance term* ($\mathrm{perf}(\cdot) : \mathbb{S}^{k,\mathcal{L}} \times \mathbb{S}^{k,\mathcal{L}} \rightarrow \mathbb{Z}^+$) is a soft constraint that quantifies the application-specific performance improvement (e.g. runtime, power consumption, etc.) of a code sequence with respect to a reference implementation. The extent to which this term is an accurate representation of the behavior of the underlying architecture determines the extent to which it can adequately represent the performance trade-offs associated with program transformations.

Using these constraints, the connection to program synthesis and optimization as defined by a traditional compiler is straightforward. Program synthesis is the search for functionally correct rewrites and optimization is the search for rewrites that are both functionally correct and represent a performance improvement over the target. We explore solutions to both of these tasks in subsequent chapters, program synthesis in Chapters 4 and 7 and program optimization in Chapters 4, 5, 6, and 7.

Definition 26 (Program Synthesis)**.** The goal of *program synthesis* is to produce elements from the set of code sequences that are functionally equivalent to the target. This set consists of rewrites that are sufficient with respect to the target and the equality term described above.

$$\begin{aligned} \text{result} \subseteq \Big\{ r \in \mathbb{S}^{k,\mathcal{L}} \;\;\big|\;\; \text{suff}(\text{cost}, \mathcal{T}, r) \Big\} \\ \text{where cost} = w_{\text{h}} \cdot \text{eq}(\mathcal{T}, r) \end{aligned} \tag{2.17}$$

Definition 27 (Program Optimization)**.** The goal of *Program Optimization* is to produce elements from the set of functionally correct code sequences that represent a performance improvement over the target. This set consists of rewrites that are preferable with respect to the target and both the equality and performance terms described above.

$$\begin{aligned} \text{result} \subseteq \Big\{ r \in \mathbb{S}^{k,\mathcal{L}} \;\;\big|\;\; \text{pref}(\text{cost}, \mathcal{T}, r) \Big\} \\ \text{where cost} = w_{\text{h}} \cdot \text{eq}(\mathcal{T}, r) + w_{\text{s}} \cdot \text{soft}(\mathcal{T}, r) \end{aligned} \tag{2.18}$$

2.4 Solution Methods

Generating elements from either of the sets described above requires the use of a cost minimization procedure. Although many such approaches exist, in general we expect cost functions that capture complicated features such as semantic equality and performance improvement to be highly irregular and not amenable to exact optimization techniques. For functions of this form which are highly non-convex and characterized by sharp discontinuities and many local minima, the only know solution methods that are also tractable involve the use of a random search procedure.

Despite the apparent variety, most random search procedures can be shown to be instances of the generalized algorithm shown in Figure 2.2. The basic idea is simple. The algorithm maintains a current rewrite $\mathcal{R}$ and uses a *transformation kernel* ($\mathbb{F}(\cdot)$) to propose a modified rewrite. The *proposal* ($\mathcal{R}'$) is then either accepted or rejected based on the evaluation of an *acceptance function* ($\alpha(\cdot)$) which is defined in terms of the cost function. If the proposal is accepted then $\mathcal{R}'$ becomes the current rewrite. Otherwise, another proposal based on $\mathcal{R}$ is generated in its place. The algorithm iterates until its computational budget is exhausted, and the set of preferable rewrites that were discovered in the process is returned as a result.

Most random search procedures are agnostic with respect to transformation kernel and require only that it be *ergodic* (sufficient to transform any rewrite into any other through some repeated sequence of applications). Empirically, it is also desirable for a transformation kernel to produce a wide variety of modifications that strike a balance between inducing small local changes and large global changes to the current rewrite. In practice, transformation kernels that have this property

```
procedure SEARCH($\mathcal{T}$, $\mathcal{R}$, $\mathbb{F}$, $\alpha$, cost)
    result = $\{\mathcal{T}\}$
    repeat
        $\mathcal{R}' \sim \mathbb{F}(\mathcal{R})$
        if $\alpha(\mathcal{T}, \mathcal{R}, \mathcal{R}', \text{cost})$ then
            $\mathcal{R} = \mathcal{R}'$
            if $\text{pref}(\text{cost}, \mathcal{T}, \mathcal{R})$ then
                result = result $\cup\, \mathcal{R}$
            end if
        end if
    until timeout
    return result
end procedure
```

Figure 2.2: Generalized random search procedure. A transformation kernel ($\mathbb{F}(\cdot)$) is used to repeatedly modify an initial rewrite ($\mathcal{R}$). Transformations are accepted or rejected based on an acceptance function ($\alpha(\cdot)$) and a set of preferable rewrites is returned when a computational budget is exhausted.

tend to produce a higher volume of lower cost rewrites in a smaller number of proposals. We describe the implementation of a transformation kernel for `x86_64` that satisfies these criteria in Chapter 3.

Random search procedures differ primarily in their definition of acceptance function. Before describing MCMC sampling, the search procedure that we focus on in this thesis, we discuss the acceptance function for several other well known procedures and briefly characterize the formal guarantees that they provide. In principle, nothing prevents the use of these other methods for solving the optimization problem described above. In our experience, MCMC sampling simply offers an attractive trade-off between complexity of implementation of quality of experimental results. We compare the performance of MCMC sampling and these alternative methods in Chapters 4 and 5.

Any of the random search procedures shown below can be generalized to the simultaneous modification of multiple rewrites by lifting $\mathcal{R}$ to a set, and extending $\mathbb{F}$ to include both point- and pair-wise transforms. Algorithms of this form are collectively referred to as *population-based methods.* Although we do not consider their use in this thesis they are a common choice for extending the applicability of random search techniques and will almost certainly find use in future work.

Definition 28 (Pure Random Search)**.** The simplest implementation of a random search procedure is *pure random search*, which is characterized by an acceptance function that always returns true. Pure random search is guaranteed in the limit to enumerate the set of all preferable rewrites. However, for even modestly sized domains the number of proposals that must be evaluated to enumerate even a small fraction of that set is so prohibitively large as to be useless in practice. For most applications, pure random search offers extremely slow runtimes and poor overall performance.

$$\alpha_{\text{pure}}(\mathcal{T}, \mathcal{R}, \mathcal{R}', \text{cost}) = \top \tag{2.19}$$

Definition 29 (Greedy Hill Climbing)**.** A small refinement to pure random search results in *greedy hill climbing*, which is characterized by an acceptance function that returns true if and only if a proposal results in a reduction in the value of the cost function. In contrast to pure random search, greedy hill climbing generally produces better overall results in much less time, but is prone to becoming stuck in local minima. Greedy hill climbing is extremely sensitive to initial conditions; once the lowest cost code sequence in the immediate vicinity of the initial rewrite has been discovered, no further progress can be made.

$$\alpha_{\text{greedy}}(\mathcal{T}, \mathcal{R}, \mathcal{R}', \text{cost}) = \Big(\text{cost}(\mathcal{T}, \mathcal{R}') \leq \text{cost}(\mathcal{T}, \mathcal{R})\Big) \tag{2.20}$$

Definition 30 (Simulated Annealing)**.** Another small refinement to greedy hill climbing results in *simulated annealing*, which is characterized by an acceptance function that sometimes also returns true for proposals that result in an increase in the value of the cost function. Although there are many ways to define such a function, a common method for doing so is shown below. The important properties of this function are the following. If $\mathcal{R}'$ produces a lower cost than $\mathcal{R}$, the proposal is always accepted. If $\mathcal{R}'$ produces a larger cost than $\mathcal{R}$, the proposal may still be accepted with a probability that decreases as a function of the ratio between costs associated with $\mathcal{R}'$ and $\mathcal{R}$. This property prevents simulated annealing from becoming trapped in local minima while remaining less likely to accept a proposal that is much worse than the available alternatives.

Simulated annealing derives its name from the *temperature constant* (β) which decays over time and controls the likelihood with which the acceptance function will accept a proposal that increases the value of the cost function. Intuitively, simulated annealing can be thought of a method for interpolating between pure random search and greedy hill climbing. If β_{init} is chosen large enough to overwhelm the costs associated with $\mathcal{R}'$ and $\mathcal{R}$, then early evaluations of the acceptance function will always return true. As time goes on and the value of β decays, the acceptance function evolves towards only accepting proposals that decrease the value of the cost function. In contrast to either alternative, simulated annealing tends to produce far superior results and is generally as effective as any other random search procedure that has some mechanism for dealing with local minima.

$$\begin{aligned}
\alpha_{\text{annealing}}(\mathcal{T}, \mathcal{R}, \mathcal{R}', \text{cost}) &= \Big(x \leq \min(1, \exp(k)\Big) \\
\text{where } k &= -\frac{1}{\beta_i} \cdot \Big(\text{cost}(\mathcal{T}, \mathcal{R}') - \text{cost}(\mathcal{T}, \mathcal{R})\Big) \\
x &\sim \text{uniform}(0, 1) \\
\beta_0 &= \beta_{\text{init}} \\
\beta_{i+1} &= w_{\text{decay}} \cdot \beta_i
\end{aligned} \tag{2.21}$$

Definition 31 (Mcmc Sampling)**.** Mcmc sampling is a technique for drawing elements from a probability density function in direct proportion to its value: regions of higher probability are sampled more often than regions of low probability. When applied to cost minimization, it has two attractive properties. In the limit, the most samples will be taken from the minimum (optimal) values of a function. And in practice, well before that behavior is observed, it functions similarly to simulated annealing as an intelligent hill climbing method which is robust against irregular functions that are dense with local minima. A common method [38] for transforming an arbitrary cost function into a probability density function is shown below, where β is a constant and Z is a partition function that normalizes the resulting distribution.

$$p(\mathcal{T},\mathcal{R}) = \frac{1}{Z}\exp\Big(-\beta\cdot\mathrm{cost}(\mathcal{T},\mathcal{R})\Big) \tag{2.22}$$

Although computing Z is generally intractable, the Metropolis-Hastings acceptance function is designed to explore density functions such as $p(\cdot)$ without having to do so directly, and will in the limit produce a sequence of samples that are distributed in proportion to their cost [48]. In the equation shown below, we use the notation $q(\mathcal{R}'|\mathcal{R},\mathbb{F})$ to represent the proposal distribution from which a new rewrite $\mathcal{R}'$ is sampled given the current rewrite $\mathcal{R}$ and the transformation kernel $\mathbb{F}(\cdot)$.

$$\begin{aligned}\alpha_{\mathrm{mh}}(\mathcal{T},\mathcal{R},\mathcal{R}',\mathrm{cost}) &= \left(x \le \min\left(1, \frac{p(\mathcal{T},\mathcal{R}')\cdot q(\mathcal{R}|\mathcal{R}',\mathbb{F})}{p(\mathcal{T},\mathcal{R})\cdot q(\mathcal{R}'|\mathcal{R},\mathbb{F})}\right)\right)\\ &\text{where } x \sim \mathrm{uniform}(0,1)\end{aligned} \tag{2.23}$$

In the event that the proposal distribution is *symmetric* ($q(\mathcal{R}'|\mathcal{R},\mathbb{F}) = q(\mathcal{R}|\mathcal{R}',\mathbb{F})$) and rewrites have fixed length — this simplifying assumption is crucial as it places a constant value on the dimensionality of the resulting search space [1] — the acceptance function can be reduced to the much simpler Metropolis ratio, which can be computed directly from $\mathrm{cost}(\cdot)$. Under these conditions, Mcmc sampling can be shown to be a special case of simulated annealing in which the temperature constant remains fixed. Intuitively, it can be thought of as occupying a fixed point in the space between pure random search and greedy hill climbing.

$$\begin{aligned}\alpha_{\mathrm{mh}}(\mathcal{T},\mathcal{R},\mathcal{R}',\mathrm{cost}) &= \left(x \le \min\left(1, \frac{p(\mathcal{T},\mathcal{R}')}{p(\mathcal{T},\mathcal{R})}\right)\right)\\ &= \Big(x \le \min\Big(1, \exp(k)\Big)\Big)\\ \text{where } k &= -\beta\cdot\Big(\mathrm{cost}(\mathcal{T},\mathcal{R}') - \mathrm{cost}(\mathcal{T},\mathcal{R})\Big)\\ x &\sim \mathrm{uniform}(0,1)\end{aligned} \tag{2.24}$$

Chapter 3

STOKE: A Stochastic Optimizer for x86_64

Beginning with this chapter, we transition from discussing code sequences in the abstract to the x86_64 instruction set. We start by describing the high-level design of STOKE, a prototype implementation of the concepts described in the previous chapter, and focus on techniques for improving efficiency. Random search procedures such as STOKE are generally effective only insofar as they are able to maintain a high throughput rate of proposals. As a result, the time spent proposing modifications to a rewrite and evaluating the terms of a cost function must be kept to an absolute minimum. While subsequent chapters build on the material described below to apply STOKE to an increasingly complex set of application domains, the implementation described here forms the core of what is necessary to successfully apply the use of random search procedures to exploring the space of all possible x86_64 code sequences.

3.1 Code Sequences

STOKE uses an in-memory data structure to represent code sequences as finite length vectors of x86_64 instructions. As described in the Chapter 2, the distinguished instruction PASS is used to enforce the constraint that all code sequences have fixed-length. For x86_64 code sequences, the semantics of PASS are identical to the nop instruction. STOKE additionally uses a dataflow model of the semantics of the x86_64 instruction set to maintain a control flow graph data structure on top of code sequences. These views are used for keeping track of basic block boundaries, identifying loops, and performing standard dataflow analyses.

Several components of STOKE require information related to the set of live or defined registers at a program point. For the most part these sets are computed using the standard fixed-point dataflow

```
# Original

# BB 1
xorq rax, rax
addq rax, rdi
PASS
jmp .L0

# BB2
.L0:
subq rdi, rsi
retq
```

```
# Insert

# BB1
xorq rax, rax
addq rax, rdi
sbbq rdi, rdi
jmp .L0

# BB2
.L0:
subq rdi, rsi
retq
```

```
# Delete

# BB1
PASS
addq rax, rdi
PASS
jmp .L0

# BB2
.L0:
subq rdi, rsi
retq
```

```
# Opcode

# BB1
xorq rax, rax
addq rax, rdi
PASS
jmp .L0

# BB2
.L0:
cmpq rdi, rsi
retq
```

```
# Operand

# BB1
xorq rax, rax
addq r15, rdi
PASS
jmp .L0

# BB2
.L0:
subq rdi, rsi
retq
```

```
# Swap

# BB1
xorq rax, rax
subq rdi, rsi
PASS
jmp .L0

# BB2
.L0:
addq rax, rdi
retq
```

```
# Resize

# BB1
xorq rax, rax
addq rax, rdi
jmp .L0

# BB2
.L0:
PASS
subq rdi, rsi
retq
```

Figure 3.1: Stoke transforms applied to a representative (original) code. Instruction moves (insert) or (delete) and instruction, (opcode) moves change opcodes, (operand) moves change operands, (swap) moves interchange the location of two instructions, and (resize) moves move PASS instructions between basic blocks.

algorithm. However we note that both general purpose and Sse vector registers can be decomposed into sub-registers. For example, `dil`, `di`, and `edi` refer to the lowest 8, 16, and 32-bits of `rdi`, respectively:

$$\texttt{dil} \subset \texttt{di} \subset \texttt{edi} \subset \texttt{rdi} \tag{3.1}$$

To account for this register aliasing when calculating the read set of an instruction Stoke includes the registers that it reads, along with all subsets of those registers. An instruction that reads `di` for example, also reads `dil`. Analogously, when calculating the write set of an instruction Stoke includes the registers that it writes, along with all super- and subsets of those registers. Finally, some instructions produce undefined values and are treated as having the effect of killing any live or defined registers.

Stoke uses the set of transformations show in Figure 3.1 to define the connectivity of the space of `x86_64` code sequences. The proposal distribution chooses from five possible moves: the first three are designed to induce major changes in the underlying code sequence whereas the last two are designed to induce minor changes. The distribution contains redundancy. For example, an instruction move can be decomposed into opcode and operand moves. However as described in the Chapter 2, there is empirical evidence to suggest that a combination of transformations that induce both minor and major changes can improve the performance of random search techniques.

- **Instruction.** An instruction is randomly selected and replaced either by a random instruction or the PASS instruction. As shown in Figure 3.1, proposing PASS corresponds to deleting an instruction and replacing PASS by an instruction corresponds to inserting an instruction.
- **Swap.** Two lines of code are randomly selected and interchanged.
- **Resize.** Two basic blocks are randomly selected and if a PASS instruction appears in one it is moved to the other. As shown in Figure 3.1, this corresponds to changing the number of instructions that can appear in a basic block while preserving a fixed length on code size.
- **Opcode.** An instruction is randomly selected and its opcode is replaced by a random opcode.
- **Operand.** An instruction is randomly selected and one of its operands is replaced by a random operand.

These transformations are simple to implement and satisfy the ergodicity property described in the previous chapter: any code sequence can be transformed into any other through repeated application of instruction moves. These moves also satisfy the symmetry property. To see why, note that the probabilities of performing all five moves are equal to the probabilities of undoing the transformations they produce using a move of the same type: opcode and operand moves are constrained to sample from identical equivalence classes before and after acceptance, and swap, resize, and instruction moves are unconstrained in either direction. Regardless of type, STOKE uses dataflow information to guarantee that a transformation never produces a code sequence that contains a read from an undefined register location. STOKE is similarly constrained in its inability to modify control flow. No sequence of transformations is permitted to insert, delete, or modify a label, jump, or function call.

We note briefly that the transformations shown above all have the potential to modify both STOKE's control flow graph view of a code sequence and the dataflow values that are derived from that graph. Because recomputing these data structures is expensive, STOKE takes a lazy approach to maintaining their values. For example, STOKE only recomputes the structure of a control flow graph after performing a resize move. Similarly, because an instruction move does not require the computation of every dataflow value in a graph, STOKE only recomputes the values at the program points that follow that instruction.

3.2 Cost Functions

For any reasonably sophisticated application domain, a satisfactory definition of the hard(·) and soft(·) constraint terms is likely to be prohibitively slow to compute in STOKE's inner-most proposal loop. Our implementation of STOKE takes a practical approach to this problem by using a two-tiered method for computing the terms of a cost function. This approach is formalized in the equations shown below which expand both the hard(·) and soft(·) constraint terms into fast and slow variants. In general, we expect the majority of the random transformations proposed by STOKE to either violate a hard constraint or disimprove a soft constraint. As a result, the fast variants of each term are tuned to quickly identify those cases, whereas the slow variants of each term are reserved for code sequences that pass the fast variants and warrant a more computationally intensive evaluation.

$$\begin{aligned} \text{hard}(\mathcal{T},\mathcal{S}) &= \begin{cases} \text{hard}_{\text{slow}}(\mathcal{T},\mathcal{S}) & \text{hard}_{\text{fast}}(\mathcal{T},\mathcal{S}) = 0 \\ \text{hard}_{\text{fast}}(\mathcal{T},\mathcal{S}) & \text{otherwise} \end{cases} \\ \text{soft}(\mathcal{T},\mathcal{S}) &= \begin{cases} \text{soft}_{\text{slow}}(\mathcal{T},\mathcal{S}) & \text{hard}(\mathcal{T},\mathcal{S}) = 0 \\ \text{soft}_{\text{fast}}(\mathcal{T},\mathcal{S}) & \text{otherwise} \end{cases} \end{aligned} \tag{3.2}$$

Many of the cost functions that we discuss in subsequent chapters have the property that they can be partially computed once when STOKE is initialized rather than from scratch for every proposed rewrite. We capture this property formally using the notion of function parameterization.

Definition 32 (Parametrization)**.** A function $f(x;y)$ takes arguments x and is *parameterized* by y. Any function $f(x,y)$ that can be partially computed in terms of a constant argument y can be promoted to a parameterized function $f(x;y)$.

In the chapters that follow, we also make use of the following notation when describing application-specific cost function definitions. Recall that $\sigma(\cdot)$ is the side effect that appears in a location after executing a code sequence $\mathcal{S}$ in state t. We say that $\text{val}_{\text{T}}(x,t)$ is the value of the bits at location x in state t when interpreted to have type T. For notational convenience we also define functions that represent the return value produced by executing a code sequence and interpreting the value in an x86_64 ABI-specified location in a manner that is appropriate for the expected type (e.g. *64-bit unsigned integer* (fixed) or *64-bit double-precision floating-point* (double)).

$$\begin{aligned} \text{ret}_{\text{fixed}}(\mathcal{S},t) &= \text{val}_{\text{fixed}}(\sigma(\mathcal{S},t,\texttt{rax})) \\ \text{ret}_{\text{double}}(\mathcal{S},t) &= \text{val}_{\text{double}}(\sigma(\mathcal{S},t,\texttt{xmm0}[\texttt{63}:\texttt{0}])) \end{aligned} \tag{3.3}$$

In many cases, the most natural way to define the fast versions of the hard and soft constraint terms is with respect to evaluation of both the target and rewrite on *test cases* (τ). For our purposes, we define a test case as a machine state that consists of assignments to the general purpose, SSE vector, and condition registers of the x86_64 architecture along with a set of values for defined locations in either the stack or the heap. Although a hand-written set of test cases is always preferable, STOKE has a mechanism for automatically generating test cases from a user's application. For large software projects, we expect that a representative set of whole-program tests will exist and STOKE may simply observe the behavior of the code during normal program execution. When no such inputs exist a user must provide a small unit test harness that will establish whatever input values are necessary to enable the target to execute correctly. In either case, STOKE generates test cases using Intel's PinTool [65]; it executes the binary provided by the user and for every invocation of the target records both initial register state and the set of values dereferenced from memory.

Because test case evaluation must be performed in STOKE's inner-most proposal loop, it is crucial that the time required to do so be minimized. Our implementation of STOKE evaluates rewrites using a JIT assembler that supports the entire x86_64 instruction set available on Intel's Haswell processors. This includes over 4000 variants of 400 distinct opcodes and Intel's newest fused multiply-add, AVX, and advanced bit manipulation instructions. Although the correctness assumptions that we make for the target (e.g. the absence of exceptional behavior and dereferences from undefined memory locations) do not necessarily hold for rewrites, even after we consider the overhead required to sandbox this behavior, our implementation is able to safely dispatch test cases at a rate of between 1 and 10 million per second on a single core.

Our implementation of STOKE protects against exceptional memory behavior by observing the behavior of the target. From the execution of the target on test cases we obtain the maximum number of stack locations which were used, and the minimum and maximum heap addresses that were dereferenced in its evaluation, and then use those values to define stack and heap sandboxes to guard the execution of a rewrite. Instructions that have the potential to perform memory dereferences are instrumented with a function that redirects valid accesses to each of these sandboxes, and traps all other accesses and replaces them by a safe premature termination.

In addition to sandboxing memory accesses, several other behaviors need to be checked to protect STOKE from undefined rewrite behavior. First, simpler types of exceptional behavior that do not have the potential to silently corrupt STOKE's internal state (e.g. floating-point exceptions) are handled by overloading the operating system's signal handling routines. Second, STOKE protects itself against rewrites that enter into infinite loops by emitting a small amount of code that counts the number of times that a back edge is taken and causes a premature termination if a bound calculated from the number of backwards jumps taken by the target execution is ever exceeded. Finally, STOKE guarantees that return instructions take place only after certain invariants specific to the x86_64 ABI have been restored. This property is guaranteed by emitting a function that checks and then guarantees that the value of callee-saved registers are restored to the state they held when a rewrite began executing.

3.3 Synthesis and Optimization

Stoke can be run in either synthesis or optimization mode. These modes share the same implementation and differ only in starting point and acceptance function. Synthesis mode begins with a random code sequence whereas optimization mode begins from a code sequence that is known to satisfy the hard constraints of an application domain. As described previously, whereas synthesis mode ignores the soft constraints of an application domain and only considers its hard constraints, optimization mode uses both. Doing so allows optimization mode to improve the soft constraints of an application domain while also experimenting with "shortcuts" that (temporarily) violate its hard constraints. We discuss the value of this approach in the following chapter.

3.4 Solution Methods

Our implementation of Stoke uses Mcmc sampling as a random search procedure. Although the separation of constraints into fast and slow variants is generally sufficient to achieve an acceptable proposal throughput, there are additional performance improvements that can be obtained. Notably, many instances of fast constraint terms are defined in terms of a set of test cases, and will involve a *reduction operator* ($\oplus$) over a sub-computation $f(\cdot)$ that is applied on a per-test basis. (Note in the example below the first use of parameterization, where we assume that the behavior of the target on the set of test cases may be precomputed once when Stoke is initialized.)

$$\text{constraint}_{\text{fast}}(\mathcal{S};\mathcal{T},\tau) = \bigoplus_{t\in\tau} f(\mathcal{S};\mathcal{T},\tau) \tag{3.4}$$

If $\oplus$ is monotonic (e.g. sum or max) then it is possible to rearrange the terms of the Metropolis Hastings acceptance function to take advantage of this property. As described previously, the acceptance function is computed by evaluating the cost function on a proposed rewrite, noting the ratio in total cost with the current rewrite, and then sampling a random variable x to decide whether to accept the proposal. However, by first sampling x and then computing the maximum ratio that the function will accept for that value, it is possible to terminate the evaluation of the cost function as soon as that bound is exceeded and the proposal is guaranteed to be rejected. We consider the effect of this optimization on Stoke's proposal throughput in the following chapter.

$$
\begin{aligned}
\alpha_{\mathrm{mh}}(\mathcal{T}, \mathcal{R}, \mathcal{R}', \mathrm{cost}) &= \left(x \leq \min\Big(1, \exp(k)\Big)\right) \\
\text{where } k &= -\beta \cdot \Big(\mathrm{cost}(\mathcal{R}'; \mathcal{T}, \tau) - \mathrm{cost}(\mathcal{R}; \mathcal{T}, \tau)\Big) \\
x &\sim \mathrm{uniform}(0, 1) \\
&= \left(\mathrm{cost}(\mathcal{R}'; \mathcal{T}, \tau) \leq \mathrm{cost}(\mathcal{R}; \mathcal{T}, \tau) - \frac{\log(x)}{\beta}\right)
\end{aligned}
\tag{3.5}
$$

Chapter 4

Fixed-Point Code Sequences

In this chapter, we consider the application of STOKE to the first of an increasingly complex set of application domains. The optimization of short sequences of loop-free, fixed-point assembly code sequences is a common problem in high-performance computing. However, the competing constraints of transformation correctness and performance improvement often force even special purpose compilers to produce sub-optimal code. Nonetheless, by combining the theory described in Chapter 2 with the implementation described in Chapter 3, it is possible to generate aggressively optimized versions of many non-trivial target code sequences. Specifically, beginning from binaries compiled by `llvm -O0`, STOKE is able to produce provably correct code sequences that either match or outperform the code produced by `gcc -O3`, `icc -O3`, and in some cases expert hand-written assembly.

Most of the discussion in this chapter focuses on the Montgomery multiplication kernel used by the OpenSSL RSA encryption library. Figure 4.1 shows two versions of this kernel, which highlight both the complexity of production `x86_64` code sequences and STOKE's ability to produce non-obvious high-performance optimizations. Beginning from code compiled by `llvm -O0` (116 lines, not shown), STOKE is able to produce code (right) that is 16 lines shorter and 1.6 times faster than the code produced by `gcc -O3` (left) and even slightly faster than the expert handwritten assembly included in the OpenSSL repository.

4.1 Fixed-Point Equality

Adapting STOKE to the domain of fixed-point computation requires a formal definition of eq$(\cdot)$ constraint described in Chapter 2. Moreover, as described in Chapter 3, this definition should be split into fast and slow variants to guarantee that STOKE is able to sustain a sufficiently high proposal throughput.

```
[r8:rdi] = rsi * [ecx:edx] + r8 + rdi
```

```
# gcc -O3

movq rsi, r9
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
imulq rdx, r9
imulq rsi, rdx
imulq rsi, rcx
addq rdx, rax
jae .L0
movabsq 0x100000000, rdx
addq rdx, rcx
.L0:
movq rax, rsi
movq rax, rdx
shrq 32, rsi
salq 32, rdx
addq rsi, rcx
addq r9, rdx
adcq 0, rcx
addq r8, rdx
adcq 0, rcx
addq rdi, rdx
adcq 0, rcx
movq rcx, r8
movq rdx, rdi
```

```
# STOKE

shlq 32, rcx
movl edx, edx
xorq rdx, rcx
movq rcx, rax
mulq rsi
addq r8, rdi
adcq 0, rdx
addq rdi, rax
adcq 0, rdx
movq rdx, r8
movq rax, rdi
```

Figure 4.1: Montgomery multiplication kernel from the OpenSSL RSA library. Compilations shown for `gcc -O3` (left) and a STOKE (right).

	al	**bl**	**cl**	**dl**
$\sigma(\mathcal{T}, t, \cdot)$	1111	0000	0000	0000
$\sigma(\mathcal{R}, t, \cdot)$	0000	1000	1100	1111
$\delta = \sigma(\mathcal{T}, t, \texttt{al}) \oplus \sigma(\mathcal{R}, t, \cdot)$	1111	0111	0011	0000
$\text{popcnt}(\delta)$	4	3	2	0
$w_{\text{mis}} \cdot \mathbf{1}(\texttt{al} \neq \cdot)$	0	1	1	1

$$\begin{aligned}\text{reg}(\mathcal{R}; \mathcal{T}, \tau) &= 4\\ \text{reg}'(\mathcal{R}; \mathcal{T}, \tau) &= \min(4, 3+1, 2+1, 1)\\ &= 1\end{aligned}$$

Figure 4.2: Strict versus relaxed equality functions for a target in which `ax` is live out and the correct result appears in an incorrect location.

A natural choice for the implementation of the $\text{eq}_{\text{slow}}(\cdot)$ constraint is the use of a *symbolic model checker* (validate($\cdot$)) [12] and a *binary indicator function* ($\mathbf{1}(\cdot)$) that returns one if its argument is true and zero otherwise. Stoke uses a sound procedure to validate the equality of loop-free code sequences [6]: both target and rewrite are converted into Smt formulae in the quantifier free theory of bit-vector arithmetic used by Z3 [36], producing a query that asks whether both sequences produce the same side effects on live outputs when executed from the same initial machine state. Depending on type, registers are modeled as between 8- and 256-bit vectors and memory is modeled as two vectors: a 64-bit address and an 8-bit value (`x86_64` is byte addressable).

$$\text{eq}_{\text{slow}}(\mathcal{R}; \mathcal{T}) = 1 - \mathbf{1}(\text{validate}(\mathcal{T}, \mathcal{R})) \tag{4.1}$$

Stoke asserts the constraint that both sequences agree on initial machine state with respect to the live inputs defined by the target. For each instruction in the target, it also asserts a constraint that encodes the transformation it represents on a machine state and chains these together to produce a constraint that describes the state of the live outputs defined by the target. An analogous set of constraints are asserted for the rewrite, and for all pairs of memory accesses at addresses $\texttt{addr}_1$ and $\texttt{addr}_2$, Stoke adds an additional constraint that relates their values: $\texttt{addr}_1 = \texttt{addr}_2 \Rightarrow \texttt{val}_1 = \texttt{val}_2$.

Using these constraints, Stoke asks Z3 if there exists an initial machine state that causes the two sequences to produce different results. If the answer is "no", then the sequences are determined to be equal. If the answer is "yes", then Z3 produces a witness that can be added to the set of test cases used by the $\text{eq}_{\text{fast}}(\cdot)$ constraint described below. Although doing so changes the search space defined by the cost function, in practice the number of failed validations that are required to produce a robust set of test cases that can be used to accurately predict correctness is quite low.

Stoke makes two simplifying assumptions to keep runtimes tractable. It assumes that stack addresses are represented exclusively as constant offsets from `rsp`. This allows stack locations to be treated as named variables in `llvm -O0` code, which exhibits heavy stack traffic. Additionally,

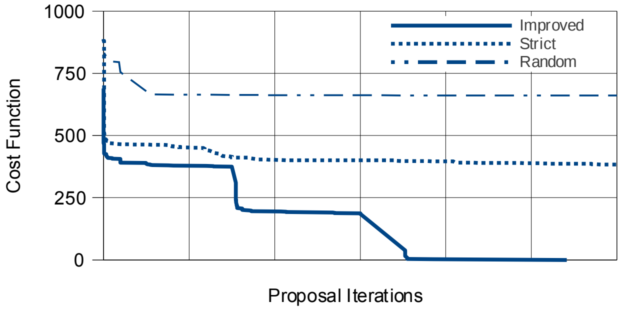


Figure 4.3: Strict versus relaxed cost functions during synthesis for the Montgomery multiplication benchmark. Random search results shown for reference.

it treats 64-bit multiplication and division as uninterpreted functions which are constrained by a small set of special-purpose axioms. Whereas Z3 diverges when reasoning about two or more such operations, the benchmarks that we consider in this chapter can contain up to four per sequence.

While the effectiveness of a model checker in this domain is encouraging, the total number of invocations that can be performed per second using current symbolic validator technology is quite low. For even modestly sized code sequences, it still well below 100. This observation motivates a dramatically different definition of the $\mathrm{eq}_{\mathrm{fast}}(\cdot)$ constraint which is based on the evaluation of test cases. Intuitively, we execute a proposed rewrite on a set of inputs and measure "how close" the output matches the target for those same inputs by counting the number of bits that differ between live outputs (i.e., the Hamming distance). In addition to being much faster than using a theorem prover, this approximation of equivalence has the added advantage of producing a smoother landscape than the 0/1 output of a symbolic equality test; it provides a useful notion of "almost correct" that can be used to help guide STOKE's search procedure.

$$\begin{aligned} \mathrm{eq}_{\mathrm{fast}}(\mathcal{R};\mathcal{T},\tau) &= \bigoplus_{t\in\tau} \mathrm{reg}(\mathcal{R};\mathcal{T},t) + \mathrm{mem}(\mathcal{R};\mathcal{T},t) + \mathrm{err}(\mathcal{R};\mathcal{T},t) \\ \text{where } \bigoplus &= \sum(\cdot) \end{aligned} \tag{4.2}$$

Recall that $\ell(\cdot)$ are the live outputs of a code sequence and that $\sigma(\cdot)$ are the side effects produced by evaluating a code sequence on a test case. In Equation 4.2, $\mathrm{reg}(\cdot)$ is used to compare the side effects that both target and rewrite produce on the *live register outputs* ($\rho(\cdot)$) defined by the target. These outputs can include general purpose, SSE, and condition registers; $\mathrm{reg}(\cdot)$ simply computes the number of bits that the results differ by using the *population count function* ($\mathrm{popcnt}(\cdot)$) which returns the number of 1-bits in the 64-bit representation of an integer.

$$\mathrm{reg}(\mathcal{R};\mathcal{T},t) = \sum_{r\in\rho(\mathcal{T})} \mathrm{popcnt}(\sigma(\mathcal{T},t,r)\oplus\sigma(\mathcal{R},t,r)) \tag{4.3}$$

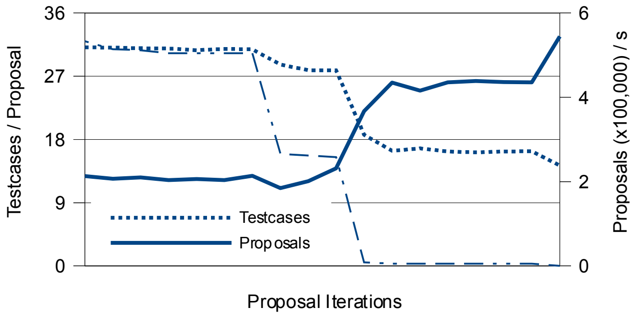


Figure 4.4: Proposals evaluated per second versus test cases evaluated prior to early termination, for the Montgomery multiplication synthesis benchmark. Cost function shown for reference.

For brevity, we omit the definition of mem$(\cdot)$ which is analogous. The remaining term err$(\cdot)$ is used to distinguish code sequences that exhibit undefined behavior by counting and then penalizing the number of segfaults, floating-point exceptions, and reads from undefined memory or registers, that occur during execution of a rewrite. We note that sigsegv$(\cdot)$ is defined in terms of the target which determines the set of addresses that may be successfully dereferenced by a rewrite for a particular test case. As described in Chapter 3, rewrites must be run in a sandbox to ensure that this behavior can be detected safely at runtime. The extension to additional types of exceptions is straightforward.

$$\begin{aligned} \mathrm{err}(\mathcal{R};\mathcal{T},t) &= w_{ss} \cdot \mathrm{sigsegv}(\mathcal{R};\mathcal{T},t) \\ &+ w_{sf} \cdot \mathrm{sigfpe}(\mathcal{R};t) \\ &+ w_{ur} \cdot \mathrm{undef}((\mathcal{R};t) \end{aligned} \tag{4.4}$$

Although the definition given above is intuitive, it is not as effective at guiding STOKE towards correct rewrites as it could be. An important improvement stems from the observation that the definition of reg$(\cdot)$ in Equation 4.3 is unnecessarily strict. An example is shown in Figure 4.2 for a target in which register `al` is live out. A rewrite that produces the inverse of the desired value in `al` is assigned the maximum possible cost in spite of the fact that it produces the correct value, only in the incorrect location: `dl`. We can improve this definition by rewarding rewrites that produce correct (or nearly correct) values in incorrect locations. The relaxed definition shown below examines all registers of equivalent *bit-width* (bw$(\cdot)$), selects the one that most closely matches the value of the target register, and assigns a small *misalignment penalty* (w_{mis}) if the selected register differs from the original. Using this definition, the rewrite is assigned a cost of just w_{mis}.

$$\begin{aligned} \mathrm{reg}'(\mathcal{R};\mathcal{T},\tau) &= \sum_{r\in\rho(\mathcal{T})} \min_{r'\in\mathrm{bw}(r)} R(\mathcal{T},\mathcal{R},r,r') \\ \text{where } R(\mathcal{T},\mathcal{R},r,r') &= \mathrm{popcnt}(\sigma(\mathcal{T},t,r)\oplus\sigma(\mathcal{R},t,r')) + w_{\mathrm{mis}}\cdot\mathbf{1}(r\neq r') \end{aligned} \tag{4.5}$$

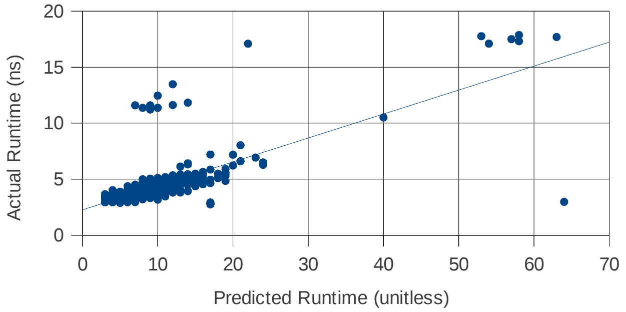


Figure 4.5: Predicted versus observed runtimes for selected code sequences. Outliers are characterized by instruction level parallelism and memory effects.

Although it is possible to relax the definition of memory equality analogously, the time required to compute this term grows quadratically with the size of the target's memory footprint. While this approach suffices for the experiments described in this and the following chapters, a more efficient implementation is necessary for more complex code sequences. Figure 4.3 shows the result of using these improved definitions during synthesis for the Montgomery Multiplication benchmark. In the amount of time required for the relaxed cost function to converge, the original strict version obtains a minimum cost that is only slightly superior to a purely random search. The dramatic improvement can be explained as an implicit parallelization of the search procedure. Accepting correct values in arbitrary locations allows Stoke to simultaneously explore as many alternate computations as can fit within a fixed length code sequence.

Because our implementation of $\mathrm{eq}_{\mathrm{fast}}(\cdot)$ is defined in terms of a monotonic reduction operator, it is also possible to use the early termination optimization described in Chapter 3 for short-circuiting test case evaluations. Figure 4.4 shows the result of applying this optimization during synthesis for the Montgomery multiplication benchmark. As the value of the cost function decreases, so too does the average number of test cases that must be evaluated prior to early termination. This produces a considerable increase in the number of proposals evaluated per second, which at peak falls between 100,000 and 1 million.

4.2 Performance Estimation

Similar considerations to the ones described above apply to the implementation of the $\mathrm{perf}_{\mathrm{fast}}(\cdot)$ term. Although it might seem natural to simply JIT compile both target and rewrite and compare runtimes, the amount of time required to execute a code sequence sufficiently many times to eliminate transient machine effects is prohibitively expensive. Moreover, natively executing randomly generated rewrites requires that they be sandboxed. As a result, any performance estimates obtained in this fashion would reflect the performance of the instrumented code rather than the code that we were interested in evaluating.

To account for this, we define a simple heuristic for approximating runtime which is based on a static approximation of the *average latencies* (lat($\cdot$)) of instructions. Although we do not consider code sequences that contain loops in this chapter, we note that the definition can be extended to loops by simply penalizing instruction within basic blocks at *loop nesting depth* (nd($\cdot$)) greater than zero by a penalty term w_{nest}. We will revisit this definition in subsequent chapters, As a result, we assign it the distinguished identifier $\lambda(\cdot)$.

$$\text{soft}_{\text{fast}}(\mathcal{R}) = \lambda(\mathcal{R}) = \sum_{b \in \mathcal{R}} \sum_{\text{i} \in b} w_{\text{nest}}^{\text{nd}(b)} \cdot E\Big[\text{lat}(i)\Big] \tag{4.6}$$

Figure 4.5 shows a reasonable correlation between this heuristic and actual runtimes for a representative corpus of code sequences. Outliers are characterized either by disproportionately high instruction level parallelism at the micro-op level or inconsistent memory access times. A more accurate model of the higher order performance effects introduced by a modern Cisc processor is feasible if tedious to construct and would likely be necessary for more complex code sequences. Regardless, the approximation is sufficient for the benchmarks that we consider in this and subsequent chapters.

Errors that result from this fast approximation are addressed by defining the $\text{perf}_{\text{slow}}(\cdot)$ constraint in terms of an uninstrumented JIT compilation method that *benchmarks* (time($\cdot$)) rewrites on a representatively large set of test cases. Recall from Chapter 3 that it is safe to use uninstrumented code sequences in this definition because the $\text{perf}_{\text{slow}}(\cdot)$ constraint is only computed for code sequences that have a zero $\text{eq}_{\text{slow}}(\cdot)$ cost. We will revisit this definition in subsequent chapters as well and assign it the distinguished identifier $\Lambda(\cdot)$.

$$\text{soft}_{\text{slow}}(\mathcal{R}, \tau) = \Lambda(\mathcal{R}, \tau) = \sum_{t \in \tau} \text{time}(\mathcal{R}, t) \tag{4.7}$$

4.3 Experiments

We evaluated our fixed-point implementation of Stoke on benchmarks drawn from both the literature and high-performance codes. For each benchmark we used 32 random test cases and a uniform proposal distribution. Four Stoke search threads were run in synthesis mode with a computational budget of 15 minutes, and for each zero cost rewrite that was discovered, a second search thread was run in optimization mode for an additional 15 minutes. Equal weight given to the equality and performance terms, the annealing constant β was set to 1.0, and moves were proposed with equal probability. Performance improvements and runtimes (we report best results on an 8 core 3.5 GHz Intel i7-4770K) are summarized in Figure 4.9. Beginning from binaries compiled using `llvm -O0`, Stoke was able to produce rewrites that matched the performance of code produced by `gcc` and

```
int p21(int x, int a, int b, int c) {
  return ((-(x == c)) & (a ^ c)) ^
         ((-(x == a)) & (b ^ c)) ^ c;
}
```

```
# gcc -O3

movl edx, eax
xorl edx, edx
xorl ecx, eax
cmpl esi, edi
sete dl
negl edx
andl edx, eax
xorl edx, edx
xorl ecx, eax
cmpl ecx, edi
sete dl
xorl ecx, esi
negl edx
andl esi, edx
xorl edx, eax
```

```
# STOKE

cmpl edi, ecx
cmovel esi, ecx
xorl edi, esi
cmovel edx, ecx
movq rcx, rax
```

Figure 4.6: Cycling Through 3 Values benchmark.

`icc` (the two produce essentially identical results). In several cases, the rewrites were comparable in performance to handwritten assembly.

Hacker's Delight [102] — often referred to as "the bible of bit-twiddling hacks" — is a collection of techniques for encoding otherwise complex algorithms as small loop-free sequences of bit-manipulating instructions. Gulwani [46] cites this text as a source of benchmarks for program synthesis and optimization and identifies a 25 function benchmark that ranges in complexity from turning off the right-most bit in a word, to rounding up to the next highest power of 2 or returning the upper 32 bits from a 64-bit multiplication. For brevity, we discuss only the programs for which STOKE was able to discover an algorithmically distinct rewrite, meaning a rewrite beyond the standard semantics preserving transformations of a traditional optimizing compiler.

Figure 4.6 shows the "Cycle Through 3 Values" benchmark, which takes an input `x`, and transforms it to the next value in the sequence $\langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle$: `a` becomes `b`, `b` becomes `c`, and `c` becomes `a`. Hacker's Delight suggests that the most natural implementation of this function is a sequence of conditional assignments, but that for an ISA without conditional move intrinsics the implementation shown is cheaper than one that uses branch instructions. For `x86_64`, which has conditional move intrinsics, this is an instance of premature optimization. However, neither `gcc` nor `icc` are able to detect this and are forced to transcribe the code as written. There are no sub-optimal subsequences in the resulting code; both are simply unable to reason about the semantics of the function as a whole. STOKE on the other hand, was able to rediscover the natural implementation from the 41 line `llvm -O0` binary.

In similar fashion, for machines without 64-bit instructions, the implementation that Hacker's Delight recommends for the "Compute the Higher Order Half of a 64-bit Product" multiplies two 32-bit inputs in four parts and aggregates the results. The computation resembles the Montgomery multiplication benchmark, and STOKE is able to discover a rewrite that requires a single multiplication using the appropriate 64-bit intrinsic. STOKE was also able to discover a number of typical superoptimizer rewrites. These include using the popcnt intrinsic, which counts the number of 1-bits in an integer, as an intermediate step in the "Compute Parity" and "Determine if an Integer is a Power of 2" benchmarks.

SAXPY (Single-precision Alpha X Plus Y) is a level 1 vector operation in the Basic Linear Algebra Subsystems Library [9]. The function makes heavy use of heap accesses and presents the opportunity for optimization using vector intrinsics. To give STOKE the opportunity to take advantage of this property, our implementation is unrolled four times by hand, as shown in Figure 4.7. Despite heavy annotation to indicate that the arrays pointed to by x and y are aligned and do not alias each other, the production compilers either could not detect the possibility of a compilation using vector intrinsics, or were precluded from doing so by some internal heuristic.

STOKE on the other hand was able to discover the natural implementation: the constant a is broadcast four ways from a general purpose register into an SSE register, and then multiplied by and added to the contents of x and y, which are loaded into SSE registers four elements at a time. The four way broadcast does not appear anywhere in either the `gcc -O3` code, or in the original 61 line `llvm -O0` code.

```
void SAXPY(int* x, int* y, int a) {
  x[i]   = a * x[i]   + y[i];
  x[i+1] = a * x[i+1] + y[i+1];
  x[i+2] = a * x[i+2] + y[i+2];
  x[i+3] = a * x[i+3] + y[i+3];
}
```

```
# gcc -O3

movslq ecx,rcx
leaq (rsi,rcx,4),r8
leaq 1(rcx),r9
movl (r8),eax
imull edi,eax
addl (rdx,rcx,4),eax
movl eax,(r8)
leaq (rsi,r9,4),r8
movl (r8),eax
imull edi,eax
addl (rdx,r9,4),eax
leaq 2(rcx),r9
addq 3,rcx
movl eax,(r8)
leaq (rsi,r9,4),r8
movl (r8),eax
imull edi,eax
addl (rdx,r9,4),eax
movl eax,(r8)
leaq (rsi,rcx,4),rax
imull (rax),edi
addl (rdx,rcx,4),edi
movl edi,(rax)
```

```
# STOKE

movd edi,xmm0
shufps 0,xmm0,xmm0
movups (rsi,rcx,4),xmm1
pmullw xmm1,xmm0
movups (rdx,rcx,4),xmm1
paddw xmm1,xmm0
movups xmm0,(rsi,rcx,4)
```

Figure 4.7: SAXPY benchmark.

```
while (head != 0) {
  head->val *= 2;
  head = head->next;
}
```

```
# gcc -O3

movq -8(rsp), rdi
.L1:
sall (rdi)
movq 8(rdi), rdi
.L2:
testq rdi, rdi
jne .L1
```

```
# STOKE

.L1:
movq -8(rsp), rdi
sall (rdi)
movq 8(rdi), rdi
movq rdi, -8(rsp)
.L2:
movq -8(rsp), rdi
testq rdi, rdi
jne .L1
```

Figure 4.8: Linked List Traversal benchmark.

Figure 4.8 shows the linked-list traversal benchmark from [6]. The code iterates over a list of integers and multiplies each of the elements by two. The code is unique with respect to the benchmarks discussed so far in that it contains a loop. As a result, STOKE is unable to optimize the function as a whole, and must focus only on its inner-most loop-free fragment — we will discuss solutions to this limitation in Chapter 6. STOKE discovers the same transformations as the optimizer described in [6]: the elimination of stack traffic and a strength reduction from multiplication to bit shifting. However it fails to eliminate the instructions that copy the head pointer from, and back to, the stack on every iteration of the loop. In contrast, both production compilers were able to eliminate the memory traffic by caching the pointer in a register prior to entering the loop. Unsurprisingly, the rewrite discovered by STOKE is slower.

Finally, as shown in Figure 4.9, STOKE was unable to synthesize a rewrite for three of the Hacker's Delight Benchmarks. All three benchmarks, despite being quite different, have the same interesting property that they produce results that differ by only a single bit from a simple yet completely incorrect alternative. The "Round Up to the Next Highest Power of 2" benchmark is nearly indistinguishable from the function that always returns zero. The same is true of the "Next Highest with Same Number of 1-bits", and a small transformation to the "Exchanging Two Fields" benchmark with respect to the identity function. Using its optimization routine alone STOKE was still able to discover rewrites that performed comparably to the code produced by the production compilers. Nonetheless, we do not expect this to be the case in general. For benchmarks of this form, a more sophisticated cost function is surely necessary.

	Speedup (×100%)		Runtime (s)	
	gcc/icc -03	STOKE	Synth.	Opt.
p01	1.60	1.60	0.15	3.05
p02	1.60	1.60	0.16	3.14
p03	1.60	1.60	0.34	3.45
p04	1.60	1.60	2.33	3.55
p05	1.60	1.60	0.47	3.24
p06	1.60	1.60	1.57	6.26
p07	2.00	2.00	1.34	3.10
p08	2.20	2.20	0.63	3.24
p09	1.20	1.20	0.26	3.21
p10	1.80	1.80	7.49	3.61
p11	1.50	1.50	0.87	3.05
p12	1.50	1.50	5.29	3.34
p13	3.25	3.25	0.22	3.08
p14	1.86	1.86	1.43	3.07
p15	2.14	2.14	2.83	3.17
p16	1.80	1.80	6.86	4.62
p17	2.60	2.60	10.65	4.45
p18	2.44	2.50	0.30	4.04
p19	1.93	1.97	-	18.37
p20	1.78	1.78	-	36.72
p21	1.62	1.65	6.97	4.96
p22	3.38	3.41	0.02	4.02
p23	5.53	6.32	0.13	4.36
p24	4.67	4.47	-	48.90
p25	2.17	2.34	3.29	4.43
mont mul	2.84	4.54	319.03	111.64
linked list	1.10	1.09	3.94	8.08
SAXPY	1.82	2.46	10.35	6.66

Figure 4.9: Speedups over `llvm -O0` versus Stoke runtimes. Benchmarks for which an algorithmically distinct rewrite was discovered are shown in bold; synthesis timeouts are annotated with a −.

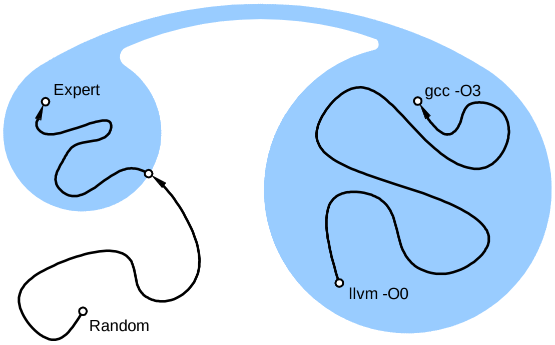


Figure 4.10: Search space for the Montgomery multiplication benchmark: `O0` and `O3` codes are densely connected, whereas expert code is reachable only by an extremely low probability path.

4.4 Discussion

An early version of our fixed-point implementation of Stoke was able to transform `llvm -O0` code into the equivalent of `gcc -O3` code, but was unable to produce results that were competitive with expert hand-written code. The reason is suggested by Figure 4.10, which abstractly depicts the search space for the Montgomery multiplication benchmark shown in Figure 4.1. For loop-free code sequences, `llvm -O0` and `gcc -O3` differ primarily in stack use and instruction selection, but otherwise produce algorithmically similar results. Compilers are generally designed to compose many small local transformations: dead code elimination deletes an instruction, constant propagation changes a register to an immediate, and strength reduction replaces a multiplication by an add. Sequences of local optimizations such as these correspond to regions of equivalent code sequences that are densely connected by very short sequences of moves (often just one) and easily traversed by local search methods. Beginning from `llvm -O0` code, Mcmc sampling will quickly improve local inefficiencies one by one and hill climb its way to the equivalent of `gcc -O3` code.

The code discovered by Stoke occupies an entirely different region of the search space; it represents a completely distinct algorithm for implementing the kernel at the assembly level. The only method for a local search procedure to produce code of this form from compiler generated code is to traverse the extremely low probability path that builds the code in place next to the original (all the while increasing its cost) only to delete the original code at the very end.

Although Mcmc sampling is guaranteed to traverse this path in the limit, the likelihood of it doing so in any reasonable amount of time is so low as to be useless in practice. It was this observation that motivated the separation of Stoke's cost minimization routine into two separate phases:

- A **synthesis** phase focused solely on correctness, which attempts to locate regions of equivalent code sequences that are distinct from the region occupied by the target.

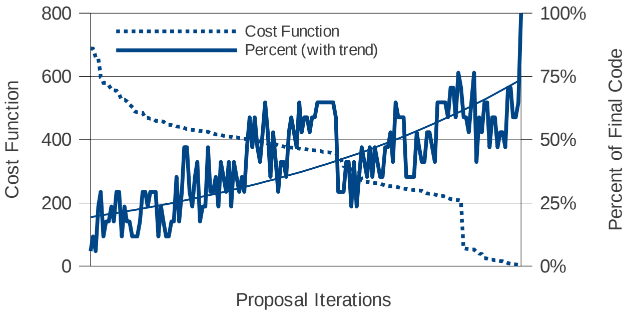


Figure 4.11: Cost over time versus percentage of instructions that appear in the final zero-cost rewrite for the Montgomery multiplication synthesis benchmark.

- An **optimization** phase focused on performance, which searches for the fastest sequence within each of those regions.

Even given this insight, it is still perhaps unintuitive that synthesis should be able to produce a correct rewrite from such an enormous search space in a tractable amount of time. In our experience, synthesis is effective precisely when it is possible to discover portions of a correct rewrite incrementally, rather than all at once. Figure 4.11 compares cost over time against the percentage of instructions that appear in the final rewrite for the Montgomery multiplication benchmark. As synthesis proceeds, the percentage of correct code increases in inverse proportion to the value of the cost function. While this is encouraging and there are many code sequences that can be synthesized in pieces, there are many that can not. In the limit, any complex computation that produces a single boolean value will pose a serious problem. Fortunately, even when synthesis fails, optimization is still possible. It must simply proceed only from the region occupied by the target as a starting point.

In this chapter, we showed a new approach to improving the runtime performance of loop-free fixed-point code sequences which formulates program optimization as a stochastic search problem. Compared to a traditional compiler, which factors optimization into a sequence of small independently solvable subproblems, our framework is based on cost minimization and considers the competing constraints of transformation correctness and performance improvement simultaneously. We showed that an Mcmc sampler can be used to rapidly explore cost functions of this form and produce low cost samples which correspond to high quality optimizations. Although our method sacrifices completeness, the scope of programs which we are able to consider, and the quality of the rewrites we produce, far exceed those of preexisting techniques.

Although our fixed-point implementation of Stoke is in many cases able to produce rewrites that are competitive with or outperform the code produced by production compilers, there remains substantial room for improvement. Future work will certainly involve the development of cost functions that are robust against targets with numerous deceptively attractive — albeit completely incorrect — synthesis alternatives.

4.5 Related Work

Although techniques that preserve completeness are effective within certain domains, their general applicability remains limited. The shortcomings are best highlighted in the context of the Montgommery Multiplication code sequence shown at the beginning of this chapter. The code does the following: two 32-bit values, `ecx` and `edx`, are concatenated and multiplied by the 64-bit value `rsi` to produce a 128-bit product. The 64-bit values in `rdi` and `r8` are added to that product, and the result is stored in `r8` and `rdi`. The version produced by `gcc -O3` (left) implements the 128-bit multiplication as four 64-bit multiplications and a summation of the results. In contrast, the version produced by Stoke (right), uses a hardware intrinsic which requires that the inputs be permuted and moved to distinguished register locations so that the multiplication may be performed in a single step. The odd looking move on line 4 (right) produces the non-obvious but necessary side effect of zeroing the upper 32 bits of `rdx`.

Massalin's superoptimizer [70] explicitly enumerates sequences of code of increasing length and selects the first that behaves identically to the input sequence on a set of test cases. Massalin reports optimizing instruction sequences of up to length 12, but only after restricting the set of enumerable opcodes to between 10 and 15. In contrast, Stoke uses a large subset of the nearly 400 `x86_64` opcodes, some with over 20 variations, to produce the 11 instruction kernel shown in Figure 4.1. It is unlikely that Massalin's approach would scale to an instruction set of this size.

Denali [57], and Equality Saturation [97], achieve improved scalability by only considering code sequences that are equivalent to the input sequence; candidates are explored through successive application of equality preserving transformations. Because both techniques are goal-directed, they dramatically improve the number of primitive instructions and the length of sequences that can be considered in practice. However, both also rely heavily on expert knowledge. It is unclear whether an expert would know to encode the multiplication transformation shown in Figure 4.1, or whether a set of expert rules could ever cover the set of all possible interesting optimizations.

Bansal's peephole superoptimizer [6] automatically enumerates 32-bit `x86` optimizations and stores the results in a database for later use. By exploiting symmetries between code sequences that are equivalent up to register renaming, Bansal was able to scale this approach to optimizations mapping code sequences of up to length 6 to sequences of up to length 3. The approach has the dual benefit of hiding the high cost of superoptimization by performing search once-and-for-all offline and eliminating the dependence on expert knowledge. To an extent, the use of a database also allows the system to overcome the low upper bound on instruction length through the repeated application of the optimizer along a sliding code window. Regardless, the kernel shown in Figure 4.1 has a property shared by many real world code sequences that no sequence of local optimizations will transform the code produced by `gcc -O3` into the code produced by Stoke.

Finally, Sketching [93] and Brahma [46] address the closely related component-based sequence synthesis problem. These systems rely on either a declarative specification, or a user-specified partial sequence, and operate on statements in simplified bit-vector calculi rather than directly on hardware instructions. Liang [64] considers the task of learning code sequences from test cases alone, but at a similarly high level of abstraction. Although useful for synthesizing non-trivial code, the internal representations used by these systems preclude them from reasoning about the low-level performance properties of the code that they produce.

Chapter 5

Floating-Point Code Sequences

In this chapter we extend the application of STOKE to floating-point computations. The aggressive optimization of floating-point code sequences is another important problem in high-performance computing. Nonetheless, both programming languages and compilers are limited in their ability to produce such optimizations, which in many cases can only be obtained by carefully sacrificing precision. Because most programming languages lack the appropriate mechanisms for a programmer to communicate precision requirements to the compiler there is often little opportunity for optimization.

In most cases a programmer's only mechanism for communicating precision requirements to a compiler is through the choice of data-type (e.g. `float` or `double`). This is at best a coarse approximation and at worst wholly inadequate for describing the subtleties of many high-performance numeric computations. Consider the task of building a customized implementation of the exponential function, which must be correct only to 48-bits of precision and defined only for positive inputs less than 100. An expert could certainly craft this kernel at the assembly level, however the process is well beyond the abilities of the average programmer.

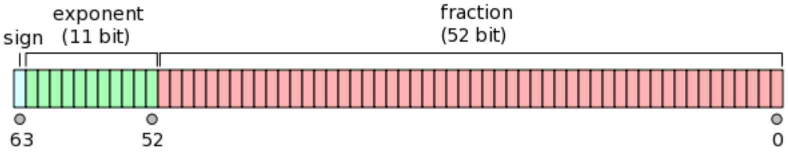
In this chapter, we show that by using STOKE we are able to generate custom implementations of the trigonometric and exponential kernels in Intel's handwritten implementation of the C numeric library `math.h`, which are specialized to between 1- and 64-bits of floating-point precision, and are up to 6 times faster than the original code. Additionally, we show that for real world programs such as a massively parallel direct numeric simulation of heat transfer and a ray tracer, we are able to obtain 30% *full-program* speedups by aggressively optimizing floating-point kernels that can tolerate a loss of precision while retaining end-to-end correctness.

The aggressive nature of the optimizations produced by STOKE creates a difficulty in checking the correctness of the resulting code sequences. There are two possible approaches using currently known static verification techniques. However, neither is capable of formally verifying the kernels that arise in our applications. Existing decision procedures for floating-point arithmetic that are based on bit-blasting could in principle be used to prove equivalence of an original and an optimized code within a specified error tolerance. However in practice these techniques can only handle instruction sequences on the order of five lines long, which is two orders of magnitude too small for our benchmarks. Abstract interpretation is the alternative, but no static analysis has even attempted to deal with the complexity of high-performance floating-point kernels. In particular, no existing analysis is capable of reasoning about mixed floating- and fixed-point code, yet many floating-point computations include non-trivial sections of fixed-point computation that affect floating-point outputs in non-obvious ways.

Randomized testing is another possible approach, but the guarantees that it offers are only empirical. Although passing any reasonable set of test cases — random or not — is encouraging and for many real code bases represents a de facto definition of correctness, random testing offers no formal guarantees beyond the inputs used in those tests. Several researchers have attempted to give statistical guarantees based on the absence of errors following the observation of a very large number of tests. However, in the absence of a formal characterization of the distribution of errors relative to program inputs these guarantees are statistically unsound and offer no stronger guarantees than plain random testing.

To address this issue, this chapter shows a novel randomized method that does not suffer from these shortcomings and can be used to establish strong evidence for the correctness of floating-point optimizations. We treat the difference in outputs produced by a floating-point kernel, f, and an optimization, f' as an error function, $E(x) = |f(x) - f'(x)|$, and use a robust randomized search technique to attempt to find the largest value of $E(y)$ for some input y. In the limit the search is theoretically guaranteed to find the maximum value in the range of E. However, well before that limiting behavior is observed we show that it is possible to use a statistical measure to establish confidence that the search has converged and that an optimization is correct to within the specified error tolerance. Borrowing a term from the computational science community, we refer to this as a technique for *validating* optimizations to distinguish it from the stricter standard of formal verification. Although our technique provides evidence of correctness, we stress that it does not guarantee it.

Using this technique, we are able to establish upper bounds on the imprecision of an optimization that are tighter than those produced by either sound decision procedures or abstract interpretation techniques for benchmarks where these techniques are applicable. For benchmarks not amenable to either static technique we are able to produce upper bounds that either cannot be refuted or are within a small margin of those exposed by random testing that is orders of magnitude more intensive than the effort expended on validation. Although there exist optimizations that we do not expect this technique to perform adequately on we believe that the results we obtain represent the potential for a considerable improvement over current practice.



Type	exponent (e)	fraction (f)	Value
Zero	0	0	$(-1)^s \cdot 0$
Denormal	0	$\neq 0$	$(-1)^s \cdot 2^{-1022} \cdot 0.\text{f}$
Normal	$[1, 2046]$	unconstrained	$(-1)^s \cdot 2^{e-1023} \cdot 1.\text{f}$
Infinity	2047	0	$(-1)^s \cdot \infty$
NaN	2047	$\neq 0$	$(-1)^s \cdot \perp$

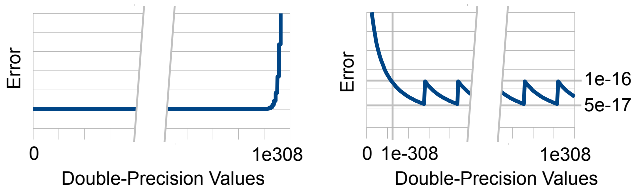
Figure 5.1: IEEE-754 double-precision floating-point standard.

5.1 Floating-Point Equality

The primary complication in adapting STOKE to floating-point programs is identifying an appropriate notion of correctness. The floating-point design goal of being able to represent both extremely large and small values in a compact bit-wise representation is fundamentally at odds with the ability to maintain uniform precision within that range.

Figure 5.1 shows the IEEE-754 standard for double-precision floating-point values. The standard is capable of representing magnitudes between 10^{-324} and 10^{308} as well as symbolic constants such as *infinity* and *not a number*, but cannot precisely represent the value 0.1. Worse, values are distributed extremely non-uniformly, with exactly half of all floating-point values located between -1.0 and 1.0. The inability to precisely represent real values is further complicated by the rounding error introduced by basic arithmetic operations. Most calculations on floating-point numbers produce results that must be rounded back to a representable value. Furthermore, many operations are implemented in a way that requires their operands to be normalized to the same exponent. For operands that vary greatly in magnitude, many digits of precision are discarded in the normalization process. Although the IEEE standard places strict bounds on these errors they are nonetheless quite difficult to reason about as even basic arithmetic identities such as associativity do not generally hold. As a result, floating-point optimizers are often forced to preserve programs as written and abandon aggressive optimization.

Our method for reasoning about correctness in the presence of these complications is the following. Given two floating-point programs, we say that an optimization is *correct* if the results it produces are all within a rounding error η of the result placed in the corresponding location by the original program. Two simple methods for representing rounding error are the absolute and relative error functions. These functions are defined over the real numbers $\mathbb{R}$, of which the representable



(a) Absolute Error (b) Relative Error

Figure 5.2: Error functions computed for adjacent double-precision values. Absolute error diverges for large inputs; relative error for sub-normal inputs. Neither are defined for infinity or NaN.

floating-point numbers $\mathbb{F}$ are a subset.

$$\begin{aligned} \mathrm{abs}(r_1, r_2) &: \mathbb{R} \times \mathbb{R} \to \mathbb{R} = |r_1 - r_2| \\ \mathrm{rel}(r_1, r_2) &: \mathbb{R} \times \mathbb{R} \to \mathbb{R} = \left|\frac{r_1 - r_2}{r_1}\right| \end{aligned} \tag{5.1}$$

It turns out however that there are serious problems with using either absolute or relative error. Figure 5.2 shows the results obtained when abs(·) is computed over adjacent floating-point values; errors between large values are weighted more heavily than errors between small values. A similar phenomenon occurs for the relative error function, which diverges for denormal and zero values. In both cases, these inconsistencies are only magnified when applied to non-adjacent values. For these reasons, rounding error is typically measured in terms of *uncertainty in the last place* (ULPs), which measures the difference between a real number and the closest representable floating point value.

$$\begin{gathered} \mathrm{ULP}(f, r) : \mathbb{F} \times \mathbb{R} \to \mathbb{R} = \left| d_1.d_2 \ldots d_p - \frac{r}{\beta^e} \right| \beta^{p-1} \\ \text{where } f \equiv d_1.d_2 \ldots d_p \cdot \beta^e \end{gathered} \tag{5.2}$$

Compared to absolute and relative error, this measure has the advantage of representing error uniformly across the entire range of representable floating-point values (including infinity and NaNs). Furthermore, it can be shown [100] that for normal values the relationship between ULPs and relative error is well-behaved and follows the relation shown below where the upper and lower bounds correspond to the maximum and minimum values in the right half of Figure 5.2(b). Intuitively, ULPs can be thought of as a uniform measure of rounding error which for most representable floating-point inputs is an approximation of relative error that is correct to within an order of magnitude.

$$\forall r \exists f. \left(\frac{1}{2}\beta^{-p} \leq \frac{1}{2}\mathrm{ULP}(f, r) \leq \frac{\beta}{2}\beta^{-p} \right) \tag{5.3}$$

We use this definition of rounding error to extend the fast component of STOKE's equality term as follows. Recall that STOKE defines test case error with respect to the weighted sum of three terms that represent errors appearing in registers, errors appearing in memory, and divergent signal behavior. Our extension is parameterized by a new user-defined constant η which defines the minimum unacceptable ULP rounding error.

$$\begin{aligned} \mathrm{eq}_{\mathrm{fast}}(\mathcal{R};\mathcal{T},t,\eta) \; &= w_{\mathrm{reg}} \cdot \mathrm{reg}(\mathcal{R};\mathcal{T},t,\eta) \; + \\ &\quad\; w_{\mathrm{mem}} \cdot \mathrm{mem}(\mathcal{R};\mathcal{T},t,\eta) \; + \\ &\quad\; w_{\mathrm{sig}} \cdot \mathrm{sig}(\mathcal{R};\mathcal{T},t) \end{aligned} \tag{5.4}$$

Of the three terms shown above we leave the last unmodified and only describe our new definition of register error. Our definition of memory error is analogous and it is a straightforward extension to combine these definitions with the original fixed-point definitions in a way that is sensitive to the type (fixed- or floating-point) of the value stored in a particular location. Recall that $\sigma(\cdot)$ is the side effect that appears in a location after evaluating a code sequence on a test case. We define the error in a register to be the difference in ULPs between the value it contains, and the value placed in that same register by the target. All values in excess of η are scaled down toward zero and values below this bound are replaced by zero.

$$\begin{aligned} \mathrm{reg}(\mathcal{R};\mathcal{T},t,\eta) &= \sum_{r \in \rho(\mathcal{T})} \max\Big(0, \mathrm{ULP}(f_{\mathcal{R}}, f_{\mathcal{T}}) - \eta\Big) \\ \text{where } f_{\mathcal{R}} &= \mathrm{val}_{\mathrm{double}}(\sigma(\mathcal{R},t,r)) \\ f_{\mathcal{T}} &= \mathrm{val}_{\mathrm{double}}(\sigma(\mathcal{T},t,r)) \end{aligned} \tag{5.5}$$

In contrast to the fixed-point implementation of Stoke described in the previous chapter, here we lift the definition of test case correctness to sets by using max$(\cdot)$ as a reduction operator in place of summation. This follows from the observation that for large test case sets the repeated summation of very large ULP errors has the potential to produce integer overflow. Using max$(\cdot)$ guarantees that regardless of the number of test cases used the value of the equality term never exceeds `ULLONG_MAX`.

$$\begin{aligned} \mathrm{eq}_{\mathrm{fast}}(\mathcal{R};\mathcal{T},\tau,\eta) &= \bigoplus_{t \in \tau} \mathrm{eq}_{\mathrm{fast}}(\mathcal{R};\mathcal{T},t,\eta) \\ \text{where } \bigoplus &= \max(\cdot) \end{aligned} \tag{5.6}$$

As with Stoke's fixed-point equality term, this function provides a useful measure of partial correctness, which has the effect of smoothing the search space and guiding Stoke gradually towards optimizations. Additionally, it also provides Stoke with a robust mechanism for ignoring errors that the user considers insignificant and not worth penalizing. Because the sig$(\cdot)$ term in Equation 5.4 is not parameterized by η this function is guaranteed to return positive values for any set of test cases that exposes divergent signal behavior.

The computation of ULP rounding error shown in Equation 5.2, which is defined in terms of the

```
uint64_t ULP(double x, double y) {
  int64_t xx = *((int64_t*)&x);
  xx = xx < 0 ? LLONG_MIN - xx : xx;

  int64_t yy = *((int64_t*)&y);
  yy = yy < 0 ? LLONG_MIN - yy : yy;

  return xx >= yy ? xx - yy : yy - xx;
}
```

Figure 5.3: Platform-dependent C code for computing ULP distance between two double-precision values. Note the reordering of negative values.

comparison between a real number and a floating-point number is unnecessarily complicated when applied to the comparison of two floating-point values. In comparing the live outputs of two programs we instead use the simpler version shown below which simply counts the number of floating-point numbers between two values. This function has the advantage of producing only integer results and greatly simplifies the program logic associated with cost manipulation.

$$\mathrm{ULP}'(f_1, f_2) : \mathbb{F} \times \mathbb{F} \to \mathbb{N} = \Big| \big\{ x \in \mathbb{F} \mid f_1 < x \leq f_2 \big\} \Big| \tag{5.7}$$

Figure 5.3 shows the `C` implementation of this function for double-precision values. The floating-point representation shown in Figure 5.1 has the interesting property that when reinterpreted as signed integers iterating from `LLONG_MIN` to 0 corresponds to iterating in descending order from negative zero to negative NaN, and iterating from 0 to `LLONG_MAX` corresponds to iterating in ascending order from positive zero to positive NaN. The comparison against `LLONG_MIN` inverts the relative ordering of negative values so that the entire set of floating-point values is arranged in ascending order and ULPs can be computed using simple signed subtraction.

We now consider the definition of the slow component of STOKE's equality term. The fixed-point implementation of STOKE described in the previous chapter defines this routine in terms of the invocation of a sound decision procedure for fixed-point operations over bit vectors. Having redefined test case correctness for floating-point programs in terms of a minimum acceptable ULP rounding error η it is straightforward to redefine the design constraints of this term as well. We say that a target and rewrite are equal if there does not exist a test case t that exposes an ULP error in excess of η and we define $\top_\eta$ to be larger than all η to ensure that divergent signal behavior is always exposed.

$$\begin{aligned}\mathrm{eq}_{\mathrm{slow}}(\mathcal{R};\mathcal{T},\eta) &= \neg\exists t.\Big(\mathrm{error}(\mathcal{R};\mathcal{T},t) > \eta\Big)\\ \text{where } \mathrm{error}(\mathcal{R};\mathcal{T},t) &= \sum_{l\in\ell(\mathcal{T})} \mathrm{ULP}(f_{\mathcal{R}}, f_{\mathcal{T}}) \;\; + \top_{\eta}\cdot \mathrm{sig}(\mathcal{R};\mathcal{T},t)\\ f_{\mathcal{R}} &= \mathrm{val}_{\mathrm{double}}(\sigma(\mathcal{R},t,r))\\ f_{\mathcal{T}} &= \mathrm{val}_{\mathrm{double}}(\sigma(\mathcal{T},t,r))\end{aligned} \tag{5.8}$$

The complexity of floating-point programs precludes the use of the most obvious implementations of this routine for many optimizations of interest. In general neither sound decision procedures nor abstract interpretation techniques are currently practical for non-trivial programs.

SMT float [85] is a new standard for decision procedures that contains support for floating-point operations however it is only now beginning to be widely adopted. Z3's implementation [26] for example, is based on bit-blasting and does not scale beyond programs containing a very small number of instructions [23]. Other approaches to deciding floating-point formulas are unable to handle code sequences that interleave fixed-point bit-vector and floating-point arithmetic [47, 54]. Because bit-wise operations such as the extraction of exponent bits are quite common in performance critical code these procedures are currently incapable of verifying many interesting `x86_64` optimizations. And although it is possible to replace floating-point instructions by uninterpreted functions this abstraction cannot bound the error between two codes that are not bit-wise equivalent. As a result, the application of symbolic execution based approaches such as [20] is limited.

The abstract domains used for floating-point reasoning in methods based on abstract interpretation such as those in [27] are also unable to handle the presence of fixed-point bit-wise operations. Furthermore, even in cases where the target and rewrite are in fact bit-wise equivalent the approximations made by these abstractions render them unable to prove bit-wise equivalence in situations that commonly arise in practice. Nonetheless, for programs that perform exclusively floating-point computations it is possible to bound — if coarsely — the absolute error between two floating-point programs [23].

Where neither of these approaches are appropriate, some work has been done in providing guarantees based on randomized testing [79]. However, in the absence of any knowledge of how errors are distributed relative to program inputs these guarantees are extremely weak. Essentially, they are no stronger than those obtained by a test suite and apply only to correctness with respect to the tests in that suite. As a simple example, consider a program $f(x)$ and an optimization $f'(x)$ for which the magnitude of the error function $E(x) = |f'(x) - f(x)|$ is distributed non-uniformly as the function $|\sin(x)|$. Pure randomized testing is unlikely to uncover the values of x for which $E(x)$ is maximized.

In contrast to the methods described above we use Mcmc sampling to draw values from the error(·) function. The idea is simple: our goal is to identify a test case t that will cause error(·) to produce a value greater than η. Although in general we cannot produce a closed form representation of error(·), sampling from it is straightforward: we simply evaluate the target and the rewrite on t and compare the results. As discussed in Chapter 2, in the limit MCMC sampling will draw test cases in proportion to the value of the error function. Not only will this expose the maximum value of the function, but it will do so more often than for any other value. Using the Metropolis-Hastings algorithm, we define a concrete implementation of Equation 5.8, which is based on the iterative evaluation of the samples taken from the sequence of proposals $t_0, t_1, \ldots, t_\infty$.

$$\mathrm{eq}_{\mathrm{slow}}(\mathcal{R};\mathcal{T},\eta) = \max\Big(\mathrm{error}(\mathcal{R};\mathcal{T},\mathrm{t_i})\Big)_{\mathrm{i=0}}^{\infty} \leq \eta \tag{5.9}$$

The proposal distribution over test cases is defined as follows. Recall that a test case contains values for each of a function's live inputs. For every test case t_i we define its successor t_{i+1} by modifying each of the live in register locations in t_i by a value sampled from a normal distribution. In generating these values we discard proposals for the value in location l which are outside the range of valid inputs $[l_{min}, l_{max}]$ specified by the user.

$$\begin{aligned} t_{i+1} &= \Big\{(l,v) \mid l \in Dom(t_i), v = v'\Big\} \\ \text{where } v' &= \begin{cases} v & t_i(l) + x < l_{min} \\ v & t_i(l) + x > l_{max} \\ t_i(l) + x & \text{otherwise} \end{cases} \\ x &\sim \mathcal{N}(\mu,\sigma) \end{aligned} \tag{5.10}$$

Discarding values outside this range is sufficient to guarantee that we never propose a test case which results in a computation that produces erroneously divergent behavior. A floating-point function might for example take two arguments: a floating-point value and a pointer to a location in which to write the result. In proposing test cases it is crucial that the latter not be modified to point to an undefined address. Discarding values outside a user-defined range also allows a user to customize the rewrites discovered by STOKE to a particular range of desired inputs. Both the ergodicity and symmetry of this proposal distribution follow from the properties of a normal distribution.

The last remaining issue is the definition of a termination condition that can be used to determine when to stop generating steps in the Markov chain. Effectively, we must define a criterion under which we can be reasonably confident that an observed sequence of samples contains the maximum value of the error function. A common method for doing so is to check whether the chain of samples has *mixed well*. Mixing well implies that the chain has reached a point where it represents a stationary distribution and contains samples that represent a uniform distribution over the test cases in the domain of the error function. Under these conditions we can be confident that the chain has sampled from the regions that contain all local maxima and as a result should contain the global maximum.

Metrics for determining whether a chain has mixed well are well known and available in many statistical computing packages [80]. We use the Geweke diagnostic test [37] which is appropriate for measuring the convergence of single chains. The Geweke diagnostic divides a chain into two windows and compares the means of the two chains which should be equal if the chain is stationary. If we define the two chains of samples as

$$\begin{gathered}\theta_1 = \{\mathrm{error}(\mathcal{R}; \mathcal{T}, \mathrm{t_i}) : \mathrm{i} = 1, \ldots, \mathrm{n_1}\} \\ \theta_2 = \{\mathrm{error}(\mathcal{R}; \mathcal{T}, \mathrm{t_i}) : \mathrm{i} = \mathrm{n_a}, \ldots, \mathrm{n}\} \\ \text{where } 1 < n_1 < n_a < n \text{ and } n_2 = n - n_a + 1\end{gathered} \tag{5.11}$$

then we can compute the following statistic, where $\hat{s}_1(0)$ and $\hat{s}_2(0)$ are spectral density estimates at zero frequency for the two chains.

$$\begin{gathered}Z_n = \frac{\bar{\theta}_1 - \bar{\theta}_2}{\sqrt{\frac{\hat{s}_1(0)}{n_1} + \frac{\hat{s}_2(0)}{n_2}}} \\ \text{where } \bar{\theta}_1 = \frac{1}{n_1} \sum \theta_1 \text{ and } \bar{\theta}_2 = \frac{1}{n_2} \sum \theta_2\end{gathered} \tag{5.12}$$

If the ratios n_1/n and n_2/n are fixed, $(n_1 + n_2)/n < 1$, and the chain is stationary, then Z_n will converge to a standard normal distribution as $n \to \infty$. Intuitively, computing this statistic for a poorly mixed chain will produce a large absolute value of Z_n. Should this occur we can simply continue to sample from error(·) and recompute Z_n as necessary. Once Z_n achieves a sufficiently small value we can conclude that the chain is approximately equal to a stationary distribution and return the largest observed sample as a bound on ULP rounding error between the target and rewrite.

We call this Mcmc-based randomized testing method *validation* to distinguish it from formal verification. While our method comes with a mathematical — if asymptotic — guarantee and provides strong evidence of correctness we stress that this is not the same level of assurance that formal verification provides. In particular, the sig(·) term in Equation 5.8 has the potential to introduce discontinuities that may be very difficult to discover. Thus $\mathrm{eq}_{\mathrm{slow}}(\cdot)$ is not an appropriate test of correctness for say, optimizations applied to safety-critical code. Nonetheless, for many real world performance-critical applications where correctness is already defined entirely with respect to confidence in the compiler writer and behavior on regression test suites this definition represents a considerable improvement over current practice.

5.2 Experiments

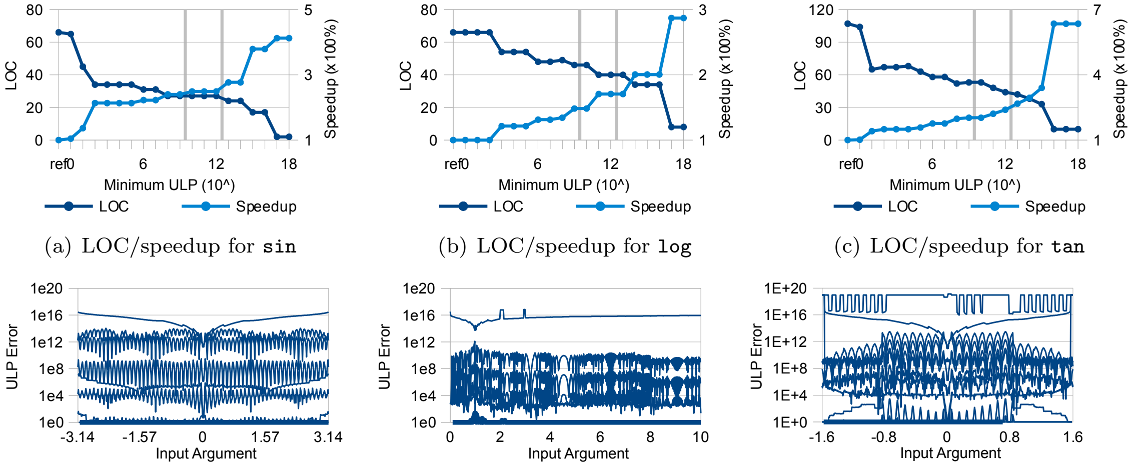
We evaluated our implementation of Stoke with support for floating-point optimizations on three high performance benchmarks: Intel's implementation of the `C` numerics library `math.h`, a three dimensional direct numeric simulation solver for HCCI combustion, and a full featured ray tracer. Each benchmark contains a mixture of kernels some of which require bit-wise correctness and some of which can tolerate a loss of precision while still being able to produce useful results. As we demonstrate below a large performance improvement can be obtained by aggressively optimizing the latter. We close with a case study that compares the Mcmc search kernel used by Stoke against several alternate implementations.

All of the experiments in this chapter were run on a four core Intel i7 with support for the full Haswell instruction set. For benchmarks where handwritten assembly was unavailable targets were generated using `gcc` with full optimizations enabled (`icc` produced essentially identical code). Stoke was run in optimization mode using 16 search threads, 1024 test cases, a timeout of 10 million proposals, and equal weight given to the equality and performance terms. The annealing constant β was set to 1.0, and moves were proposed with equal probability. For estimating a bound on optimization error test case modifications were proposed using the standard normal distribution $\mathcal{N}(0,1,)$.

For each benchmark Stoke search threads ran to completion in 30 minutes, and memory consumption never exceeded one gigabyte. Mcmc validation reached convergence after fewer than 100 million proposals and runtimes never exceeded one minute. Some benchmarks were amenable to formal verification; we compared the results against our validation method whenever it was possible to do so.

`libimf` is Intel's implementation of the `C` numerics library, `math.h`. It contains hand-coded implementations of the standard trigonometric, exponential, and logarithmic functions. In addition to taking advantage of many non-obvious properties of polynomial approximation and floating-point algebra these implementations are able to dynamically configure their behavior to use the most efficient hardware instructions available on a user's machine. The library is many times faster than GNU's implementation of `math.h` and able to produce results within 1 ULP of the true mathematical answer.

Figure 5.4(a-c) shows the results obtained by running Stoke on three representative kernels: a bounded periodic function (`sin`), a continuous unbounded function (`log`), and a discontinuous unbounded function (`tan`). We have omitted results for `sin` and `exp` which are similar. The library approximates the true mathematical functions using an implementation based on polynomials. Although there is a direct relationship between the number of terms in those polynomials and the precision of a kernel the details are quite complex at the binary level. Simply deleting an instruction does not correspond to deleting a term and will usually produce wildly inaccurate results.



(a) LOC/speedup for `sin` (b) LOC/speedup for `log` (c) LOC/speedup for `tan`

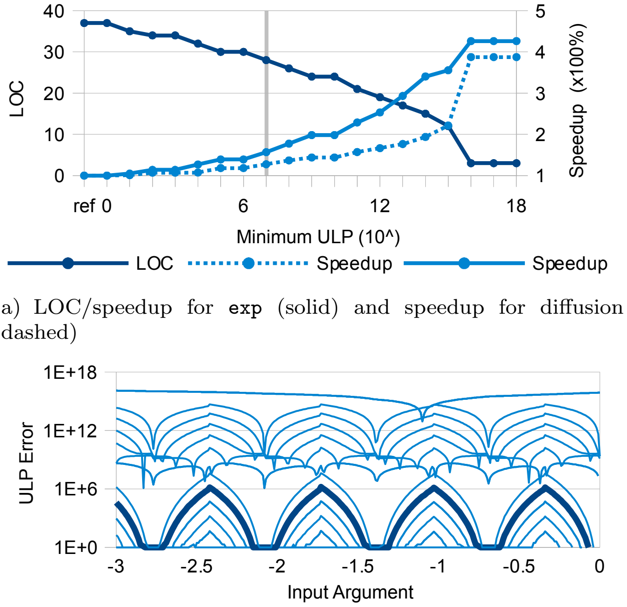
(d) ULP error for `sin` rewrites (e) ULP error for `log` rewrites (f) ULP error for `tan` rewrites

Figure 5.4: Representative kernels from Intel's implementation of `math.h`. Increasing η produces rewrites that interpolate between double- single- and half-precision (vertical bars, shown for reference) (a-c). Errors functions introduced by reduced precision are well-behaved (d-f) and amenable to MCMC sampling.

Reference points are given on the far left of each plot. The original kernels range in length from 66 to 107 lines of code (LOC) and represent a baseline speedup of 1x. Figures 5.4(a-c) show the results of varying η between 1 and 10^{18}, approximately the number of representable double-precision floating point values. For reference we have highlighted $\eta = 5 \cdot 10^9$, and $4 \cdot 10^{12}$ which correspond respectively to the ULP rounding error between the single- and half-, and double-precision representations of equivalent floating-point values. Setting η to either value and providing STOKE with a double-precision target corresponds to asking STOKE to produce a single- or half-precision version of a double-precision algorithm.

By increasing η STOKE was able to experiment with modifications to the code which meet the desired bounds on precision. The result is a set of implementations that interpolate between double-, single-, half-precision, and beyond. For very large η STOKE was able to remove nearly all instructions (for $\eta =$ `ULLONG_MAX`, not shown, STOKE produced the empty rewrite) and produce speedups of up to 6x over the original implementations. However performance improvements grew smoothly and were still significant for reasonable precisions. Although no longer available, Intel at one point offered a variable-precision implementation of `libimf` that allowed the user to trade performance for higher quality results. Using only the full double-precision implementation STOKE was effectively able to automatically generate the equivalent library.

Two factors complicate the verification of the rewrites discovered by STOKE for `libimf`: code length and the mixture of fixed- and floating-point computation. At over 100 instructions in length



(a) LOC/speedup for `exp` (solid) and speedup for diffusion (dashed)

(b) ULP error for `exp` rewrites

Figure 5.5: The diffusion leaf task from the S3D direct numeric simulation solver. Increasing η allows STOKE to trade precision for shorter code and higher performance (a). The task can tolerate a less-than-double-precision implementation (vertical bar, shown for reference), which produces a a 27% overall speedup. Errors are well-behaved (b).

verifying a rewrite against the kernels in `libimf` is well beyond the capabilities of current floating-point decision procedures. Additionally, the kernels in `libimf` use hardware-dependent bit-wise arithmetic to extract the fields from a floating-point value for use as indices into a table of constants. Although an abstract interpretation based on ranges might be able to deal with the primarily polynomial operations of those kernels, without an appropriate set of invariants for the values contained in those tables any sound results of the analysis would be uselessly imprecise.

Our MCMC sampling method for characterizing maximum error works well on these kernels. Figure 5.4(d-f) shows the error function for each of 20 rewrites depicted in Figure 5.4(a-c) for a representative range of inputs. Although an exhaustive enumeration of these functions is intractable they are reasonably well-behaved and MCMC sampling was quickly able to determine their maximum values.

S3D is a three dimensional direct numeric simulation solver for HCCI combustion. It is one of the primary applications used by the U.S Department of Energy for investigating combustion mechanisms as potential replacements for gasoline-based engines. S3D is designed to scale on large

```
float dot(V& v1, V& v2) {
  // v1  = [xmm0[63:32], xmm0[31:0], xmm1[31:0]]
  // v2  = [(rdi),       4(rdi),     8(rdi)    ]
  // ret = [xmm0[63:32], xmm0[31:0], xmm1[31:0]]

  return v1.x*v2.x + v1.y*v2.y + v1.z*v2.z;
}
```

```
# gcc -O3            # STOKE

movq  xmm0, -16(rsp) vpshuflw -2, xmm0, xmm2
mulss 8(rdi), xmm1   mulss 8(rdi), xmm1
movss (rdi), xmm0    mulss (rdi), xmm0
movss 4(rdi), xmm2   mulss 4(rdi), xmm2
mulss -16(rsp), xmm0 vaddss xmm0, xmm2, xmm5
mulss -12(rsp), xmm2 vaddss xmm5, xmm1, xmm0
addss xmm2, xmm0
addss xmm1, xmm0
```

Figure 5.6: Vector dot-product. Stoke is unable to make full use of vector intrinsics due to the program-wide data structure layout chosen by `gcc`. The resulting code is nonetheless faster, and amenable to verification using uninterpreted functions.

supercomputers and relies on a significant amount of parallelism to achieve high performance. Computation in S3D is split into parallel and sequential phases known as tasks. Despite the enormous amounts of data that these tasks consume they are orchestrated in such a way that the time spent in inter-node data communication between tasks is kept to a minimum. For some kernels this orchestration is so effective that the resulting runtimes have been made compute bound. As a result, substantial performance gains can be made by optimizing these computations.

We applied our implementation of Stoke to a CPU implementation of S3D, namely the diffusion task which computes coefficients based on temperature, pressure, and molar-mass fractions of various chemical species, and is representative of many high-performance numerical simulations in that its compute time is dominated by calls to the `exp` function. The performance of this function is so important that the developers ship a hand-coded implementation which approximates the function using a Taylor series polynomial and deliberately omits error handling for irregular values such as infinity or NaN. As above, Figure 5.5 shows the result of varying η between 1 and 10^{18}. By decreasing precision Stoke was able to discover rewrites that were both faster and shorter than the original implementation. Despite its heavy use of the `exp` kernel the diffusion leaf task loses precision elsewhere and does not require full double-precision to maintain correctness. The vertical bar in Figure 5.5(a) shows the maximum precision loss that the diffusion kernel is able to tolerate. Using the rewrite discovered when $\eta = 10^7$ which corresponds to a 2x performance improvement for the `exp` kernel produces a full leaf task performance improvement of 27%.

	Latency		LOC		Total	Bit-wise	
Kernel	$\mathcal{T}$	$\mathcal{R}$	$\mathcal{T}$	$\mathcal{R}$	Speedup	Correct	OK
$k \cdot \bar{v}$	13	13	8	6	3.7%	yes	yes
$\langle \bar{v}_1, \bar{v}_2 \rangle$	20	18	8	6	5.6%	yes	yes
$\bar{v}_1 + \bar{v}_2$	13	10	9	4	30.2%	yes	yes
$\Delta(\bar{v}_1, \bar{v}_2)$	26	19	29	14	36.6%	no	yes
$\Delta'(\bar{v}_1, \bar{v}_2)$	26	13	29	10	n/a	no	no

Figure 5.7: Speedups for aek. Bit-wise correct optimizations produce a cumulative speedup of 30.2%. Lower precision optimization to the camera perturbation kernel, $\Delta(\cdot)$, produces an additional 6.4% speedup. More aggressive optimization, $\Delta'(\cdot)$, produces further latency reduction, but is unusable.

As above, Figure 5.5(b) shows the error function for each of the 20 rewrites shown in Figure 5.5(a). For reference we have highlighted the error curve which corresponds to the most aggressive rewrite that the diffusion leaf task was able to tolerate. The functions are well-behaved and a global maximum of $1,730,391$ ULPs is discovered quickly. Bit-shifts, bit-extraction, and kernel length prevented the application of current sound symbolic techniques to this example.

aek is an extremely compact implementation of a ray tracer written in response to Paul Heckbert's famous challenge that an author be able to fit the entire source program on the back of a business card [52]. Despite its compactness the program is quite complex and is capable of generating scenes with textured surfaces, gradients, reflections, soft shadows and depth of field blur. The application is typical of ray tracers in that it spends the majority of its compute time performing vector arithmetic. The core loop of the algorithm computes the path and intersection of light rays with objects in the environment to determine the color deposited on each pixel of the resulting image.

aek is unique among the benchmarks shown so far in that its kernels all operate on compound data structures: vectors represented as triplets of floats. As a result Stoke was forced to obey the program-wide data structure layout chosen by `gcc`. Consider the vector dot product kernel shown in Figure 5.6, in particular the fact that `gcc` has chosen to split the layout of vectors between two sse registers (`xmm0` and `xmm1`) when it could easily have fit into just one. This decision precluded Stoke from producing the obvious optimization of performing all three multiplications using a single vector instruction; the overhead required to move the data back and forth into the appropriate configuration was too great. Nonetheless, Stoke was able to discover an optimization that performs less data movement, eliminates stack traffic, and is 2 cycles faster than the original. The resulting code is additionally amenable to verification using uninterpreted functions and can be shown using Z3 to be bit-wise correct for all inputs. We omit discussion of the remaining vector kernels which have similar properties and produce the performance improvements summarized in Figure 5.7.

By focusing only on vector kernels, Stoke was able to produce a cumulative improvement of just over 30% in total program runtime. However by identifying portions of the program where bit-wise correctness is unnecessary even further improvements can be made. aek uses randomness to induce depth of field blur by perturbing the camera angle in its inner-most loop. So long as the imprecision introduced by Stoke rewrites are at or below the order of magnitude of the noise injected by that randomness the difference in results is imperceptible to the application as a whole.

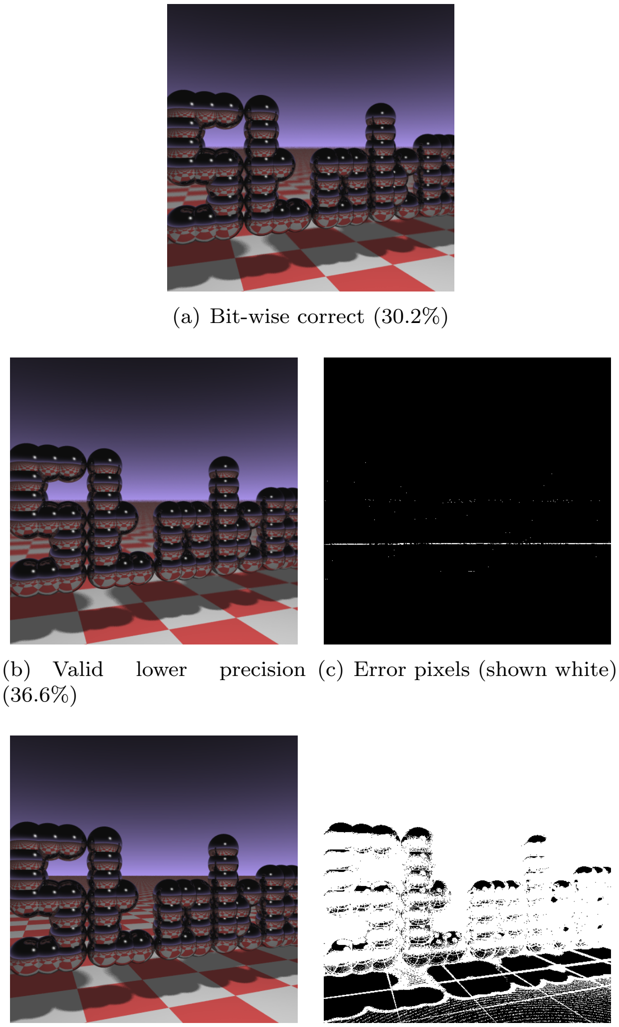
Figure 5.8 shows the result of applying Stoke to the kernel which perturbs the camera angle. Stoke was able to discover a rewrite that takes advantage of the bit-imprecise associativity of multiplication and drops terms that take negligibly small values due to the values of program-wide constants. The optimized kernel results in an additional 6.4% overall performance improvement for the application as a whole. Because this kernel was generated using a general purpose compiler as opposed to having been written by an expert it does not perform bit-fiddling on the internal representation of floating-point values and is amenable to verification using an abstract interpretation based on ranges. The resulting static bound of 1363.5 ULPs is considerably weaker than the maximum 5 ULP error discovered using Mcmc sampling.

Figure 5.9 summarizes the cumulative effects of the optimizations discovered by Stoke. Figure 5.9(a) shows the image that was generated using only bit-wise correct optimizations whereas Figure 5.9(b) shows the image that was generated using imprecise optimizations as well. Although the images appear identical they are in fact different as show in Figure 5.9(c). Further optimizations to the kernel that perturbs camera angle are possible and result in a latency reduction of 50% compared to the code produced by `gcc` but at the price of unintended program behavior. For values of η that exceed the noise introduced by randomness, Stoke removed the code that produces the camera perturbation altogether. The result is an image without depth of field blur which is depicted in Figure 5.9(d). As shown in Figure 5.9(e) the image differs dramatically from the original. (This effect may be difficult to observe in printed copy; note in particular the absence of depth of field blur on the horizon.)

```
V delta(V& v1, V& v2, float r1, float r2) {
  // v1  = [(rdi),      4(rdi),      8(rdi)    ]
  // v2  = [(rsi),      4(rsi),      8(rsi)    ]
  // ret = [xmm0[63:32], xmm0[31:0], xmm1[31:0]]

  assert(0.0 <= r1 <= 1.0 && 0.0 <= r2 <= 1.0);

  // gcc -O3:
  return V(99*(v1.x*(r1-0.5))+99*(v2.x*(r2-0.5)),
           99*(v1.y*(r1-0.5))+99*(v2.y*(r2-0.5)),
           99*(v1.z*(r1-.05))+99*(v2.z*(r2-0.5)));
  // STOKE:
  return V(99*(v1.x*(r1-0.5)),
           99*(v1.y*(r1-0.5)),
           v2.z*(99*(r2-0.5)));
}
```

```
# gcc -O3               # STOKE

movl 0.5, eax           movl 0.5 eax
movd eax, xmm2          movd eax, xmm2
subss xmm2, xmm0        subps xmm2, xmm0
movss 8(rdi), xmm3      movl 99.0, eax
subss xmm2, xmm1        subps xmm2, xmm1
movss 4(rdi), xmm5      movd eax, xmm4
movss 8(rsi), xmm2      mulss xmm4, xmm1
movss 4(rsi), xmm6      lddqu 4(rdi), xmm5
mulss xmm0, xmm3        mulss xmm0, xmm5
movl 99.0, eax          mulss (rdi), xmm0
movd eax, xmm4          mulss xmm4, xmm0
mulss xmm1, xmm2        mulps xmm4, xmm5
mulss xmm0, xmm5        punpckldq xmm5, xmm0
mulss xmm1, xmm6        mulss 8(rsi), xmm1
mulss (rdi), xmm0
mulss (rsi), xmm1
mulss xmm4, xmm5
mulss xmm4, xmm6
mulss xmm4, xmm3
mulss xmm4, xmm2
mulss xmm4, xmm0
mulss xmm4, xmm1
addss xmm6, xmm5
addss xmm1, xmm0
movss xmm5, -20(rsp)
movaps xmm3, xmm1
addss xmm2, xmm1
movss xmm0, -24(rsp)
movq -24(rsp), xmm0
```

Figure 5.8: Random camera perturbation. STOKE takes advantage of the bit-imprecise associativity of floating point multiplication, and the negligibly small values of some terms due to program-wide constants to produce a rewrite which is 7 cycles faster than the original code.



(a) Bit-wise correct (30.2%)

(b) Valid lower precision (36.6%)

(c) Error pixels (shown white)

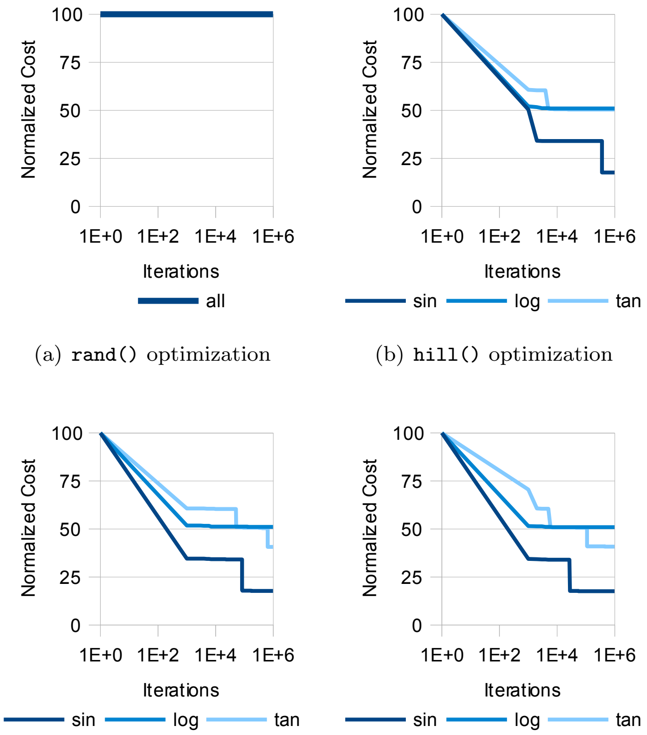
(d) Invalid lower precision

(e) Error pixels (shown white)

Figure 5.9: Images generated using bit-wise correct (a) and lower precision (b) optimizations. The results appear identical, but are in fact different (c). Further optimization is possible but incorrectly eliminates depth of field blur (d,e). In particular, notice the sharp horizon.

As remarked in Chapter 2, although Mcmc has been successfully applied to many otherwise intractable application domains it is not the only stochastic search procedure that could have been used in the implementation of Stoke. We modified both the optimization and validation procedures used by our implementation to use three alternate algorithms, pure random search, greedy hill-climbing, and simulated annealing, and for each variant reran a subset of the experiments described above.

Figure 5.10 shows the results of running Stoke's optimization routine on each of the three `libimif` kernels for $\eta = 10^6$. Each curve corresponds to a different kernel and represents the best



(a) `rand()` optimization

(b) `hill()` optimization

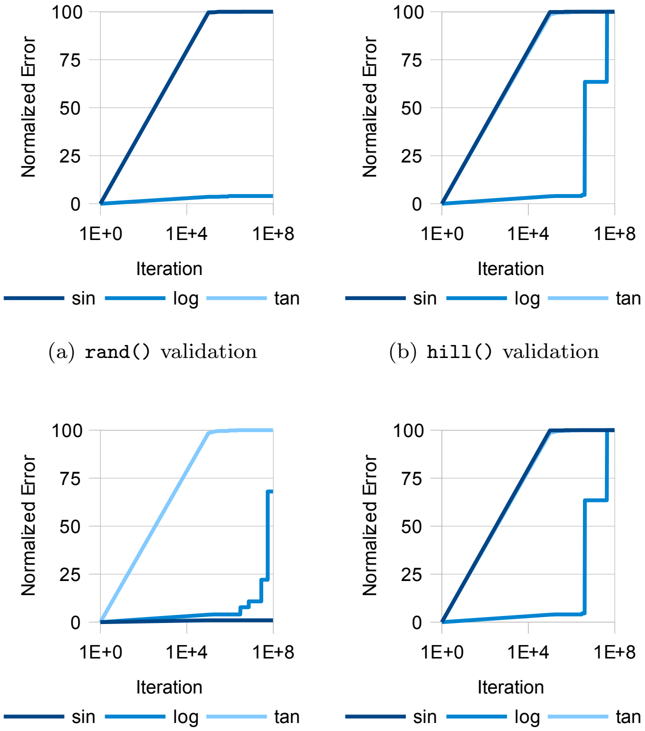
(c) `anneal()` optimization

(d) `mcmc()` optimization

Figure 5.10: Alternate implemenatations of the search procedure used by Stoke for optimization: random search (`rand()`), greedy hill-climbing (`hill()`), simulated annealing (`anneal()`) and Mcmc sampling (`mcmc()`). Mcmc sampling outperforms the alternatives for optimization.

cost discovered over time. In all cases one million iterations was sufficient for each search procedure to reach a point where no further progress was made. For all three kernels random search was unable to improve on the original input. Without a mechanism for measuring correctness even a small number of random moves are most likely sufficient to guarantee that a random walk will never return to a correct implementation. Greedy hill-climbing performed comparably to Mcmc sampling but produced slightly larger final costs and simulated annealing performed comparably to greedy hill-climbing but took longer to do so. As remarked in Chapter 2, simulated annealing can be thought of as a hybrid method that behaves similarly to random search initially and then tends towards the behavior of greedy hill-climbing. As random search seems ill-suited to the optimization task the longer convergence times for simulated annealing can be attributed to time spent initially on wasted effort.

Figure 5.11 shows the results of running Stoke's validation routine on an identical representative result for each of the three `libimf` kernels produced for $\eta = 10^6$. As above each curve corresponds



(a) `rand()` validation

(b) `hill()` validation

(c) `anneal()` validation

(d) `mcmc()` validation

Figure 5.11: Alternate implemenatations of the search procedure used by Stoke for validation: random search (`rand()`), greedy hill-climbing (`hill()`), simulated annealing (`anneal()`) and Mcmc sampling (`mcmc()`). Mcmc is not clearly better-suited to validation.

to a different kernel and represents the largest error discovered over time. In all cases ten million iterations was sufficient for each search procedure to reach convergence. Both Mcmc sampling and greedy hill-climbing were able to produce nearly identical results and differed from each other by no more than 2 ULPs. However neither was able to produce a consistently larger result. Random search performed similarly for some kernels but poorly for others. This result is likely due to the shape of the error functions shown in Figure 5.4(d-f) which are dense with sharp peaks. Without a mechanism for tracking towards local maxima it is unlikely that a purely random search procedure would be able to produce consistently good results. As above, the performance of simulated annealing appears to be something of a mix between that of random search and greedy hill-climbing.

Overall it is difficult to assess the performance of stochastic optimization algorithms. Different application domains can be more or less amenable to one technique than another and parameter settings can make the difference between a successful implementation and one that is indistinguishable from pure random search. The results shown above suggest that Mcmc sampling is well-suited to the program optimization task and perhaps less uniquely so to the validation task. However this observation may be an artifact of the relative complexity of the two. The space of all `x86_64` code sequences is considerably larger and higher dimensional than that of inputs to the primarily unary floating-point kernels used by our benchmarks. Comparing the relative performance of different search strategies on the validation of higher arity kernels remains a direction for future work.

5.3 Discussion

In this chapter we described a new approach to the optimization of high-performance floating-point kernels which is based on random search. Floating-point instruction sets have complicated semantics and our technique both eliminates the dependence on expert-written optimization rules and allows a user to customize the extent to which precision is sacrificed in favor of performance.

The general purpose verification of loop-free kernels containing floating-point computations remains a hard problem and we are unaware of any existing systems that can be used to verify the large variety of optimizations produced by STOKE. To address this limitation, we described a general-purpose randomized validation technique that can be used to establish strong evidence for the correctness of floating-point code sequences. Although our technique is not applicable to application domains that require formal verification, for many high-performance applications it is a large improvement over the state of the art which relies on nothing more than confidence in the compiler writer and regression test suites.

The implementation of STOKE described in this chapter is the first instance of a stochastic optimizer that has been successfully applied to full programs; it is able to produce significant speedups on both Intel's handwritten implementation of the `C` numerics library and full end-to-end performance improvements on a direct numeric simulation solver and a ray tracer. Nonetheless, further opportunities for optimization remain. Notably extensions that allow for optimizations based on the reorganization of program-wide data structures are a high-value direction for future work.

5.4 Related Work

We are not the first to propose program transformations that sacrifice bit-wise precision in favor of performance. Loop perforation dynamically removes loop iterations to balance an efficiency-precision trade-off [91], and given an input distribution some perforations can be statistically guaranteed [73]. However, the resulting guarantees are statistically unsound for any other input distribution. Green [5] and other systems for performing approximate computation [104] allow a programmer to provide multiple implementations of the same algorithm, each of which represent different precision/power trade-offs. These systems emit runtime code that switches between implementations based on power consumption. Our approach provides a method for generating a range of implementations automatically instead of relying on an expert to provide them. Program synthesis techniques such as [46, 93] are currently inapplicable to this task as they rely on decision procedures that do not scale to non-trivial floating-point programs.

Techniques that can be extended to bounding the error of floating-point optimizations are not limited to those described in this chapter. A method for proving robustness and continuity of programs is presented in [16] and a black box testing approach for establishing Lipschitz continuity is described in [56]. Nonetheless, many high-performance kernels such as `exp`(·) are not Lipschitz continuous and functions such as `tan`(·) are not even continuous. Thus the applicability to Stoke is limited. Interactive theorem provers such as [24] may be applicable to the non-trivial optimizations discovered by Stoke but the result would be far from a fully automatic system.

Random interpretations [75] can provide strong statistical guarantees of program behavior. However, the number of samples required to produce these guarantees depends crucially on the operators used by a program. Bit-extraction operators for example, can require an intractable number of samples. Additional testing infrastructures for floating-point code that have no statistical guarantees include [62] and [8]. Monte Carlo testing approaches for checking numerical stability which is a concern orthogonal to the correctness of optimizations include [95, 58, 96]. Mcmc sampling has also been applied to a related program analysis task, the generation of test cases that obtain good program coverage [86].

The application of search techniques to the optimization and verification of floating-point code has recently attracted considerable attention. A search procedure for producing inputs that maximize the relative error between two floating-point kernels which is similar to simulated annealing appears in [19]. A brute-force approach to replacing double-precision instructions with their single-precision equivalents appears in [63], and a randomized technique for producing floating-point narrowing conversions at the source code level is discussed in [84]. Both tools preserve semantics as written and are incapable of producing the aggresive optimizations discovered by Stoke.

Examples of how unsound compiler optimizations can negatively affect the stability of numerical algorithms are described in [41] and a critique of using execution time for measuring improvement in performance is discussed in [22]. The architecture of Stoke addresses both of these concerns. Although it is true that different memory layouts can result in inconsistent speedups, Stoke produces multiple rewrites by design and can produce a suitable rewrite for each such layout. For Stoke, execution time is a suitable optimization metric. With respect to correctness, Stoke uses test case data to generate customized optimizations that are specialized to user-specified input ranges. These optimizations may be incorrect in general but perfectly acceptable given the constraints on inputs described by the user or generated by a technique such as [19].

Chapter 6

Loops

In this chapter we extend Stoke to the optimization of code sequences that contain loops. As we will show, the primary obstacle to doing so is formal verification. Equivalence checking of loops is a fundamental problem with potentially significant applications, particularly in the area of compiler optimizations. However, the current state of the art in equivalence checking is quite limited: given two `x86_64` loops, no existing technique is capable of verifying equivalence automatically, even if they differ only in the application of standard loop optimizations. Here we present the first practically useful, automatic, and sound equivalence checking engine for `x86_64` loops.

Existing techniques for proving equivalence can be classified into three categories: sound algorithms for loop-free code [2, 34, 33, 21, 71]; algorithms that analyze finite unwindings of loops or finite spaces of inputs [81, 77, 61, 55]; algorithms that require knowledge of the particular transformations used for turning one program into another [72, 97] and the order in which the transformations have been applied [78, 74, 40]. In contrast to these approaches, we do not assume any knowledge about the optimizations performed.

In outline, the approach works as follows. First, we guess a *simulation relation* [74]. Roughly speaking, a simulation relation breaks two loops into a set of pairs of loop-free code fragments where logical formulas associated with each pair describe the relationship of the input states of the fragments to the output states of the fragments. Second, we generate verification conditions encoding the `x86_64` instructions contained in each loop-free fragment as Smt [25] constraints. Finally, we construct queries that verify that the guessed relationships between the code fragments in fact hold. By construction, if the queries succeed they constitute an inductive proof of equivalence.

It is worth stressing that this approach works directly on unmodified binaries. The `x86_64` instruction set is large, complex, and difficult to analyze statically. The key idea that makes the approach effective in practice, and even simply feasible to build, is that the process of guessing a simulation relation is constructed not via static code analyses, but by using data collected from test cases. Because the approach is data driven, it is able to directly examine the precise net effect of code sequences without first going through a potentially lossy abstraction step. The use of test cases is of course an under-approximation and may not capture all possible loop behaviors. However it is sound; a lack of test case coverage may cause equivalence checking to fail, but it cannot result in the unsound conclusion that two loops are equivalent when they are not.

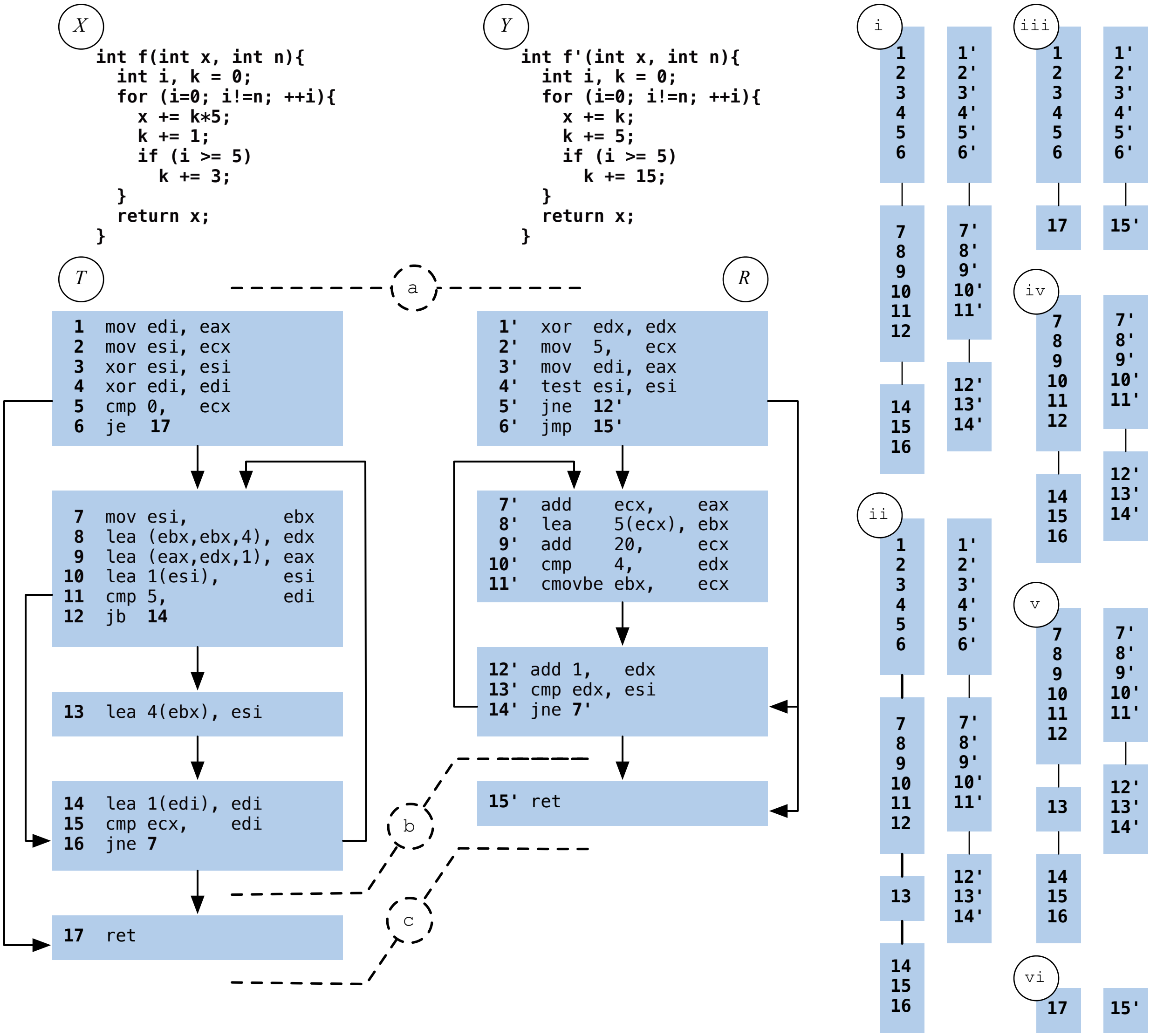
In this chapter we describe the implementation of these ideas in a tool called Ddec (Data-Driven Equivalence Checker) and use Ddec to extend the applicability of Stoke to loops. By replacing Stoke's implementation of the $\text{eq}_{\text{slow}}(\cdot)$ constraint by Ddec we show that Stoke is able to perform optimizations beyond its original capabilities, in fact producing verified code that contains loops and is comparable in performance to `gcc -O3`.

6.1 Loop Equality

Figure 6.1 shows two versions of a function taken from [97] where a straightforward implementation X was optimized using a strength reduction [4] to produce the code Y; corresponding `x86_64` code sequences are shown below. We use primes to denote program points and registers corresponding to the rewrite (right) and unprimed characters for those corresponding to the target (left). In addition to the strength reduction, the rewrite also takes advantage of several low level compiler optimizations such as the use of an `x86_64` conditional-move on line `11`$'$ to eliminate the jump on line `12`. We are unaware of any fully automatic technique that is capable of verifying the equivalence of these two code sequences. Nonetheless, our goal is to verify equivalence in the absence of source code, manually written expert rules for equivalence, compiler source code, or compiler annotations.

To verify that the two code sequences are equivalent, we must confirm that whenever the target and the rewrite begin execution in identical machine states and the target runs to completion, the rewrite is guaranteed to terminate in the same machine state with the same return value. In this example which does not use memory, we limit our discussion of machine states to a valuation of the subset of hardware registers that the two codes use: `eax`, `ebx`, `ecx`, `esi`, and `edi` (our approach does not make this assumption in general and handles memory reads and writes soundly) and assume that `eax` is live on exit from the function.

We use the well-known concept of a *cutpoint* [99] to decompose equivalence checking into manageable sub-parts. A cutpoint is a pair of program points — one in each program — that is chosen to divide loops into loop-free segments. The cutpoints in Figure 6.1, `a`, `b`, and `c`, segment the sequences into three parts: the code from `a` to `b` which excludes the backedge of the loop, the code that begins from and returns to `b`, and the code that begins from `b`, exits the loop, and terminates at `c`. Using these three cutpoints, we can produce an inductive proof that shows that the executions of the code sequences move together from one cutpoint to the next and that at each cutpoint certain invariants are guaranteed to hold.



X

```
int f(int x, int n){
  int i, k = 0;
  for (i=0; i!=n; ++i){
    x += k*5;
    k += 1;
    if (i >= 5)
      k += 3;
  }
  return x;
}
```

Y

```
int f'(int x, int n){
  int i, k = 0;
  for (i=0; i!=n; ++i){
    x += k;
    k += 5;
    if (i >= 5)
      k += 15;
  }
  return x;
}
```

T

```
 1  mov edi, eax
 2  mov esi, ecx
 3  xor esi, esi
 4  xor edi, edi
 5  cmp 0,   ecx
 6  je  17
```

```
 7  mov esi,          ebx
 8  lea (ebx,ebx,4), edx
 9  lea (eax,edx,1), eax
10  lea 1(esi),      esi
11  cmp 5,           edi
12  jb  14
```

```
13  lea 4(ebx), esi
```

```
14  lea 1(edi), edi
15  cmp ecx,    edi
16  jne 7
```

```
17  ret
```

R

```
 1'  xor   edx, edx
 2'  mov   5,   ecx
 3'  mov   edi, eax
 4'  test  esi, esi
 5'  jne   12'
 6'  jmp   15'
```

```
 7'  add    ecx,     eax
 8'  lea    5(ecx),  ebx
 9'  add    20,      ecx
10'  cmp    4,       edx
11'  cmovbe ebx,     ecx
```

```
12' add 1,   edx
13' cmp edx, esi
14' jne 7'
```

```
15' ret
```

Figure 6.1: Equivalence checking for two possible compilations: (X) no optimizations applied either by hand or during compilation, (Y) optimizations applied. Cut points (a,b,c) and corresponding paths (i-vi) are shown.

1. If both sequences begin at a with identical machine states and transition immediately to c, they terminate with identical return values.
2. If both sequences begin at a with identical machine states and transition to b, they both satisfy I.
3. If both sequences begin at b, satisfy I, and return to b, they both satisfy I.
4. If both sequences begin at b, satisfy I, and transition to c, they terminate with identical return values.

Figure 6.2: Partial inductive proof of equivalence for the code sequences shown in Figure 6.1.

The required invariants at a and c follow directly from the problem statement. At a, we require that the sequences agree on initial machine states and at c we require that they agree on the return value stored in eax. The first problem that we encounter is identifying the invariant that must hold at b. This invariant is a relation between the machine states of the code sequences. Here we restrict ourselves to invariants that consist of equality relationships between elements of the two states. Once we have identified the appropriate invariant I, an inductive proof would take the form shown in Figure 6.2. However the proof is incomplete. It does not guarantee that if the target makes a transition, then the rewrite does the same. We do not for instance want the target to transition from a to b while the rewrite transitions from b to c. The proof can be completed as follows.

Every transition between cutpoints is associated with the *code paths* that must be executed to move from one cutpoint to the next. For example, in moving from cutpoint b to c, both sequences execute the instructions shown in code paths vi. We say that a code path for the target $p_{\mathcal{T}}$ *corresponds* to a code path for the rewrite $p_{\mathcal{R}}$ if they begin and end at the same cutpoints, and whenever the target follows $p_{\mathcal{T}}$ the rewrite follows $p_{\mathcal{R}}$. Figure 6.1 shows a complete set of corresponding paths for this example: i and ii are associated with transition a-b, iii is associated with a-c, iv and v are associated with b-b and vi is associated with b-c. Our proof must ensure that whenever the target follows a code path then the rewrite must follow the corresponding path. Identifying these paths is a second major difficulty. The question of what invariants hold at b is crucial, as these must be strong enough to statically prove that the execution of the rewrite follows the corresponding paths of the target and that the executions of both sequences proceed through the same series of cutpoints.

Our solution to both problems, identifying the equalities that hold at b and the corresponding paths of the two code sequences, is to analyze execution data. We identify corresponding paths by matching traces for both loops on identical test inputs against cutpoints. We observe the instructions executed by the sequences in moving from one cutpoint to the next and label them as corresponding. For example, for a test case with initial state edi $= 0$ and esi $=$ 1 the target begins its execution from a and executes instructions 1 to 16 to reach b. It then exits the loop and transitions from b to

$$\begin{bmatrix} \texttt{eax} & \texttt{esi} & \texttt{edi} & \texttt{ecx} & \texttt{eax}' & \texttt{ecx}' & \texttt{edx}' & \texttt{esi}' \\ 0 & 1 & 1 & 2 & 0 & 5 & 1 & 2 \\ 5 & 2 & 2 & 2 & 5 & 10 & 2 & 2 \end{bmatrix}$$

Figure 6.3: Live register values at `b` for the code sequences shown in Figure 6.1 for a test input where $\texttt{edi} = 0$ and $\texttt{esi} = 2$.

`c` by executing instruction `17`. The rewrite begins its execution from `a` and executes instructions $\texttt{1}'$ to $\texttt{14}'$ to reach `b`. It then executes instruction $\texttt{15}'$ to reach `c`. From this test case, we conclude that code paths with instructions `1` to `16` correspond to instructions $\texttt{1}'$ to $\texttt{14}'$ (`i`) and that `17` corresponds to $\texttt{15}'$ (`vi`).

The equality conditions at `b` can be determined by inserting instrumentation at `b` to record the values of the live registers for both sequences and observing the results. For a test input where $\texttt{edi} = 0$ and $\texttt{esi} = 2$, we obtain the matrix shown in Figure 6.3 where each row corresponds to the values in live registers when both programs pass through `b`. The first row can be interpreted as saying that when the target reaches `b` for the first time, the registers have the values shown in columns `eax` , `esi`, `edi`, and `ecx`, and when the rewrite reaches `b` for the first time the registers have the values shown in $\texttt{eax}'$, $\texttt{ecx}'$, $\texttt{edx}'$, and $\texttt{esi}'$. The second row shows the values of the registers when `b` is reached for the second time (i.e. during the next iteration of the loops). Using standard linear algebra techniques, it is possible to extract the following relationships, which are sufficient candidates for the equalities that must hold at `b`: $\texttt{eax} = \texttt{eax}'$, $5 * \texttt{esi} = \texttt{ecx}'$, $\texttt{edi} = \texttt{edx}'$, and $\texttt{ecx} = \texttt{esi}'$. The use of linear algebra on test data for invariant inference is well studied in software verification and the limitations are known. The process may generate spurious equality relationships [76], however these can be systematically eliminated using a theorem prover [89]. We note that this method assumes that equality relationships are sufficient to prove program equivalence and we do not for example, consider invariants that contain inequalities. Previous work on translation validation [97, 74] makes the same assumption, which we find to be largely sufficient in practice.

Although it is possible that the tests used to generate these values do not produce sufficient coverage, or that either more corresponding paths exist or spurious equality relationships are discovered at `b`, the consequence is simply that the proof will fail, and we will report that the two functions could be different. If this is the case. we can simply reattempt the proof with more test cases. Barring this possibility, almost all of the limitations of our technique can be mapped to the restricted expressiveness of invariants. Better invariant generation techniques will only improve performance.

We now present a formal description of the algorithm sketched above. We assume that we are given two code sequences — each of which contains one natural loop and no function calls — and that our goal is to infer a candidate *simulation relation* which consists of cutpoints and linear equalities as invariants, and to check whether the candidate is an actual simulation relation. If so, we have a proof of equivalence. We begin with a suitable definition of equality and proof obligations:

Definition 33 (Equivalence)**.** Two code sequences are *equivalent* if for all possible states s, when execution of the target begins from s and the target terminates in s' without aborting, the rewrite does the same.

Definition 34 (Proof Obligation)**.** Let t be a code path in the target, r a code path in the rewrite, and C the pair $\langle t, r\rangle$. For predicates P and Q, a *proof obligation* ($\{P\}\langle t, r\rangle\{Q\}$) states that if t begins execution from a state s_1 and r begins execution from a state s_2, and if $P(s_1, s_2)$ holds and t terminates in s'_1 without aborting, then r does not abort and $Q(s'_1, s'_2)$ holds for all possible final states s'_2 of r.

We generate corresponding paths t and r for proof obligations using *cutpoints* [99]. A cutpoint n is a pair of program points $\langle \eta_{\mathcal{T}}, \eta_{\mathcal{R}}\rangle$ where $\eta_{\mathcal{T}}$ is part of the target and $\eta_{\mathcal{R}}$ is part of the rewrite. We select cutpoints using a heuristic that prefers pairs of program points in which the corresponding memory states agree on the largest number of values. Intuitively, cutpoints with higher agreement between memory states generally correspond to simpler invariants than cutpoints where the relationship is more involved.

We create a candidate cutpoint for every program point pair $\langle \eta_{\mathcal{T}}, \eta_{\mathcal{R}}\rangle$ and for every test t we compute $m^t_{\eta_{\mathcal{T}}}$ (resp. $m^t_{\eta_{\mathcal{R}}}$), the number of times control flow passes through $\eta_{\mathcal{T}}$ (resp. $\eta_{\mathcal{R}}$) when the execution of the target (resp. the rewrite) begins from t. If there do not exist constants a, b such that $\forall t . m^t_{\eta_{\mathcal{T}}} = a m^t_{\eta_{\mathcal{R}}} + b$ then $\langle \eta_{\mathcal{T}}, \eta_{\mathcal{R}}\rangle$ is rejected as a candidate cutpoint. We additionally reject candidates for which the number of heap locations in the observed heap states for both code sequences at $\langle \eta_{\mathcal{T}}, \eta_{\mathcal{R}}\rangle$ is not constant across all tests. This operation is feasible as we only run terminating tests that guarantee a bounded memory footprint. For the remaining candidates, we assign a penalty that represents how different the observed heap states are for the two code sequences at $\langle \eta_{\mathcal{T}}, \eta_{\mathcal{R}}\rangle$ and select candidate cutpoints in increasing order of penalty until the code sequences are decomposed into loop free segments.

Redundant candidates are removed so that the minimum number of program point pairs are returned as candidates — we say that a candidate is redundant if the decomposition is still loop free after it has been removed. The resulting set of cutpoints may not be unique and multiple cutpoints may be associated with the same program point in either code sequence. A loop unrolling for example, may produce several cutpoints that share the same program point in the target.

If satisfactory cutpoints cannot be found then our technique will fail. In general, our approach will fail to prove equivalence for code sequences that differ from each other in an unbounded number of memory locations. A loop fusion transformation or an array traversal reordering for example, will result in loops that cannot be proven equivalent. Reasoning about loops of this form requires quantified invariants for which the current state of the art in invariant inference remains immature.

We refer to a set of cutpoints as a *cutset* ($\mathcal{S}$). For every pair of cutpoints n_1 and n_2 in $\mathcal{S}$ we define a *transition* ($\tau \equiv n_1 \triangleright n_2$) from n_1 to n_2. This transition is *associated* with a set of static code paths $\mathcal{P}_1$ of the target and $\mathcal{P}_2$ of the rewrite that go from n_1 to n_2 without passing through some other cutpoint $n_3 \in \mathcal{S}$. The instructions that are executed during a terminating and non-aborting execution of the target and the rewrite can be represented by a sequence of transitions. Using this notation, we define corresponding paths as follows.

Definition 35 (Corresponding Paths)**.** Given a cutset $\mathcal{S}$, we say that a code path t of the target *corresponds* to a code path r of the rewrite if they are associated with the same transition τ and for some execution $\tau_1, \dots, \tau, \dots, \tau_m$, for the transition τ, if the target follows t then the rewrite follows r.

We use the following data-driven algorithm to generate a set of corresponding paths $\mathcal{C}$. We run both the target and the rewrite on a test input and record the sequence of instructions that are executed by both. Next, we traverse the traces in parallel until a cutpoint is reached in both. The pair of paths taken from the previous pair of cutpoints is added to $\mathcal{C}$ and the process is repeated until we reach the end of the traces. We repeat this process for every test input that we have and if the coverage is sufficient then $\mathcal{C}$ will contain all corresponding paths.

Following the work in [74] we consider invariants which are equalities over registers, stack locations, and a finite set of heap locations, and replace stack accesses by reads and writes to named temporaries. To generate these invariants, we perform a liveness analysis over both the target and the rewrite and for each cutpoint (η_1, η_2) create a set of matrices, whose columns are the *live features* (variables or simple functions of those variables) at η_1 and η_2. If the bit-width of the longest live feature is b, we create one column for every b bit live feature. For every live feature of fewer than b bits we create two columns: one with the feature's value zero-extended to b bits and the other with its value sign-extended. The sign- and zero-extensions are necessary as many `x86_64` instructions implicitly perform these operations on registers of different bit-widths. We then instrument both code sequences to record the values of the features at each cutpoint. If the cutpoint (η_1, η_2) is executed m times then we generate m rows in the matrix for (η_1, η_2). Negative values are transformed to large positive values.

We use elementary linear algebra to compute the linear equality relationships between features by computing the *nullspace* or *kernel* for the matrix associated with each cutpoint. Every vector of a matrix's nullspace corresponds to an equality relationship between features at that cutpoint for all test inputs. We simply take the conjunction of equalities generated by all vectors in the basis of the nullspace and return the resulting predicate as a candidate invariant for our candidate simulation relation.

One desirable feature of nullspaces is that no sound equality relationship is missed. Nullspaces can produce spurious equality relationships for lack of sufficient data but if an equality holds statically then our approach will discover it [89]. Intuitively, this is because every possible equality is contained in a candidate invariant unless there is a test that violates it. In the limit, if there are no no tests then the candidate invariant will consist of every possible equality between features, and hence also include the equalities that are present in the true invariants. We generalize the results in [89] which uses nullspaces to compute invariants for a single program to features as follows:

Lemma 1 (Sound Under-approximation)**.** *If x is a set of features at a cutpoint n, and $\mathcal{I}(x)$ is the strongest invariant at n that holds statically and is expressible by conjunctions of linear equalities, then the candidate invariant $I(x)$ obtained by computing the nullspace of test data is a sound under-approximation of $\mathcal{I}$.*

We implemented the technique described above in a tool call Ddec. Ddec computes invariants using the `nullspace` function of the Integer Matrix Library [18] which is specialized for computing the nullspace of integer and rational matrices using p-adic arithmetic, and uses sixteen random tests to generate data for the invariant computation. For a production system, a larger number of tests would likely be necessary. However at under 2 ms for each null space computation we do not expect that this computation would be a bottleneck.

As in Chapter 4, proof obligations are discharged using Z3 [25]. Ddec translates proof obligations $\{P\}\langle t, r\rangle\{Q\}$, where P and Q are predicates over registers, to VCs as follows. Ddec first asserts the constraint P. Next, it iterates over the instructions in t, and for each instruction asserts a constraint that encodes the transformation the instruction induces on the current machine state. These constraints are chained together to produce a constraint on the final state of live outputs with respect to t and analogous constraints are asserted for r. As in previous chapters, operators that are very expensive for Z3 to analyze such as bit vector multiplication and division or floating-point arithmetic are replaced by uninterpreted functions which constrained by some common axioms. Finally, for all pairs of memory accesses at addresses $\texttt{addr}_1$ and $\texttt{addr}_2$, Ddec asserts additional constraints relating their values: $\texttt{addr}_1 = \texttt{addr}_2 \Rightarrow \texttt{val}_1 = \texttt{val}_2$. These aliasing constraints grow quadratically in the number of addresses, though in many cases these constraints can be simplified through dependency analysis.

Using these constraints, Ddec constructs a Z3 query that asks whether there exists an initial state that satisfies P and causes the two code paths to produce values for live outputs that either violate Q or result in different machine states. If the answer is “no” then the obligation is discharged. Otherwise the prover produces a counter example and Ddec fails to verify equivalence. If every VC is discharged successfully, then Ddec has proven that the two functions are equivalent.

Although this implementation is correct, it is overly conservative with respect to stack accesses. Notably, an access to a spill slot [4] will appear indistinguishable from a memory dereference and Z3 will produce a counter-example in which the input addresses to t and r alias with respect to that slot. As a result, any sound optimized code that eliminates stack traffic will be rejected. We address this issue by borrowing an idea from [74], where spill slots are replaced by temporary registers that eliminate the possibility of aliasing between addresses that are passed as arguments to the stack

	LOC			Speedup	
Program	Stoke	Stoke+Ddec	Runtime	vs. `gcc -O0`	vs. `gcc -O3`
Bansal	9	6	5492.75s	1.58×	1.04×
Saxpy	9	9	0.62s	9.22×	1.48×

Table 6.1: Performance results for the loop failure benchmarks from Chapter 4. Stoke+Ddec is able to produce shorter rewrites than Stoke alone. Verification runtimes are tractable and the resulting code outperforms the output of `gcc -O0` and `gcc -O3`.

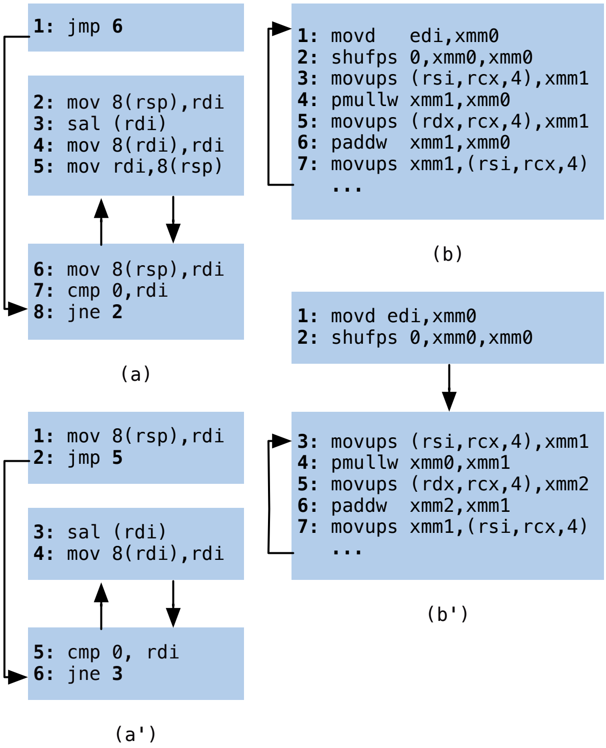
frame. For well-understood compilers such as `gcc`, simple pattern matching heuristics are a good match to this task [74]. For code compiled by `gcc`for example, all stack accesses appear at constant offsets from the register `rbp`. Although modeling spill slots in this way can produce unsound results if an input address is used to form an offset into the current stack frame, this behavior is undefined even in the high-level languages that in theory permit it (e.g., `C`). To the best of our knowledge, existing optimizing compilers are also unsound for such programs.

6.2 Experiments

The experiments in this chapter were performed on a two core 3.40 GHz Intel Core i7-2600. We evaluated our implementation of Ddec by using it as a replacement for the implementation of the $\text{eq}_{\text{slow}}(\cdot)$ constraint described in Chapter 4. Our modified version of Stoke (Stoke+Ddec) was able to produce the two optimizations shown (simplified) in the lower half of Figure 6.4. The optimizations produced by the original version of Stoke are shown above for reference.

Figure 6.4 (a) shows Bansal's linked-list traversal bencmark which iterates over the elements in a list until it discovers a null pointer and multiplies each element by two. In contrast to the result described in Chapter 4, Stoke+Ddec was able to discover an optimization that caches the value of the head pointer in a register across loop iterations. Figure 6.4 (b) shows the Saxpy kernel. Whereas Stoke was only able to optimize the loop-free core of this kernel using vector intrinsics, Stoke+Ddec was able to further improve on that optimization by performing a register renaming and removing the invariant computation of a 128-bit constant from the loop. The code motion is non-trivial and not possible if the registers are not suitably renamed first.

Performance data for these experiments are shown in Table 6.1. We note that the time spent verifying the linked-list benchmark is significantly greater than what was required for the Saxpy benchmark. This is because Stoke+Ddec must reason about complex 64-bit memory equality constraints. The goal of this experiment is simply to describe the effectiveness of our technique, and the application of standard heuristics for constraint simplification — which we have not implemented — could be used to achieve improved performance [29]. To demonstrate just how important constraint simplification is, we performed slicing on VCs [103] for the Saxpy benchmark to eliminate



```
1: jmp 6

2: mov 8(rsp),rdi
3: sal (rdi)
4: mov 8(rdi),rdi
5: mov rdi,8(rsp)

6: mov 8(rsp),rdi
7: cmp 0,rdi
8: jne 2
```

(a)

```
1: mov 8(rsp),rdi
2: jmp 5

3: sal (rdi)
4: mov 8(rdi),rdi

5: cmp 0, rdi
6: jne 3
```

(a')

```
1: movd   edi,xmm0
2: shufps 0,xmm0,xmm0
3: movups (rsi,rcx,4),xmm1
4: pmullw xmm1,xmm0
5: movups (rdx,rcx,4),xmm1
6: paddw  xmm1,xmm0
7: movups xmm1,(rsi,rcx,4)
   ...
```

(b)

```
1: movd edi,xmm0
2: shufps 0,xmm0,xmm0

3: movups (rsi,rcx,4),xmm1
4: pmullw xmm0,xmm1
5: movups (rdx,rcx,4),xmm2
6: paddw  xmm2,xmm1
7: movups xmm1,(rsi,rcx,4)
   ...
```

(b')

Figure 6.4: Simplified versions of the optimizations discovered by STOKE+DDEC. The iteration variable in (a) is cached in a register (a'). The computation of the 128-bit constant in (b) is removed from an inner loop (b').

constraints that were not relevant to the verification task and obtained a result in less than one second. Without these constraint simplifications DDEC timed out after four hours.

6.3 Discussion

Given that we have gone to the effort of constructing a faithful symbolic encoding of the x86_64 instruction set, it is not obvious why we do not simply produce simulation relations statically. The answer is that inference is harder than checking. Consider an application that computes the dot product of two 32-bit arrays, where the multiplication of 32-bit unsigned numbers is used to produce 64-bit results. Say the target uses Karatsuba's trick [60] and performs three 32-bit *signed* multiplications to obtain a 64-bit result, whereas the rewrite uses a special x86_64 instruction that performs an *unsigned* 32-bit multiplication and produces the 64-bit result directly. Any static analysis that would attempt to demonstrate the relationship between these two algorithms would have a very large space of plausible proof strategies to sift through. In fact we know of no inference technique that can reliably perform such reasoning. In contrast, with Ddec the equality manifests itself in the data. Knowing that this specific equality is what needs to be checked narrows the search space to the point that off-the-shelf solvers can be used to dispatch the proof automatically.

In this chapter we described a data-driven approach for verifying the equivalence of loops, and used that technique to extend Stoke to the optimization of code sequences with non-trivial control flow. There are many other applications for this technique, such as the automatic verification that a code refactoring has preserved equivalence, and we hope to explore these in the future. Nonetheless, our approach is not without limitations. Notably, Ddec requires that the expressiveness of the invariants required for a proof to be conjunctions of linear or nonlinear equalities. However, for most interesting intra-procedural optimizations simple equalities appear to be sufficiently expressive [82, 74, 97].

6.4 Related Work

The generation of invariants from test data was pioneered by Daikon [31]. Our technique borrows a number of ideas from previous work in equivalence checking [21], translation validation [74], and software verification [89]. Combining these ideas with our approach to guessing simulation relations from test data yields the first equivalence checking engine for loops written in `x86_64`.

Equivalence checking is an old problem with references that date back to the 1950s. Equivalence checking is common practice in hardware verification where it is well known that cutpoints play a critical role in determining equality. In sequential equivalence checking for example, state-carrying hardware elements constitute cutpoints. Equivalence checking of low-level code has also been studied for embedded software [2, 34, 21, 33, 90]. Notably, none of these techniques support `while` loops and are inapplicable to our benchmarks.

Ddec is more ambitious than the state of the art in equivalence checking for general purpose languages. UC-KLEE [81] performs bounded model checking to check equivalence for all inputs up to a certain size. Other versions of this approach, differential symbolic execution [77] and SymDiff [61], bound the number of loop iterations by a constant, and Semantic Diff [55] checks only whether dependencies are preserved in two procedures. The approach described in [71] handles neither while loops nor pointers, and regression verification [39] only handles partial equivalence; it does not deal with termination. Fractal symbolic analysis [72] and translation validation [78, 74, 40] are both capable of reasoning about loops in general. However, both rely on information about the compiler such as the specific transformations that it can perform. As a result, neither technique is directly applicable to the problem of checking the equivalence of code of unknown provenance.

Porting the work described in [74] to `x86_64`, would require a static analysis that is sound and precise enough to generate simulation relations. This is a decidedly non-trivial engineering task and has never been done; DDEC side-steps this issue by using concrete executions to find cutpoints and invariants. Moreover, the constraints generated by symbolically executing `x86_64` are complex enough that we believe that the decision procedure in [74] will fail to infer equalities in most cases. Nonetheless, compiler annotations are a rich source of high level information and translation validation techniques can handle transformations that DDEC cannot, such as the reordering of traversals over matrices. The ability to infer quantified invariants fully automatically would be necessary for for DDEC to handle optimizations of this complexity.

Some of the most recent work on equivalence checking includes random interpretations [45] and equality saturation [97, 98]. The former represents programs as polynomials which it requires to be of low degree. As a result, bit-manipulations and other similar machine-level instructions are especially problematic for this technique. A one-bit left shift for example, has a polynomial of degree 2^{63} for the carry flag. Unlike equality saturation, DDEC does not rely on expert provided equality relationships between program constructs as these would be difficult to produce by hand for a CISC architecture such as `x86_64`.

Chapter 7

Higher-Order Synthesis

In this final chapter, we demonstrate the use of STOKE as a general-purpose program synthesis tool. Program synthesis is a well-studied technique for automatically generating code sequences in the presence of a formal description of the input-output relationships that those sequences should capture. Although program synthesis has been successfully applied to a number of interesting domains such as the automatic generation of tricky low level code [93, 92, 46], synchronization for concurrency [101, 13], and data-structures [49, 50], there are many compelling real-world application domains for which the current approach to program synthesis remains inadequate. As an example, consider the task of generating a high-performance random number generator which requires both a non-trivial correctness specification and the ability to reason directly about the low-level performance properties of the resulting implementation.

There are two primary reasons for why standard approaches to program synthesis face these limitations. First, current approaches focus mostly on input-output correctness constraints expressed in decidable logics that are incapable of tractably describing *higher-order constraints* (constraints that require higher-order logics to formalize) such as that a random number generator produce samples from a normal distribution. Second, current approaches to program synthesis use SAT/SMT solvers as a mechanism for reasoning about the space of potential results which in turn places a strong restriction on the type of programs that can be searched in an efficient fashion. Most implementations solve this problem by considering a restricted space of programs in a high-level intermediate representation that is close to the underlying logic of the solver. A consequence of this design decision is that low-level requirements that relate to actual machine code implementation, such as performance, cannot be expressed.

STOKE has the potential to address both of these issues. First, STOKE defines the quality of a code sequence in terms of a cost function that can be written in a general-purpose programming language. As a result STOKE can be used to specify arbitrary constraints on both functionality and implementation. For example, the distribution produced by a candidate random number generator can be evaluated by taking a sequence of samples from that generator and comparing it to a sequence of samples produced by a reference implementation. Second, STOKE uses random search to explore the space of possible code sequenes and operates directly on `x86_64` assembly. This in turn leaves STOKE free to use cost functions that take *intensional constraints* (constraints on implementation properties that are beyond input-output behavior) into account when considering potential optimizations.

In this chapter we demonstrate that STOKE is an effective — and to the best of our knowledge, the only — framework for successfully reasoning about higher-order synthesis constraints that extend beyond simple input-output relationships. Furthermore, we show that although STOKE has so far been demonstrated to be effective for synthesizing programs with performance constraints it can be easily extended to arbitrary intensional constraints. To do so, we describe support for higher-order correctness and intensional constraints as a set of cost functions for STOKE. For each of a number of interesting domains that have previously been out of the reach of general-purpose approaches to program synthesis we define both types of constraints as cost functions that associate lower costs with smaller numbers of correctness errors and a closer match to the desired intensional properties of the resulting code.

Using this approach we are able to demonstrate solutions to several sophisticated program synthesis problems. We consider the implementation of random number generators as representative of a sophisticated higher-order but soft correctness constraint, a shellcode generation task as a demonstration of a sophisticated hard intensional constraint, and the implementation of hash functions as an example of a domain that has both a very sophisticated higher-order soft correctness constraint and a complex *negative* soft constraint – namely that the synthesized functions produce results that are as *different* as possible from a pre-existing implementation.

Although the use of STOKE substantially increases the domain of applicability of program synthesis, many of the programs that we are able to produce cannot be formally verified as correct or able to satisfy the given intensional constraints for all possible inputs. Nonetheless, we stress that this limitation is not fundamental to the use of STOKE but rather due to the complexity of the domains that we consider. To the best of our knowledge there is no accepted technique for guaranteeing the higher-order constraints on program correctness and intensional properties that we describe. As in Chapter 5, we instead demonstrate domain-specific techniques for checking these properties to a high degree of confidence. Although these techniques provide evidence that the desired constraints have been satisfied we stress that they are not proofs. Nonetheless, for domains where correctness and intensional properties cannot be stated formally in tractable logics but rather only defined with respect to representative test suites, this validation method is at least as effective as state of the art.

To the best of our knowledge, none of the application domains that we consider are within the reach of existing general-purpose program synthesis techniques. For each domain, we summarize the relevant design criteria, describe how those criteria are captured in a Stoke cost function, and discuss experimental results. As our experiments demonstrate, Stoke is consistently able to synthesize code sequences that successfully capture the hard and soft constraints of each domain in a way that is competitive with hand-written alternatives. The remainder of this chapter focuses on each domain in turn.

All of the experiments that we describe in this chapter were run on a four core Intel i7 with support for the full Haswell instruction set. Unless otherwise noted Stoke was run with equal weight given to the hard($\cdot$) and soft($\cdot$) constraint terms, a timeout of 5 million proposals, a maximum length of 128 instructions, a uniform transformation proposal distribution, and the annealing constant β set to 1.0. For benchmarks where handwritten assembly was unavailable, targets were generated using `C` programs compiled using `gcc` with full optimizations enabled (compilation using `icc` produced essentially identical code).

7.1 Random Number Generators

High-performance pseudo-random number generators are an important tool in many applications. Numerical simulations, probabilistic programs and modeling algorithms all rely on high-quality random number sequences. Moreover, many high-performance applications require pseudo-random number sequences that are not necessarily cryptographically secure but that instead follow common statistical distributions and exhibit certain properties that can help to prevent bias in host applications.

The standard practice in pseudo-random number generation is based on a two phase process. First a stateful random generator is used to produce samples in a native machine format which is typically a 64-bit value. Second, a distribution routine is used to make one or more calls to that generator and then modify its state to produce a sequence of samples that are representative of the desired distribution. Different applications may require underlying sequences to varying standards of quality; this requirement is exemplified in the Diehard tests for checking sequences of random numbers [66, 68], which is used to enforce a number of strong theoretical properties. Due to the complexity of this process we do not address the problem of generating the random bits themselves but instead focus on the second phase which generates samples according to a desired distribution.

The implementations of many distribution generators — while well-understood — are nevertheless quite involved. The ziggurat algorithm [67] for example, is a rejection sampling method that invokes an underlying generator and then computes transcendental functions over those values and performs repeated lookups into precomputed tables, sometimes discarding results which do not fit the required distribution. Even in high-quality implementations such as Intel's `C` numerics library `libimf` each of these transcendental function calls can execute approximately 100 lines of assembly.

Pseudo-random number generator kernels are representative of a synthesis task with a soft higher-order intensional constraint. Defining correctness in terms of an approximate relationship between distributions is problematic for synthesis techniques that are only able to reason about single program executions. In particular, for this domain there is no requirement that the target and rewrite ever produce the same result. Nonetheless, as we demonstrate below STOKE is well-suited to this task. Given already high-performance random number generator kernels, STOKE can be used to produce even more efficient code that has only slightly weaker distribution properties.

In general, optimizing and reasoning about code that produces samples from a random distribution requires the ability to reason about the distance between distributions. This problem is well-studied and for samples from two known distributions can be solved by comparing their characteristic parameters. For two normal distributions for example, it is possible to solve for the maximum likelihood estimate of mean and variance and compute the difference between the results. For samples of unknown provenance however (as can be expected of the code sequences produced by STOKE) the task of comparing distributions is more complicated.

A well-known method for comparing distributions without a known closed-form representation is the two-sided Kolmogorov-Smirnoff test [51] which takes samples from two distributions and compares the resulting cumulative distribution functions (CDF). Given a code sequence $\mathcal{S}$ and a sequence of machine states τ that represent unique seedings of the underlying random number generator, the empirical CDF can be defined in terms of a sequence of n independently and identically distributed samples as follows:

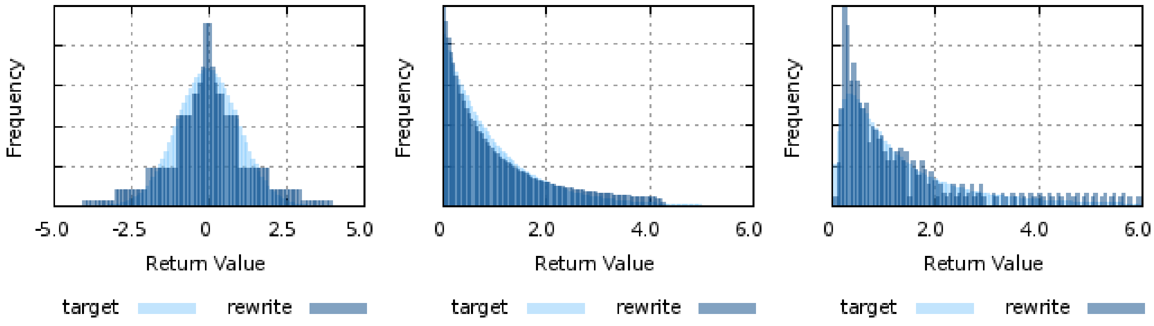
$$\mathrm{cdf}(x;\mathcal{S},\tau)=\frac{1}{|\tau|}\sum_{t\in\tau}\mathbf{1}(\mathrm{ret}_{\mathrm{double}}(\mathcal{S},t)\leq x) \tag{7.1}$$

Using this function, it is possible to define a distance function $D(\cdot)$ that returns the largest difference in height between the two functions. Intuitively, this function can be though of as a comparison of the shape of the two distributions: distributions with similar shapes produce CDFs with only small deviations whereas very different distributions produce CDFs with mass in very different places.

$$D(\mathcal{R};\mathcal{T},\tau)=\sup_{x}\left(\Big|\mathrm{cdf}(x;\mathcal{R},\tau)-\mathrm{cdf}(x;\mathcal{T},\tau)\Big|\right) \tag{7.2}$$

For both the fast and slow variants of the hard$(\cdot)$ term we adopt the portion of the correctness term described in Chapters 4 and 5 that penalizes rewrites which trigger exceptional signal behavior for inputs that the target does not. With the exception of this restriction on program semantics there are no other hard constraints that need to be encoded.

For the fast variant of the soft$(\cdot)$ term we compute the value of the distance function for a small but statistically significant number of samples $n = 2^7$. In similar fashion we define the slow variant



(a) Normal $\mu = 0.0$ $\sigma = 1.0$ (b) Exponential $\lambda = 1.0$ (c) Lognormal $\mu = 0.0$ $\sigma = 1.0$

	Transcendental Function Calls		Samples (μ)		Samples (σ^2)		LOC		Total
Kernel	$\mathcal{T}$	$\mathcal{R}$	$\mathcal{T}$	$\mathcal{R}$	$\mathcal{T}$	$\mathcal{R}$	$\mathcal{T}$	$\mathcal{R}$	Speedup
Normal	`log() sqrt()`	none	2.6	4	9.3	0	106	17	3.53×
Exponential	`log()`	none	2	1	0	0	45	7	2.14×
Lognormal	`log() sqrt()`	none	2.6	4	9.3	0	151	76	3.06×

Figure 7.1: Synthesis results for three random number generator kernels: normal (a), exponential (b) and lognormal (c). The kernels produced by STOKE generate distributions that are a close fit to those produced by the reference implementations (`libstdc++`) and represent a 2–3.5× performance improvement and a substantial reduction in code size. The kernels produced by STOKE require fewer invocations of the underlying random number generator kernel and no invocations of transcendental functions.

of the soft($\cdot$) term in terms of a substantially larger test set of $n' = 2^{20}$ samples. No standard static technique is able to reason precisely about the distributions produced by programs written in a full-fledged hardware instruction set as complex as `x86_64`. As a result, this very large sample size provides as close to a formal guarantee as possible that the constraints on the resulting distribution have been approximately satisfied. For the purpose of simultaneously reasoning about performance we include the latency estimation terms described in Chapter 4.

$$\begin{aligned} \text{soft}_{\text{fast}}(\mathcal{R};\mathcal{T},\tau) &= D(\mathcal{R};\mathcal{T},\tau) + \lambda(\mathcal{R}) \\ \text{soft}_{\text{slow}}(\mathcal{R};\mathcal{T},\tau') &= D(\mathcal{R};\mathcal{T},\tau') + \Lambda(\mathcal{R},\tau') \end{aligned} \tag{7.3}$$

We evaluate our support for random number generation using the GNU implementation of the `C++11 libstdc++` random number generator library. In particular, we consider three random number distribution generators: normal, exponential, and lognormal, each of which are defined in terms of a linear congruential underlying generator. The normal distribution is implemented using the polar method described by Marsaglia, the exponential distribution is implemented in terms of the multiplication of uniform random variates, and the lognormal distribution is implemented in terms of the normal distribution; all three implementations are described in [28].

Figure 7.1 (bottom) summarizes the important properties of these implementations. All three implementations invoke the transcendental `log`(·) function and both normal distributions invoke the `sqrt`(·) function. The distribution over the number of invocations of the underlying generator which accounts for a non-trivial percentage of total kernel runtime has a mean of at least two and a large variance for both normal distributions.

Figure 7.1 (top) compares the distributions produced by the reference implementations to the distributions obtained by running the kernels produced by Stoke using the cost function described above. In all three cases the distributions are a close fit ($D(\mathcal{R}; \mathcal{T}, \tau') \leq 0.05$). Moreover, Figure 7.1 (bottom) shows a substantial improvement in performance for the synthesized kernels. All three perform a fixed (and in the case of exponential, smaller) number of invocations to the underlying random number generator and represent both a substantial reduction in code size and between a 2 and 3.5× speedup over the reference implementation, even when including time spent in invocations of the underlying generator. None of the kernels produced by Stoke perform any invocations of transcendental functions.

7.2 Shellcode

A shellcode is a short piece of machine code that is used to exploit software vulnerabilities [35]; the name derives from the traditional use case which involves starting a shell on a compromised machine and giving an attacker remote access to that system. Shellcodes are generally transmitted to a vulnerable process in small data payloads that can be sent either over a network or supplied as part of a file. An attack typically begins with the exploitation of a vulnerability that gives the attacker control over the victim's program counter — such as a buffer-overflow — and ends with control executing the code contained in the attacker's payload.

User-supplied character strings are a well-known attack vector for shellcode payloads. As a result, many processes will only copy strings up to and including the first instance of a null-terminator character into their address space. Because null-bytes are so common in `x86_64` code sequences, this in combination with other filtering techniques is part of an effective strategy for mitigating the chance that a shell code can execute to completion. For example, systems that provide $W \oplus X$ protection offer the additional constraint that shellcode not be self-modifying. This restriction can discourage the use of tools such as [69], that can embed shellcodes in short sequences of English-like text but which also rely on a self-modifying decoding routine. Nonetheless, by crafting a shellcode either by hand or through the use of a special-purpose `x86_64` compiler to be free of null-bytes, position independent, and as short as possible, an attacker may still be able to subvert these countermeasures.

Shellcodes are representative of a synthesis task with a hard intensional constraint. Defining correctness in terms of both semantic correctness and the absence of null-bytes is problematic for synthesis techniques that are only able to reason about input-output relationships using a high-level intermediate representation. Furthermore, the mapping from `x86_64` assembly to an equivalent bit-wise representation is both non-trivial and difficult to reason about even for experts. Nonetheless, as we demonstrate below STOKE is an effective platform for automatically synthesizing shellcodes that are free of null-bytes. Although there is some potential that this technique could be used for malicious purposes, null-free hand-crafted shellcodes are already widely available on the Internet. Furthermore, by applying STOKE to this domain we hope to stimulate discussion on preventative measures for new developments in shellcode injection attacks.

Given a code sequence $\mathcal{S}$ we define the sequence of bytes that are obtained by invoking an assembler and transforming that program to machine code. Using this formulation, we define functions that count program size in bytes and null-bytes.

$$\begin{aligned} \text{size}(\mathcal{S}) &= |\mathcal{B}| \\ \text{null}(\mathcal{S}) &= \sum_{b \in \mathcal{B}} \mathbf{1}(b = 0) \\ \text{where } \mathcal{B} &= Asm(\mathcal{S}) \end{aligned} \tag{7.4}$$

We define the fast variant of the hard$(\cdot)$ term by augmenting the full eq$(\cdot)$ correctness term described in Chapter 4 with the constraint that rewrites must be free of null-bytes. In contrast to the hard constraints described in the previous section which only used a portion of the eq$(\cdot)$ term, shellcode synthesis requires full bit-wise correctness with respect to the reference implementation. Nonetheless, because the rewrites that we consider are loop-free and neither take any arguments nor dereference heap values they are effectively deterministic. As a result, there is no need to define the slow variant of the hard$(\cdot)$ term in terms of an expensive call to a theorem prover. In fact, for both variants it suffices to check correctness in terms of execution in a single machine state t. In similar fashion we define both variants of the soft$(\cdot)$ term in terms of size$(\cdot)$.

$$\begin{aligned} \text{hard}_{\text{fast}}(\mathcal{R}; \mathcal{T}, \{t\}) &= \\ \text{hard}_{\text{slow}}(\mathcal{R}; \mathcal{T}, \{t\}) &= \text{eq}(\mathcal{R}; \mathcal{T}, \{t\}) + \text{null}(\mathcal{R}) \\ \text{soft}_{\text{fast}}(\mathcal{R}) &= \\ \text{soft}_{\text{slow}}(\mathcal{R}) &= \text{size}(\mathcal{R}) \end{aligned} \tag{7.5}$$

We note briefly that a small modification to STOKE's proposal mechanism is required for successful application to this domain. Specifically, STOKE defines operand moves for immediate constants in terms of a random selection from a small set of pre-defined constants that contains both small

	Null Characters		Size (bytes)	
Code	$\mathcal{T}$	$\mathcal{R}$	$\mathcal{T}$	$\mathcal{R}$
chmod	8	0	28	27
dup2	10	0	23	15
exit	7	0	16	12
write	10	0	44	35
open	7	0	28	24
connect	14	0	53	59
execve	16	0	61	50
mmap	20	0	51	34
socket	13	0	30	25
setuid	7	0	16	14

Figure 7.2: Synthesis results for ten shellcode payloads that are designed to execute Linux system calls. In all cases, Stoke is able to produce rewrites that are free of null-bytes, and in all but one, smaller than the original. The rewrites produced by Stoke are an average of 17% smaller than the reference implementations.

powers of two and whatever immediate constants appear in the target. Because zero-padded constants are a common source of null-bytes we extend this set to include both the immediate constants that appear in the target and small random modifications to those constants. Doing so is sufficient to allow Stoke to propose constants that are free of null-bytes but that require non-trivial computation to transform back to the appropriate value.

We evaluated our support for shellcode generation using a suite of ten shellcode payloads that are known to have been used in buffer-overflow attacks. Each payload is designed to execute a Linux system call and contains a number of null-bytes. These null-bytes often appear as padding in immediate constants, as part of instruction encodings, or to accommodate the need to pass zero as an argument to a system call. Although each benchmark is small the results we describe can be scaled to shellcode sequences of arbitrary length by simply concatenating the results of each experiment. For example, by combining the results for the `setuid`, `socket`, `connect`, `dup`, `execve`, and `exit` benchmarks it is possible to produce a shellcode known as a “reverse-bind shell” payload that connects to an attacker’s system and opens a root shell on a target machine.

Figure 7.2 compares the original shellcode payloads to the shellcodes produced by Stoke using the cost function described above. In all cases Stoke was able to produce rewrites that were free of null-bytes, and in all but one case shorter than the original. On average, the synthesized shellcode were 17% smaller than the reference implementation. Furthermore, the reduction in size was with respect to code size in bytes rather than the more traditional lines of code.

```
# Reference                 # STOKE

# c1 = 0x6e69622f           # c1  = 0x6e69622f
# c2 = 0x68732f             # c2* = 0x168732f

movq 0x3b, rax              movl c2*, -0xc(rsp)
movl c1, -0x10(rsp)         movl esp, edx
movl c2, -0xc(rsp)          movl c1, -0x10(rsp)
leaq -0x10(rsp), rdi        testb dl, spl
movq rdi, -0x18(rsp)        leaq -0x10(rsp), rdi
movl 0x0, -0x20(rsp)        xorq rdx, rdx
movl 0x0, -0x1c(rsp)        movq rdx, -0x20(rsp)
leaq -0x20(rsp), rsi        seto -0x9(rsp)
movq 0x0, rdx               movq rdi, -0x18(rsp)
syscall                     leaq -0x10(rdi), rsi
                            movb 0x3b, al
                            syscall
```

```
48 c7 c0 3b 00 00 00        c7 44 24 f4 2f 73 68 01
c7 44 24 f0 2f 62 69 6e     89 e2
c7 44 24 f4 2f 73 68 00     c7 44 24 f0 2f 62 69 6e
48 8d 7c 24 f0              40 84 d4
48 89 7c 24 e8              48 8d 7c 24 f0
c7 44 24 e0 00 00 00 00     48 31 d2
c7 44 24 e4 00 00 00 00     48 89 54 24 e0
48 8d 74 24 e0              0f 90 44 24 f7
48 c7 c2 00 00 00 00        48 89 7c 24 e8
0f 05                       48 8d 77 f0
                            b0 3b
                            0f 05
```

Figure 7.3: Detailed comparison of a shellcode payload for executing the `execve` Linux system call and the equivalent null-byte free rewrite discovered by STOKE (top). Bit-wise representations of both codes are shown for reference (bottom).

Figure 7.3 (top) shows a detailed comparison of a shellcode payload for invoking the `execve` system call and the null-byte free variant discovered by STOKE. For reference, Figure 7.3 (bottom) shows the corresponding bit-wise encodings. The null-bytes introduced by the use of the 64-bit variant of the `mov` instruction of line 6 of the target are eliminated by the use of the 8-bit variant of the `mov` instruction on line 16 of the rewrite. The null-bytes introduced by the use of the zero constant on lines 11, 12, and 14 of the target are eliminated on line 11 of the rewrite by taking advantage of the invariant that any register value xor'ed with itself will produce a zero value. And finally, the null-byte that appears in the constant `c2` on line 8 of the target is eliminating by loading the slightly different constant `c2*` on line 6 of the rewrite, performing a test that is guaranteed to pass on line 9, and then using the results of that test to force the `seto` instruction on line 13 to zero out the upper 8 bits of `c2*`.

7.3 Hash Functions

Hash functions for user-defined data-types are a key part of applications that use associative data-structures. These applications require hash functions that are both easy to compute and produce values that are uniformly distributed. Although high-quality cryptographically secure hash functions are well-known [83, 94, 30] the performance properties of these algorithms are often prohibitively weak. Furthermore, because many applications do not require hash functions that provide such strong guarantees preference is often given to implementations that instead offer low collisions rates and faster runtimes [43, 44]. In either case, general-purpose hash functions are typically designed to work on contiguous blocks of memory and are insensitive to any of the potentially important design features of domain-specific user-defined types. As a result, hash functions are often written by hand in an ad-hoc fashion to best suit the demands of a particular application.

In addition to these performance constraints another important criterion is that values that compare equal should produce the same hash values. Although this property follows immediately from the strong guarantees provided by cryptographically secure hash functions it is far from obvious for hand-written implementations that are not theoretically motivated. In general, equality of arbitrary data-types in an undecidable problem and as a result many real world applications fall back on a more tractable notion of equality: equality for primitive types is defined in terms of bit-wise equivalence, equality of complex user-defined types is defined inductively in terms of their components, and equality for pointer data-types is defined only for tree-shaped heap structures.

Regardless of the definition of equality that is ultimately used all implementations must make a distinction between the portions of a data-type that are exposed with respect to equality and those which are not. As a simple example, consider two vectors with identical elements and different capacities. Alternately, consider the undefined padding bytes that a compiler emits to guarantee alignment for user-defined data-types. Because both cases are instances of implementation details that are not exposed to the user neither should be considered when computing equality nor should they be exposed to a hash function.

Hash functions are representative of a synthesis task with a higher-order soft intensional constraint. As with random number generators, the constraint that a function produce a uniform distribution over hash values is problematic for synthesis tools that are only able to reason about single program executions. As we describe below, this issue is only compounded by the not uncommon soft correctness constraint that a function be as different as possible from a pre-existing implementation. Regardless, the use of Stoke is sufficient to produce implementations that are both better able to satisfy these constraints and substantially more performant than existing expert implementations.

Given a code sequence $\mathcal{S}$ that computes a hash function and a hash table that contains b buckets we define a function that maps a data-type to a bucket as follows. We say that an input value is represented by a machine state t and define the function that counts the number of collisions that occur in each bucket over a set of test cases τ.

$$\begin{aligned} \text{hash}(t;\mathcal{S},b) &= \text{ret}_{\text{fixed}}(\mathcal{S},t) \bmod b \\ \text{coll}(x;\mathcal{S},\tau,b) &= \sum_{t\in\tau} \mathbf{1}(\text{hash}(t;\mathcal{S},b) = x) \end{aligned} \tag{7.6}$$

Representing input values as test cases allows us to express the distinction between the components of a data-type that are eligible and ineligible for being hashed. Following the example given above, a vector of integers could be represented as a Stoke test case that contained the memory image of the elements but neither the bits that correspond to capacity nor those that correspond to compiler-generated padding.

Using this representation, the definition of both variants of the hard$(\cdot)$ term is identical to the one given in the previous section which discussed random number generators. As before, the only hard constraint on the implementation of a hash function is that it not produce exceptional behavior. Using the definition of equality given above the constraint that identical inputs hash to identical values is implicit.

We define the soft$(\cdot)$ term by augmenting the standard notion of expected runtime with a term that captures the standard deviation in the number of collisions over a set of test cases. A perfect hash function will produce zero standard deviation and distribute hash values across all buckets uniformly. Using the equation shown below we define the fast variant using a small training set τ of 2^8 test case inputs and 16 buckets, whereas for the slow variant we use a larger test set τ' of 2^{20} test cases and 128 buckets. For the reasons discussed in the previous section on random number generators this very large sample size provides as close to a formal guarantee as is possible that the desired constraints have been met.

$$\begin{aligned}\text{soft}_{\text{fast}}(\mathcal{R};\tau) &= \sigma_x\Big(\text{coll}(x;\mathcal{R},\tau,16)\Big) + \lambda(\mathcal{R};\tau)\\ \text{soft}_{\text{slow}}(\mathcal{R};\tau') &= \sigma_x\Big(\text{coll}(x;\mathcal{R},\tau',128)\Big) + \Lambda(\mathcal{R};\tau')\end{aligned} \tag{7.7}$$

As an example of a practical design constraint that often accompanies the design of production hash functions, we additionally consider the issue of randomization. Recent work [11] has demonstrated techniques for automatically generating input values that can cause several widely used hash functions [43, 44] to produce identical values. These techniques can in turn be used to generate denial-of-service attacks on production systems that induce worst-case $O(n)$ behavior in their underlying hash data-structures.

Although improvements in hash function implementation [3] have demonstrated resilience against denial-of-service attacks, there is always the potential for the discovery of new exploits. As a result, an alternate strategy that is often used in practice is to simultaneously deploy a large number of unique hash functions to eliminate the threat that any one exploit can affect more than a small handful of production machines. While this technique is effective it is both time consuming and difficult to implement as it requires an expert to craft not one but many high performance implementations of the same function. In the equation shown below this design strategy is encoded in a term that counts the number of hash function collisions produced by a code sequence $\mathcal{S}$ and a reference implementation $\mathcal{S}'$.

$$\begin{aligned}\text{coll}'(t;\mathcal{S},\mathcal{S}',b) &= \Big(\text{hash}(t;\mathcal{S},b) = \text{hash}(t;\mathcal{S}',b)\Big)\\ C_{\mathcal{S},\mathcal{S}',\tau,b} &= \sum_{t\in\tau} \mathbf{1}(\text{coll}'(t;\mathcal{S},\mathcal{S}',b))\end{aligned} \tag{7.8}$$

Using this new definition, we can augment both variants of the soft$(\cdot)$ term to account for this additional criteria as follows where $\mathcal{S}$ is a preexisting hash function.

$$\begin{aligned}\text{soft}_{\text{fast}}(\mathcal{R};\mathcal{S},\tau) &= \sigma_x\Big(\text{coll}(x;\mathcal{R},\tau,16)\Big)\\ &\quad + \lambda(\mathcal{R};\tau) + C_{\mathcal{R},\mathcal{S},\tau,16}\\ \text{soft}_{\text{slow}}(\mathcal{R};\mathcal{S},\tau') &= \sigma_x\Big(\text{coll}(x;\mathcal{R},\tau',128)\Big)\\ &\quad + \Lambda(\mathcal{R};\tau') + C_{\mathcal{R},\mathcal{S},\tau',128}\end{aligned} \tag{7.9}$$

We evaluated our support for hash function generation using the two user-defined data-types shown in Figure 7.4: a stack data-type that consists of a 32-byte random length zero-padded string, a 32-bit integer, a 64-bit floating-point value, and a boolean flag, and a heap data-type that consists of a randomly sized linked-list of elements of the former type. For comparison we use SipHash [3], a

```
// Stack Data-type        // Heap Data-type

struct S {                struct H {
  char     user[32];        S      val;
  uint32_t id;              Heap* next;
  double   bal;           };
  bool     flag;
};

S stack() {               H* heap() {
  S s;                      H* h = new H();

  size_t len = rand(31);    h.val = stack();
  for (i = 0:len)           h.next = 0;
    s.user[i] = randc();    for (i = 1:rand(4)) {
  for (i = len:32)            H* hh = new H();
    s.user[i] = 0;            hh.val = stack();
  s.id = rand();              hh.next = h;
  s.bal = randd();            h = hh;
  s.flag = randb();         }

  return s;                 return h;
}                         }
```

Figure 7.4: Two representative user-defined data-types: a stack type (left) and a heap type (right). The stack type is a concatenation of randomly generated primitive data-types and the heap type is a randomly sized linked list structure.

denial-of-service resistant hash function that can be initialized with different random seeds to induce substantially different hashing behavior.

Figure 7.5(top) compares the hash functions generated by Stoke using the cost function shown in Equation 7.7 against SipHash-2-4, a variant of SipHash that is designed to maximize performance. Because SipHash is designed to be used on contiguous bytes of memory we apply a modified version to our heap data-type benchmark that first computes the hash of each element in the linked list and then computes a second hash over the resulting values. For both data-types the hash functions generated by Stoke produce both a lower standard deviation and range in the number of observed collisions and perform slightly more than five times faster than the reference implementation.

Figure 7.5(bottom) shows the result of repeating the experiment described above given the additional constraint that collisions with a preexisting hash function be minimized. In this case the function discovered by Stoke was generated using the modified terms shown in Equation 7.9 and setting $\mathcal{S}$ equal to the functions discovered in the initial experiment. In similar fashion, SipHash was reinitialized with a set of parameters. In both cases the functions discovered by Stoke were again slightly more than five times faster than the alternative and resulted in fewer collisions with

Data	Colls.	(σ)	Colls.	(range)	Total
Type	$\mathcal{T}$	$\mathcal{R}$	$\mathcal{T}$	$\mathcal{R}$	Speedup
Stack	94.3	86.4	586	423	5.4×
Heap	87.7	63.1	459	280	5.1×

Data	Colls.	(σ)	Repeats		Total
Type	$\mathcal{T}$	$\mathcal{R}$	$\mathcal{T}$	$\mathcal{R}$	Speedup
Stack'	92.9	83.2	8139	0	5.4×
Heap'	87.7	62.1	8173	7673	5.2×

Figure 7.5: Synthesis results for hash functions for the two user-defined data-types shown in Figure 7.4. The functions generated by Stoke produce a smaller standard deviation and range in number of observed collisions (top) than the reference implementation (SipHash). Similar results are observed under the added constraint that collisions with a pre-existing hash function be minimized (bottom).

respect to the preexisting implementations. In the case of the stack data-type, Stoke was able to achieve a perfect remapping that results in zero collisions with the original.

7.4 Discussion

Although Stoke offers a unique framework for describing both hard and soft instances of higher-order correctness and intensional constraints, there are domains for which the ability to express a constraint does not necessarily guarantee success. As a simple example we briefly consider the steganography synthesis task which involves concealing a piece of machine code within a host image. Whereas cryptography is a method for encrypting information steganography is a method for concealing the fact that that information is being transmitted in the first place. Steganography has many practical uses not the least of which is as an alternate method for transmitting the payloads described in the previous section on shellcode generation.

A straight-forward method for encoding the steganography task as a Stoke cost function is to define the hard constraint of functional correctness in terms of bit-wise equivalence with the target as described in Chapter 4 and to define the soft intensional constraint of obfuscation in terms of the hexadecimal representation of the rewrite. Specifically, we define a function that interprets the sequence of bytes that are obtained by invoking an assembler as a sequence of pixels and then penalizes a rewrite based on the extent to which those pixels differ from the pixels in the host image.

Although Stoke is able to synthesize code sequences that both match the target in terms of correctness and closely match the hexadecimal pixel values in the host image the results are insufficient to prevent a non-expert from distinguishing images that contain a shellcode payload from those that do not. Although pixel error is a soft intensional constraint the resolution of the human perceptual system imposes an extremely tight bound on the quality of acceptable results. As a result, although STOKE is able to rapidly explore a high volume of interestingly distinct rewrites that satisfy the hard correctness constraints of this domain it is unclear how many of those — if any — are capable of adequately satisfying its challenging intensional requirements.

In general we find that STOKE is most effective for synthesis tasks in which the space of code sequences is dense with rewrites that both satisfy the required hard constraints and offer a wide spectrum of varying quality solutions to the desired soft constraints. For domains such as the one described above where most rewrites offer an equally poor solution to the soft intensional constraints STOKE is unable to produce a satisfactory result.

In this chapter, we described a new approach to program synthesis that is able to handle both higher-order correctness constraints and arbitrary intensional constraints on the resulting implementation. In contrast to previous approaches to program synthesis that focus mainly on constraints expressed in decidable logics and search based on SAT/SMT solvers we use STOKE to express constraints as terms in a cost function using a general-purpose programming language and use random search to directly explore the space of `x86_64` code sequences. Because the use of STOKE dramatically increases the domain of applicability for program synthesis many of the results that we are able to produce cannot be guaranteed to satisfy the desired constraints for all possible inputs. Nonetheless, we are able to use data-driven domain-specific techniques to check these constraints to a high degree of confidence.

Our implementation of STOKE is the first instance of a general-purpose synthesis technique that has been successfully applied to the synthesis of high-performance random number generator kernels, null-free payloads for shellcode injection attacks, and hash functions that are resistant to denial-of-service attacks. Nonetheless, further opportunities for improvement remain, particular with respect to synthesis tasks that are characterized by search spaces that contain few instances of rewrites that are able to adequately satisfy challenging soft intensional constraints.

7.5 Related Work

The classic formulation of the program synthesis task [93, 46, 101, 7] requires that an input specification be logical or boolean. Although it is theoretically possible to convert complex specifications such as the description of a hash function to boolean objectives, the resulting specification can be exponentially larger than the resulting program. For specifications that require quantified reasoning over sets (such as those that relate to probability distributions) the use of higher-order logics is required to simply even state the correctness conditions for interesting synthesis problems. As a result, synthesis techniques [53] that do attempt to reason about higher-order correctness constraints both sacrifice expressiveness and restrict themselves to fragments of decidable logic for both specification and implementation language.

For the automata-theoretic synthesis domain [10, 14], quantitative objectives are a necessary design criteria. A well-studied example is the automatic generation of reactive systems such as resource controllers for which response time must be kept to a minimum. In contrast to the generality provided by the classic formulation of the program synthesis task the techniques that have been applied to this domain are limited to weighted-transition finite automata and use solution methods that make use of game theoretic reasoning over graphs. While these techniques are effective for this domain there is no obvious translation from this intermediate representation to one that corresponds to a hardware instruction set, nor is there one from quantitative metrics such as mean payoffs to arbitrary intensional constraints.

Genetic programming techniques have been applied to several of the benchmark domains that we consider in this thesis [32, 59]. Although fitness functions are well-suited to solving soft higher-order constraints, these applications have generally been limited to high-level IRs. It is non-trivial to apply genetic programming to the intensional constraints that Stoke is able to handle. [87] applied genetic programming to binary code repair, and [88] explains the circumstances under which application to similar domains is tractable.

A slightly weaker form of the program synthesis task is the *parameter synthesis* task. Given a partially specified program the goal is to produce initial value assignments to program variables which guarantee that the program meets some purely quantitative objective [17], or one that involves logical constraints as well [15]. While this formulation is closer to the one that we describe in this thesis these systems focus exclusively on properties of the outputs, and because program structure is fixed there is no way to improve upon the low-level implementation of the user-provided program template.

Finally, although it is conceptually possible to formulate the program synthesis task as a probabilistic program and to use inference engines such as Church [42] to sample from programs that satisfy certain probabilistic constraints, the runtime properties of this approach are currently poor. Even in the presence of concessions that trade higher-level operational semantics for a reduced ability to reason about arbitrary intensional constraints it is unlikely that this approach could be applied to benchmarks on the scale that we consider.

Chapter 8

Conclusion

In this thesis, we described a new approach to the aggressive optimization of high-performance code in domains where even a single poorly chosen instruction can lead to unacceptably slow runtimes. Our technique reformulates the competing constraints of correctness and performance improvement as terms in a cost function defined over the space of all code sequences, and recharacterizes the optimization task as a cost minimization problem. Although we sacrifice completeness, the scope of code sequences that we are able to consider, and the quality of the rewrites that we are able produce, far exceed those of previous techniques for low-level binary optimization.

In Chapter 3, we described the design and implementation of Stoke, the first instance of a stochastic optimizer to demonstrate that the high volume application of small random transformations is sufficient for producing very highly optimized code sequences. In the chapters that followed, we showed that the code produced by Stoke is not only capable of outperforming the code produced by production compilers, but in many cases expert hand-written assembly. In Chapter 4, we described the application of Stoke to loop-free fixed-point code sequences which are common in high-performance cryptographic routines, and in Chapter 5 we extended Stoke to the optimization of high-performance floating-point kernels and showed the first application of a stochastic optimizer to full programs. In Chapter 6 we described a data-driven approach for applying Stoke to code sequences that contain non-trivial control flow, and in Chapter 7 we described how Stoke can be used as part of a new approach to program synthesis that is able to handle both higher-order correctness constraints and arbitrary intensional constraints on the resulting implementation.

While the successful application of Stoke to a number of challenging optimization tasks is encouraging, there are still many problems to solve, and directions for future research that we did not discuss in this thesis. Extending Stoke to ever more sophisticated application domains will require techniques for reasoning about longer and more complex code sequences. These could include methods for learning non-trivial transformations and transformation distributions, verification techniques that are capable of checking the equivalence of more sophisticated loop transformations, and search techniques that are more resilient against cost functions that are dense with local minima.

In the introduction, we claimed that our goal was only to convince the reader that our approach to optimization could work. Whether or not the reader is left with the impression that the approach is crazy is up him or her. In the time since we began work on STOKE the number of publications that apply stochastic search techniques to programming language tasks seems to have increased; we can only take this as a hint that in some cases we may have succeeded. We hope that the reader agrees.

Bibliography

[1] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An introduction to mcmc for machine learning. *Machine Learning*, 50(1-2), 2003.

[2] Tamarah Arons, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Lenore D. Zuck. Formal verification of backward compatibility of microcode. In *CAV*, 2005.

[3] Jean-Philippe Aumasson and Daniel J. Bernstein. Siphash: A fast short-input prf. In *INDOCRYPT*, volume 7668 of *Lecture Notes in Computer Science*, 2012.

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4), 1994.

[5] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[6] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[7] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. From relational verification to simd loop synthesis. In *PPOPP*, 2013.

[8] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *PLDI*, 2012.

[9] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28, 2001.

[10] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, 2009.

[11] Martin Boßlet. Breaking murmur: Hash-flooding dos reloaded. 2012.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[13] Pavol Cerný, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In *CAV*, 2011.

[14] Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann, and Rohit Singh. Measuring and synthesizing systems in probabilistic environments. In *CAV*, 2010.

[15] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *POPL*, 2014.

[16] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity and robustness of programs. *Commun. ACM*, 55(8), 2012.

[17] Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. In *PLDI*, 2010.

[18] Zhuliang Chen and Arne Storjohann. A BLAS based C library for exact linear algebra on integer matrices. In *ISSAC*, 2005.

[19] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. Efficient search for inputs causing high floating-point errors. PPoPP, 2014.

[20] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic crosschecking of floating-point and simd code. In *EuroSys*, 2011.

[21] David W. Currie, Alan J. Hu, and Sreeranga P. Rajan. Automatic formal verification of DSP software. In *DAC*, 2000.

[22] Charlie Curtsinger and Emery D. Berger. Stabilizer: statistically sound performance evaluation. In *ASPLOS*, 2013.

[23] Eva Darulova and Viktor Kuncak. On sound compilation of reals. *CoRR*, abs/1309.2511, 2013.

[24] Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Trans. Computers*, 60(2), 2011.

[25] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[26] Leonardo Mendonça de Moura, Nikolaj Bjørner, and Christoph M. Wintersteiger. Z3.http://z3.codeplex.com/.

[27] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *FMICS*, 2009.

[28] Luc Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, 1986.

[29] Isil Dillig, Thomas Dillig, and Alex Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *SAS*, 2010.

[30] D. Eastlake, 3rd and P. Jones. US secure hash algorithm 1 (sha1), 2001.

[31] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3), 2007.

[32] César Estébanez, Julio César Hernández-Castro, Arturo Ribagorda, and Pedro Isasi. Evolving hash functions by means of genetic programming. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1861–1862, 2006.

[33] Xiushan Feng and Alan J. Hu. Automatic formal verification for scheduled VLIW code. In *LCTES-SCOPES*, 2002.

[34] Xiushan Feng and Alan J. Hu. Cutpoints for formal equivalence verification of embedded software. In *EMSOFT*, 2005.

[35] James C. Foster. *Sockets, Shellcode, Porting, and Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals.* Syngress, 2005.

[36] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, 2007.

[37] John Geweke. Priors for macroeconomic time series and their application. Technical report, Federal Reserve Bank of Minneapolis, 1992.

[38] W. R. Gilks. *Markov Chain Monte Carlo In Practice.* Chapman and Hall/CRC, 1999.

[39] Benny Godlin and Ofer Strichman. Regression verification. In *DAC*, 2009.

[40] Benjamin Goldberg, Lenore D. Zuck, and Clark W. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electr. Notes Theor. Comput. Sci.*, 132(1), 2005.

[41] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1), 1991.

[42] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *UAI*, 2008.

[43] Google. The cityhash family of functions. 2011.

[44] Google. Murmurhash. 2011.

[45] Sumit Gulwani. Program analysis using random interpretation. In *Ph.D. Dissertation, UC-Berkeley*, 2005.

[46] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[47] Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, 2012.

[48] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1), 1970.

[49] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Data representation synthesis. In *PLDI*, 2011.

[50] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.

[51] M. Hazewinkel. *Encyclopaedia of mathematics: an updated and annotated translation of the Soviet "Mathematical encyclopaedia"*. Reidel, 1990.

[52] Paul S. Heckbert, editor. *Graphics gems IV*. Academic Press Professional, Inc., 1994.

[53] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, 2010.

[54] Franjo Ivancic, Malay K. Ganai, Sriram Sankaranarayanan, and Aarti Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*, 2010.

[55] Daniel Jackson and David A. Ladd. Semantic Diff: A tool for summarizing the effects of modifications. In *ICSM*, 1994.

[56] Madhav Jha and Sofya Raskhodnikova. Testing and reconstruction of lipschitz functions with applications to data privacy. *Electronic Colloquium on Computational Complexity (ECCC)*, 2011.

[57] Rajeev Joshi, Greg Nelson, and Keith H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, 2002.

[58] Fabienne Jzquel and Jean Marie Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12), 2008.

[59] Hilmi Güneş Kayacik, Malcolm Heywood, and Nur Zincir-Heywood. On evolving buffer overflow attacks using genetic programming. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1667–1674, 2006.

[60] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition.* Addison-Wesley, 1981.

[61] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *CAV*, 2012.

[62] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. FloPSy - search-based floating point constraint solving for symbolic execution. In *ICTSS*, 2010.

[63] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. Automatically adapting programs for mixed-precision floating-point computation. ICS '13, 2013.

[64] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, 2010.

[65] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[66] George Marsaglia and Wai Wan Tsang. Diehard: a battery of tests of randomness. `www.stat.fsu.edu/pub/diehard`, 1996.

[67] George Marsaglia and Wai Wan Tsang. The ziggurat method for generating random variables. `www.jstatsoft.org/v05/i08/paper`, 2000.

[68] George Marsaglia and Wai Wan Tsang. Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3), 2002.

[69] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. English shellcode. In *CCS*, 2009.

[70] Henry Massalin. Superoptimizer - a look at the smallest program. In *ASPLOS*, 1987.

[71] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In *ISQED*, 2006.

[72] Vijay Menon, Keshav Pingali, and Nikolay Mateev. Fractal symbolic analysis. *ACM Trans. Program. Lang. Syst.*, 25(6), 2003.

[73] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In *SAS*, 2011.

[74] George C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.

[75] George C. Necula and Sumit Gulwani. Randomized algorithms for program analysis and verification. In *CAV*, 2005.

[76] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*, 2012.

[77] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. Differential symbolic execution. In *FSE*, 2008.

[78] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, 1998.

[79] Andy Podgurski. Reliability, sampling, and algorithmic randomness. TAV4, 1991.

[80] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008.

[81] David A. Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In *CAV*, 2011.

[82] M. Rinard. Credible compilers. Technical report, Massachusetts Institute of Technology, 1999.

[83] R. Rivest. The MD5 message-digest algorithm, 1992.

[84] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: tuning assistant for floating-point precision. In *SC*, 2013.

[85] Philipp Rümmer and Thomas Wahl. An smt-lib theory of binary floating-point arithmetic. In *SMT*, 2010.

[86] Sriram Sankaranarayanan, Richard M. Chang, Guofei Jiang, and Franjo Ivancic. State space exploration using feedback constraint generation and monte-carlo sampling. In *ESEC/FSE*, 2007.

[87] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 313–316, 2010.

[88] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3):281–312, 2014.

[89] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, 2013.

[90] K. C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. Verification of source code transformations by program equivalence checking. In *CC*, 2005.

[91] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *ESEC/FSE*, 2011.

[92] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[93] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[94] William Stallings. The whirlpool secure hash function. *Cryptologia*, 30(1), 2006.

[95] D. Stott Parker, B. Pierce, and P. R. Eggert. Monte Carlo arithmetic: how to gamble with floating point and win. *Computing in Science & Engineering*, 2(4), 2000.

[96] Enyi Tang, Earl T. Barr, Xuandong Li, and Zhendong Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *ISSTA*, 2010.

[97] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL*, 2009.

[98] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.

[99] A. Turing. Checking a large routine. In *The early British computer conferences*. MIT Press, Cambridge, MA, USA, 1989.

[100] Christoph W. Ueberhuber. *Numerical computation : methods, software and analysis*. Springer-Verlag, 1997.

[101] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[102] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[103] Yichen Xie and Alexander Aiken. Scalable error detection using boolean satisfiability. In *POPL*, 2005.

[104] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin C. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, 2012.