Stochastic Optimization of Floating-Point Programs with Tunable Precision

Eric Schkufza
Stanford University
eschkufz@cs.stanford.edu

Rahul Sharma
Stanford University
sharmar@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

The aggressive optimization of floating-point computations is an important problem in high-performance computing. Unfortunately, floating-point instruction sets have complicated semantics that often force compilers to preserve programs as written. We present a method that treats floating-point optimization as a stochastic search problem. We demonstrate the ability to generate reduced precision implementations of Intel's handwritten C numeric library which are up to 6 times faster than the original code, and achieve end-to-end speedups of over 30% on a direct numeric simulation and a ray tracer by optimizing kernels that can tolerate a loss of precision while still remaining correct. Because these optimizations are mostly not amenable to formal verification using the current state of the art, we present a stochastic search technique for characterizing maximum error. The technique comes with an asymptotic guarantee and provides strong evidence of correctness.

Categories and Subject Descriptors D.1.2 [*Automatic Programming*]: Program Transformation; D.3.4 [*Processors*]: Optimization

General Terms Performance

Keywords 64-bit; x86; x86-64, Binary; Markov Chain Monte Carlo; MCMC; Stochastic Search; SMT; Floating-Point; Precision

1. Introduction

The aggressive optimization of floating-point computations is an important problem in high-performance computing. For many applications, there is considerable value in being able to remove even a single instruction from a heavily used kernel. Unfortunately, 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.

PLDI '14, June 9–11, 2014, Edinburgh, UK.
Copyright 2014 ACM 978-1-4503-2784-8/14/06... $15.00.
http://dx.doi.org/10.1145/2594291.2594302

Floating-point instruction sets have complicated semantics. The ability to represent an enormous range of values in a small number of bits comes at the cost of precision. Constants must be rounded to the nearest representable floating-point value, basic arithmetic operations such as addition and multiplication often introduce additional rounding error, and as a result, even simple arithmetic properties such as associativity do not hold. Because reordering instructions can produce wildly different results, compilers are forced to preserve floating point programs almost as written at the expense of efficiency.

In most cases a programmer's only mechanism for communicating precision requirements to the compiler is through the choice of data-type (for example, `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 typical 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 tedious and error prone and well beyond the abilities of the average programmer.

We present a method for overcoming these issues by treating floating-point optimization as a stochastic search problem. Beginning from floating-point binaries produced either by a production compiler or written by hand, we show that through repeated application of random transformations it is possible to produce high-performance optimizations that are specialized both to the range of live-inputs of a code sequence and the desired precision of its live-outputs. While any one random transformation is unlikely to produce an optimization that is also correct, the combined result of tens of millions of such transformations is sufficient to produce novel and often non-obvious optimizations which very likely would otherwise be missed.

We have implemented our method as an extension to STOKE, a stochastic optimizer for loop-free sequences of fixed-point x86-64 programs (note that the original implementation of STOKE did not handle floating-point optimizations) [29]. Our system is the first implementation of a stochastic optimizer to take advantage of a JIT-assembler for the full 64-bit x86 instruction set and is capable of achieving a search throughput which outperforms the original implementation of STOKE by up to two orders of magnitude. Using our system 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. Our system is also the first implementation of a stochastic optimizer to demonstrate the optimization of full programs. 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 which can tolerate a loss of precision while retaining end-to-end correctness.

The aggressive nature of the optimizations produced by our method creates a difficulty in checking the correctness of the resulting codes. There are two possible approaches using currently known static verification techniques. Unfortunately neither is capable of verifying some of 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, and 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 a possible alternative, but the guarantees offered by randomized testing 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 the 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.

We present a novel randomized method which 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 correctness.

Using our technique, we are able to establish upper bounds on the imprecision of an optimization which 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 successive random testing that is orders of magnitude more intensive than the effort expended on validation. Although there exist optimizations that we do not expect our technique to perform adequately on, we believe that the results we obtain are representative of the potential for a considerable improvement over current practice.

The remainder of this paper is structured around the primary contributions of our work. Section 2 provides essential background and terminology, Section 3 describes the theory necessary to extend STOKE to the optimization of floating-point programs, Section 4 describes our novel validation technique for checking the equivalence of two loop-free floating point programs with respect to an upper bound on loss of precision, Section 5 discusses some of the novel implementation details of our extensions to STOKE, Section 6 summarizes experiments with real-world high-performance floating-point codes and explores the potential for alternate implementations of STOKE, and Section 7 includes a discussion of related work. We conclude with directions for future research.

2. Background

We begin with a high-level overview of MCMC sampling and the design of STOKE. Unfortunately, a complete treatment of either is beyond the scope of this paper, and the reader is referred to [1, 29, 30] for more information.

2.1 MCMC Sampling

MCMC sampling is a randomized technique for drawing samples from distributions for which a closed-form representation is either complex or unavailable. In fact, for many distributions, MCMC is the only known general sampling method which is also tractable. We make heavy use of one of its most important theoretical properties.

Theorem 1. *In the limit,* MCMC *sampling is guaranteed to take samples from a distribution in proportion to its value.*

A useful corollary of this theorem is that regions of high probability are sampled more often than regions of low probability. A common mechanism for constructing Markov chains with this property is the Metropolis-Hastings algorithm. The algorithm maintains a current sample, x, and *proposes* a modified sample x^* as the next step in the chain. The proposal is then either accepted or rejected. If x^* is accepted, it becomes the current sample, otherwise it is discarded and a new proposal is generated from x.

We say that proposals are *ergodic* if every point in the search space can be transformed into any other point by some sequence of proposals; that is, the search space is connected. If the distribution of interest is $p(x)$, and $q(x \to x^*)$ is the probability of proposing x^* when x is the current sample, then if the proposals are ergodic, the Metropolis-Hastings probability of accepting a proposal is defined as

$$\alpha(x \to x^*) = \min\left(1, \frac{p(x^*) \cdot q(x^* \to x)}{p(x) \cdot q(x \to x^*)}\right) \quad (1)$$

Although many distributions $q(\cdot)$ satisfy this property, the best results are obtained for distributions with variance that is neither too high, nor too low. Intuitively, $p(\cdot)$ is best explored by proposals that strike a balance between local proposals that make small changes to x and global proposals that make large changes to x. Although MCMC sampling has been successfully applied to a number of otherwise intractable problem domains, in practice the wrong choice of proposal distribution can lead to convergence rates that may only be slightly better than pure random search. As a result, identifying an effective proposal distribution often requires experimentation and is typically a critical precondition for success.

Because for many functions it is difficult to compute $q(x \to x^*)$ given $q(x^* \to x)$, $q(\cdot)$ is often chosen to be *symmetric* (such that $q(x \to x^*)$ and $q(x^* \to x)$ are equal). In this case, the probability of acceptance can be reduced to the much simpler *Metropolis ratio* by dropping $q(\cdot)$ from Equation 1.

2.2 STOKE

STOKE [29, 30] is a stochastic optimizer for fixed-point x86-64 binaries. It is capable of producing optimizations which either match or outperform the code produced by `gcc` and `icc` with full optimizations enabled, and in some cases, expert hand-written assembly. STOKE is an application of MCMC sampling to a combinatorial optimization problem. Given an input program (*the target*), $\mathcal{T}$, STOKE defines a cost function over potential optimizations (*rewrites*) $\mathcal{R}$.

$$c(\mathcal{R}; \mathcal{T}) = \text{eq}(\mathcal{R}; \mathcal{T}) + k \cdot \text{perf}(\mathcal{R}; \mathcal{T}) \tag{2}$$

The notation $f(x; y)$ is read "f is a function that takes x as an argument and is *parameterized* (that is, defined in terms of) y". $\text{eq}(\cdot)$ encodes the fact that $\mathcal{R}$ should be functionally equivalent to $\mathcal{T}$, $\text{perf}(\cdot)$ is a measure of the performance difference between $\mathcal{R}$ and $\mathcal{T}$, and the non-negative constant, k, is used to weight the relative importance of the terms. In the special case where $k = 0$, STOKE is said to run in *synthesis mode*, as the elimination of the $\text{perf}(\cdot)$ term renders it insensitive to the performance properties of a rewrite. For all other values of k, STOKE is said to run in *optimization mode*.

The cost function $c(\cdot)$ is transformed into a probability density function $p(\cdot)$, using the following equation, where β is a chosen *annealing constant* and Z normalizes the distribution.

$$p(\mathcal{R}; \mathcal{T}) = \frac{1}{Z} \exp\Big(-\beta \cdot c(\mathcal{R}; \mathcal{T}) \Big) \tag{3}$$

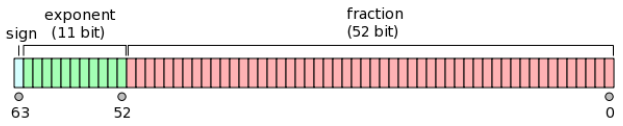
Given this transformation, samples from $p(\cdot)$ correspond to rewrites. Proposals are then generated using the following definition of $q(\cdot)$ as a random choice of any of the following transformations, which are both ergodic and symmetric.

- **Opcode.** An instruction is randomly selected, and its opcode is replaced by a random valid opcode.
- **Operand.** An instruction is randomly selected and one of its operands is replaced by a random valid operand.
- **Swap.** Two lines of code are randomly selected and interchanged.
- **Instruction.** An instruction is randomly selected and replaced either by a random instruction or the UNUSED token. Proposing UNUSED corresponds to deleting an instruction, and replacing UNUSED by an instruction corresponds to inserting an instruction.

Combining Equations 1 and 3 yields the acceptance probability used by STOKE when considering a potential rewrite

$$\begin{aligned} \alpha(\mathcal{R} \to \mathcal{R}^*; \mathcal{T}) &= \min\left(1, \exp(-k)\right) \\ \text{where } k &= \beta \cdot \big(c(\mathcal{R}^*; \mathcal{T}) - c(\mathcal{R}; \mathcal{T}) \big) \end{aligned} \tag{4}$$

STOKE begins with either the target or a random rewrite and forms a Markov chain of samples from $p(\cdot)$ by repeatedly proposing modifications using $q(\cdot)$ and evaluating $\alpha(\cdot)$ to compute the probability that the proposal should be accepted. By Theorem 1, in the limit most of these samples are guaranteed to be taken from rewrites that correspond to the lowest cost, best optimizations. However well before that limiting behavior is observed, STOKE takes advantage of the fact that MCMC sampling also functions as an efficient hill-climbing method which is robust against local maxima; all rewrites, $\mathcal{R}^*$, no matter how poor the ratio $p(\mathcal{R}^*)/p(\mathcal{R})$ is, have some chance of being accepted. In fact, it is precisely this property that makes STOKE effective. STOKE is often able to discover novel, high quality rewrites, which other compilers miss, by temporarily experimenting with shortcuts through code that is either non-performant or incorrect.



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	$[1, 2046]$	dc	$(-1)^s \cdot 2^{e-1023} \cdot 1.\text{f}$
Infinity	2047	0	$(-1)^s \cdot \infty$
NaN	2047	$\neq 0$	$(-1)^s \cdot \perp$

Figure 1. IEEE-754 double-precision floating-point format. Bit patterns marked dc (don't care) are unconstrained.

Randomized sampling techniques are generally effective only if they are able to maintain a high throughput rate of proposals. STOKE addresses this issue by using a two-tiered equality check in its cost function. A fast, unsound equality check based on test cases, τ, is used to rapidly identify incorrect rewrites, whereas a slower, sound equality check is reserved for those which pass the fast check.

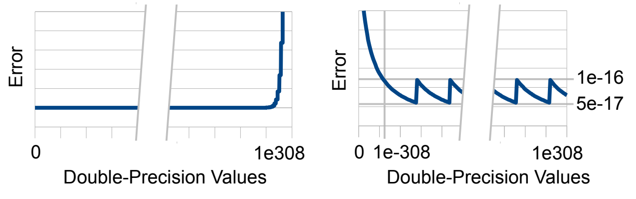
$$\text{eq}(\mathcal{R}; \mathcal{T}, \tau) = \begin{cases} \text{eq}_{\text{slow}}(\mathcal{R}; \mathcal{T}) & \text{eq}_{\text{fast}}(\mathcal{R}; \mathcal{T}, \tau) = 0 \\ \text{eq}_{\text{fast}}(\mathcal{R}; \mathcal{T}, \tau) & \text{otherwise} \end{cases} \tag{5}$$

For floating-point, these equality checks are inappropriate, and we develop suitable fast and slow equality checks in the following two sections. We use the following notation. We say a test case, t, is a function from live-in hardware locations to values. $\rho(\cdot)$ is a function which returns the set of registers which are live-out in $\mathcal{T}$, $\mu(\cdot)$ is defined similarly for memory locations, and $\ell(\mathcal{T}) = \rho(\mathcal{T}) \cup \mu(\mathcal{T})$. We say that $\text{eval}(l, \mathcal{P}, t)$ is a function which returns the value stored in register or memory location l after executing a program $\mathcal{P}$ (target or rewrite) on a test case, t. And, finally, we define the indicator function, $\mathbf{1}\{\cdot\}$, over boolean variables which returns 1 for true, and 0 for false.

3. Search

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 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. Additionally, 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 which 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 hold in general. As a result,



(a) Absolute Error (b) Relative Error

Figure 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 optimizers are often forced to preserve programs as written, and abandon aggressive optimization.

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

$$\begin{aligned} \text{abs}(r_1, r_2) &: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} = |r_1 - r_2| \\ \text{rel}(r_1, r_2) &: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} = \left|\frac{r_1 - r_2}{r_1}\right| \end{aligned} \tag{6}$$

It turns out that there are serious problems with using either absolute or relative error. Figure 2 shows the results obtained when $\text{abs}(\cdot)$ 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.

$$\begin{gathered} \text{ULP}(f, r) : \mathbb{F} \times \mathbb{R} \rightarrow \mathbb{R} = \left| d_1.d_2 \ldots d_p - \frac{r}{\beta^e} \right| \beta^{p-1} \\ \text{where } f \equiv d_1.d_2 \ldots d_p \cdot \beta^e \end{gathered} \tag{7}$$

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 [35] 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 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.

$$\forall r \exists f. \left(\frac{1}{2}\beta^{-p} \leq \frac{1}{2}\text{ULP}(f, r) \leq \frac{\beta}{2}\beta^{-p} \right) \tag{8}$$

We use this definition of rounding error to extend STOKE's fast correctness check as follows. STOKE defines testcase error with respect to the weighted sum of three terms, which respectively represent errors appearing in registers, errors appearing in memory, and divergent signal behavior. The last term is effectively an indicator function which returns 1 if the rewrite raises a signal and the target either returns normally or raises a different signal. Intuitively, smaller values are indicative of rewrites that are "more correct".

$$\begin{aligned} \text{err}_{\text{fast}}(\mathcal{R}; \mathcal{T}, t, \eta) \;&= w_r \cdot \text{reg}(\mathcal{R}; \mathcal{T}, t, \eta) + \\ &\quad w_m \cdot \text{mem}(\mathcal{R}; \mathcal{T}, t, \eta) + \\ &\quad w_s \cdot \text{sig}(\mathcal{R}; \mathcal{T}, t) \end{aligned} \tag{9}$$

Of these three, 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 which is sensitive to the type (fixed- or floating-point) of the value stored in a particular location. 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. This function can be parameterized by a user-defined constant η which defines the minimum unacceptable ULP rounding error. All values in excess of this bound are scaled down toward zero, and all values below this bound are discarded.

$$\begin{gathered} \text{reg}(\mathcal{R}; \mathcal{T}, t, \eta) = \sum_{r \in \rho(\mathcal{T})} \max\Big(0, \text{ULP}\big(f_{\mathcal{R}}, f_{\mathcal{T}}\big) - \eta\Big) \\ \text{where } f_{\mathcal{R}} = \text{eval}(\mathcal{R}, t, r),\ f_{\mathcal{T}} = \text{eval}(\mathcal{T}, t, r) \end{gathered} \tag{10}$$

We lift the definition of test case correctness to sets using a reduction operator $\oplus$. Although the original implementation of STOKE defined this operation to be summation, nothing prevents the use of other functions. We note briefly that because the $\text{sig}(\cdot)$ term in Equation 9 is not parameterized by η, this function is guaranteed to return positive values for any set of test cases that exposes divergent signal behavior.

$$\text{eq}_{\text{fast}}(\mathcal{R}; \mathcal{T}, \tau, \eta) = \bigoplus_{t \in \tau} \text{err}_{\text{fast}}(\mathcal{R}; \mathcal{T}, t, \eta) \tag{11}$$

We discuss the reduction operator used in our implementation in Section 5.2.

As with STOKE's fixed-point error function, 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 which the user considers insignificant and not worth penalizing.

4. Verification and Validation

STOKE defines slow, formal correctness, with respect to a sound routine for checking the equality of the target and rewrite. In general, proving program equivalence is a very hard problem, and as such there is no requirement that the routine be complete.

$$\text{eq}_{\text{slow}}(\mathcal{R}; \mathcal{T}, \eta) = 1 - \Big(\mathbf{1}\big\{\text{verif}(\mathcal{R}; \mathcal{T}, \eta)\big\}\Big) \tag{12}$$

The original implementation of STOKE 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 a verification routine as well. We say that a target and rewrite are equal if there does not exist a test case, t, which exposes an ULP error in excess of η. We define $\top_\eta$ to be larger than all η to ensure that divergent signal behavior is always exposed.

$$\begin{aligned} \text{err}(\mathcal{R}; \mathcal{T}, t) = \sum_{l \in \ell(\mathcal{T})} &\text{ULP}\Big(\text{eval}(\mathcal{R}, t, l), \text{eval}(\mathcal{T}, t, l)\Big) + \\ &\top_\eta \cdot \text{sig}(\mathcal{R}; \mathcal{T}, t) \end{aligned} \tag{13}$$

$$\text{verif}(\mathcal{R}; \mathcal{T}, \eta) = \neg\exists t. \Big(\text{err}(\mathcal{R}; \mathcal{T}, t) > \eta\Big) \tag{14}$$

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 [27] is a new standard for decision procedures which contains support for floating-point operations, however it is only now beginning to be widely adopted. Z3's implementation [10], for example, is based on bit-blasting and does not scale beyond programs containing a very small number of instructions [8]. Other approaches to deciding floating-point formulas are unable to handle code sequences that interleave fixed-point bit-vector and floating-point arithmetic [15, 17]. 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 [6] is limited.

The abstract domains used for floating-point reasoning in methods based on abstract interpretation, such as those in [11], 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, we note that for programs which perform exclusively floating-point computations it is possible to bound, if coarsely, the absolute error between two floating-point programs [8].

Where neither of these approaches are appropriate, some work has been done in providing guarantees based on randomized testing [24]. Unfortunately, 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 propose using MCMC sampling to draw values from the $\texttt{err}(\cdot)$ function. The idea is simple: our goal is to identify a test case, t, that will cause $\text{err}(\cdot)$ to produce a value greater than η. Although, in general, we cannot produce a closed form representation of $\text{err}(\cdot)$, sampling from it is straightforward: we simply evaluate $\mathcal{T}$ and $\mathcal{R}$ on t and compare the results. By Theorem 1, 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 modified version of Equation 14, which is based on the iterative evaluation of the samples taken from the sequence of proposals $t_0, t_1, \ldots, t_\infty$.

$$\text{validate}(\mathcal{R};\mathcal{T},\eta) = \max\left(\text{err}(\mathcal{R};\mathcal{T},t_i)\right)_{i=0}^{\infty} \le \eta \tag{15}$$

where the proposal distribution, $q(\cdot)$, is defined as follows. Recall that a test case contains values for each of a function's live inputs. As a result, our definition applies to functions which take an arbitrary number of arguments. 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. Doing so 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 $q(\cdot)$ follow from the properties of a normal distribution.

$$\begin{aligned} t_{i+1} &= \left\{(l,v) \mid l \in Dom(t_i), v = v'\right\} \\ \text{where } v' &= \begin{cases} v & t_i(l) + x < l_{min} \\ v & t_i(l) + x > l_{max} \\ t_i(l) + x & \text{otherwise} \end{cases} \\ \text{and } x &\sim \mathcal{N}(\mu,\sigma) \end{aligned} \tag{16}$$

A remaining issue is the definition of a termination condition that can be used to determine when we can 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 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 which result from 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 which contain all local maxima, and as a result, contains the global maximum. Metrics for determining whether a chain has mixed well are well known and available in many statistical computing packages [25]. We present the particular statistical test that we use in our implementation in Section 5.3.

As discussed in Section 1, we call our 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 $\text{sig}(\cdot)$ term in Equation 13 has the potential to introduce discontinuities that may be very difficult to discover. Thus, $\text{validate}(\cdot)$ 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. Implementation

Our implementation of STOKE is based on the notes in [29, 30] and uses all of the optimizations described therein. We discuss only the most relevant details of our extensions below.

5.1 JIT Assembler

Our implementation of STOKE evaluates rewrites using a JIT assembler which supports the entire x86-64 instruction set available on Intel's Haswell processors: over 4000 variants of 400 distinct opcodes including Intel's newest fused multiply-add, AVX, and advanced bit manipulation instructions. Our test case evaluation framework supports full sandboxing for instructions which dereference memory and is two orders of magnitude faster that the implementation of STOKE described in [29], which relies on an x86-64 emulator. At peak throughput, our implementation is able to dispatch test cases at a rate of almost 1 million per second on a single core. This performance improvement allows us to duplicate the synthesis results reported in [29] in at most 5 minutes on a single four core machine as opposed to a half hour on 80 cores.

5.2 Cost Function

The computation of ULP rounding error shown in Equation 7, which is defined in terms of the 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. In particular, we note that this function has the advantage of producing only integer results which greatly simplifies the program logic associated with cost manipulation.

$$\mathrm{ULP}'(f_1, f_2) : \mathbb{F} \times \mathbb{F} \to \mathbb{N} = \Big|\big\{x \in \mathbb{F} \mid f_1 < x \leq f_2\big\}\Big| \quad (17)$$

Figure 3 shows the C implementation of this function for double-precision values. The floating-point representation shown in Figure 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.

In contrast to the original implementation of STOKE, our definition of test case correctness defines $\oplus$ using the $\max(\cdot)$ operator in place of summation. 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 correctness costs never exceed ULLONG_MAX.

5.3 Validation

Many techniques are known for checking whether a Markov chain is well mixed. We use the Geweke diagnostic test [12], 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

$$\begin{aligned}\theta_1 &= \{\mathrm{err}(\mathcal{R}; \mathcal{T}, t_i) : i = 1, \ldots, n_1\} \\ \theta_2 &= \{\mathrm{err}(\mathcal{R}; \mathcal{T}, t_i) : i = n_a, \ldots, n\} \\ &\text{where } 1 < n_1 < n_a < n \text{ and } n_2 = n - n_a + 1\end{aligned} \quad (18)$$

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.

$$Z_n = \frac{\bar{\theta}_1 - \bar{\theta}_2}{\sqrt{\frac{\hat{s}_1(0)}{n_1} + \frac{\hat{s}_2(0)}{n_2}}} \qquad \text{where } \bar{\theta}_1 = \frac{1}{n_1}\sum \theta_1 \text{ and } \bar{\theta}_2 = \frac{1}{n_2}\sum \theta_2 \quad (19)$$

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 simply continue to sample from err$(\cdot)$, and recompute Z_n as necessary. Once Z_n achieves a sufficiently small value, we conclude that the chain it approximately equal to a stationary distribution, and return the largest observed sample as a bound on ULP rounding error between the target and rewrite.

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

6. Evaluation

We evaluate 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 is 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.

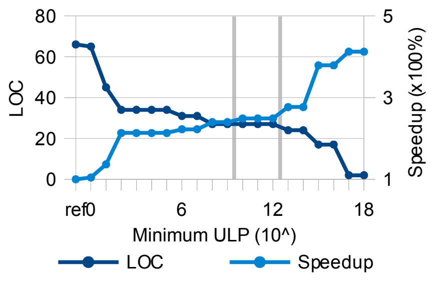
All experiments were run on a four core Intel i7 with support for the full Haswell instruction set. For benchmarks where handwritten assembly was unavailable, targets were generated using gcc with full optimizations enabled (icc produced essentially identical code). STOKE was run in optimization mode with k set to 1.0, using 16 search threads, 1024 test cases, and a timeout of 10 million proposals. 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 every benchmark, STOKE search threads typically 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 compare the results against our validation method whenever it is possible to do so.

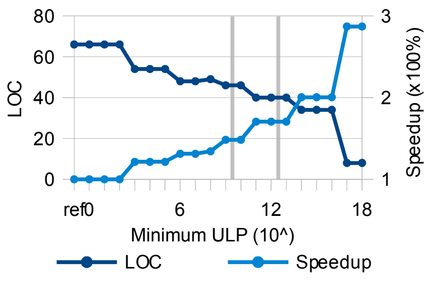
6.1 libimf

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.

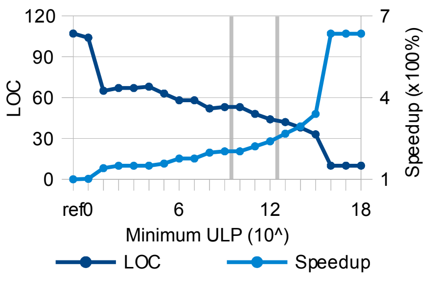
Figure 4(a-c) shows the results obtained by running STOKE on three representative kernels: a bounded periodic function (sin()), a continuous unbounded function (log()), and a discontinuous unbounded function (tan()). For brevity, we have omitted results for cos() and exp(), which are similar. The library 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



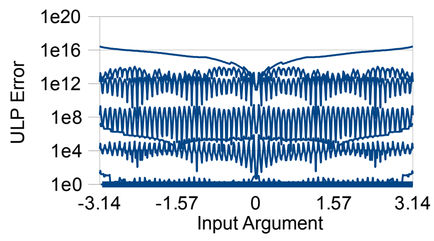
(a) LOC/speedup for `sin()`



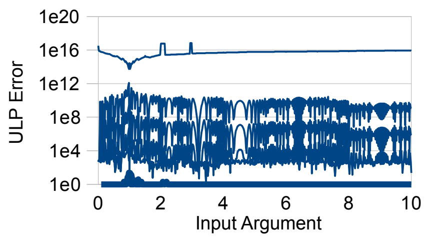
(b) LOC/speedup for `log()`



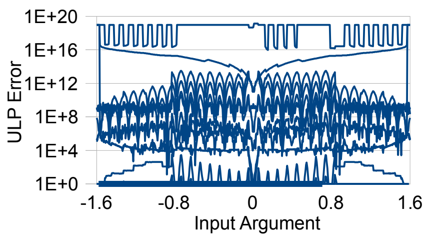
(c) LOC/speedup for `tan()`



(d) ULP error for `sin()` rewrites



(e) ULP error for `log()` rewrites



(f) ULP error for `tan()` rewrites

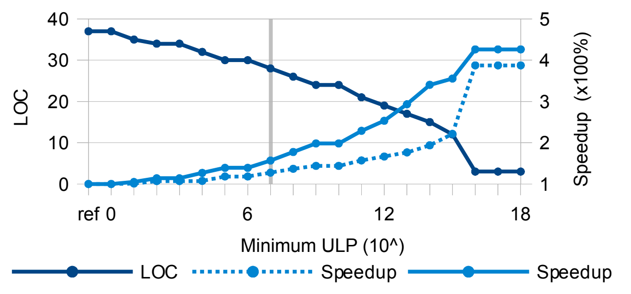
Figure 4. Representative kernels from Intel's implementation of `math.h`. Increasing η produces rewrites which interpolate between double- single- and half-precision (vertical bars, shown for reference) (a-c). Errors introduced by reduced precision are well-behaved (d-f) and amenable to MCMC sampling.

deleting an instruction does not correspond to deleting a term and will usually produce wildly inaccurate results.

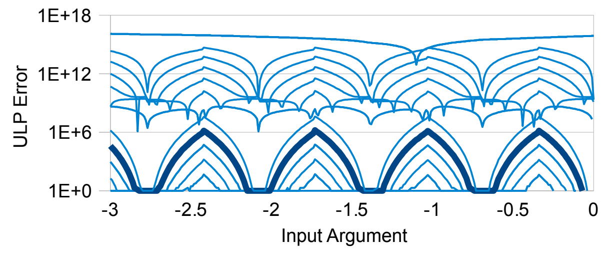
Reference points are given on the far left of each plot in Figure 4. The original kernels range in length from 66 to 107 lines of code (LOC), and represent a baseline speedup of 1x. Figures 4(a-c) show the results of varying η between 1 and 10^{18}, approximately the number of representable double-precision 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 is 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 is able to remove nearly all instructions (for η = `ULLONG_MAX`, not shown, STOKE produces the empty rewrite), and produce speedups of up to 6x over the original implementations. However performance improvements grow smoothly and are still significant for reasonable precisions. Although no longer available, Intel at one point offered a variable-precision implementation of `libimf`, which allowed the user to trade performance for higher quality results. Using only the full double-precision implementation, STOKE is 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, 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



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



(b) ULP error for `exp()` rewrites

Figure 5. The diffusion leaf task from the S3D direct numeric simulation solver. Increasing η allows STOKE to discover rewrites for the `exp()` kernel which trade precision for shorter code and higher performance (a). The diffusion leaf 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).

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.

```
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 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.

Our MCMC sampling method for characterizing maximum error works well on these kernels. Figure 4(d-f) shows the error function for each of 20 rewrites depicted in Figure 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 is quickly able to determine their maximum values.

6.2 S3D

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 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. In fact 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. Specifically, we considered 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 in the previous section, Figure 5 shows the result of varying η between 1 and 10^{18}. By decreasing precision, STOKE is able to discover rewrites which are 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(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%.

As in the previous section, Figure 5(b) shows the error function for each of the 20 rewrites shown in Figure 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.

```
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 7. 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.

	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 8. 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.

6.3 aek

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 [16]. 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 in order 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. An unfortunate consequence of this fact is that STOKE is forced to obey the program-wide data structure layout chosen by `gcc`. Consider the vector dot product kernel shown in Figure 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 precludes 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 is too great. Nonetheless, STOKE is able to discover an optimization which 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. For brevity, we omit discussion of the remaining vector kernels which have similar properties and produce the performance improvements summarized in Figure 8.

By focusing only on vector kernels, STOKE is 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 7 shows the result of applying STOKE to the kernel which perturbs the camera angle. STOKE is able to discover a rewrite that takes advantage of the bit-imprecise associativity of multiplication and drops terms which 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.

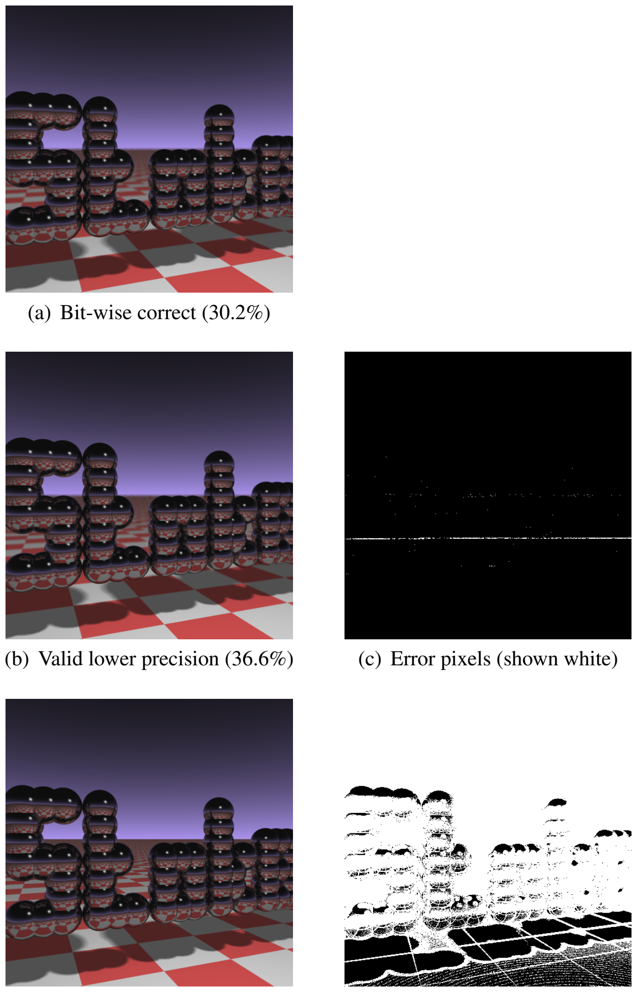
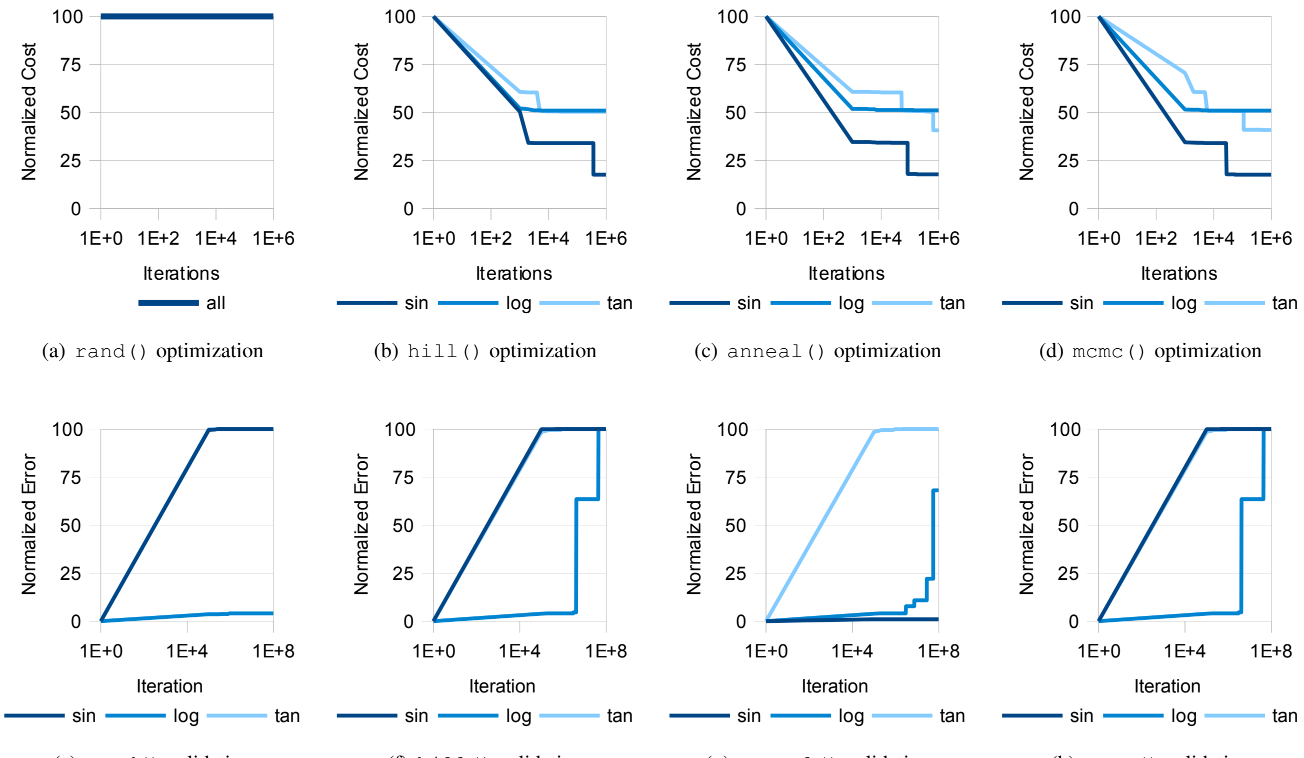

(a) Bit-wise correct (30.2%)
(b) Valid lower precision (36.6%)
(c) Error pixels (shown white)
(d) Invalid lower precision
(e) Error pixels (shown white)

Figure 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.

Figure 9 summarizes the cumulative effects of optimizations discovered by STOKE. Figure 9(a) shows the image which is generated using only bit-wise correct optimizations, whereas Figure 9(b) shows the image which is generated using imprecise optimizations as well. Although the images appear identical they are in fact different as show in Figure 9(c). Further optimizations to the kernel which 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 which exceed the noise introduced by randomness, STOKE removes the code which produces the camera perturbation altogether. The result is an image without depth of field blur which is depicted in Figure 9(d). As shown in Figure 9(e), the image differs dramatically from the original. (This effect may be difficult to observe in



(a) `rand()` optimization (b) `hill()` optimization (c) `anneal()` optimization (d) `mcmc()` optimization

(e) `rand()` validation (f) `hill()` validation (g) `anneal()` validation (h) `mcmc()` validation

Figure 10. Comparison of alternate implemenatations of the search procedure used by STOKE for optimization and validation: random search (`rand()`), greedy hill-climbing (`hill()`), simulated annealing (`anneal()`) and the MCMC sampling procedure described above (`mcmc()`). MCMC sampling outperforms the alternatives for optimization, but is not clearly better-suited to validation.

printed copy; note in particular the absence of depth of field blur on the horizon.)

6.4 Alternate Search Implementations

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 in Section 6.1.

Figure 10(a-d) 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 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 likely sufficient to guarantee that a random walk never returns 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. Intuitively, 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 10(e-h) 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 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, though 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 4(d-f) which are dense with sharp peaks. Without a mechamism for tracking towards local maxima, we do not expect that a purely random search procedure should be able to produce consistently good results. Finally, we note that 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. Nonetheless, this observation may be an artificat of the relative complexity of the two. The space of all 64-bit x86 programs 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.

7. Related Work

We are not the first to propose program transformations which sacrifice bit-wise precision in favor of performance. *Loop perforation* dynamically removes loop iterations to balance an efficiency-precision trade-off [31]. Given an input distribution, some perforations can be statistically guaranteed [22]. However, such guarantees are statistically unsound for any other input distribution. Green [2] and other systems for performing approximate computation [36] allow a programmer to provide multiple implementations of the same algorithm, each of which represent different precision/power trade-offs. These compilers emit runtime code which 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 [14, 32] are currently inapplicable to this task as they rely on decision procedures which 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 Section 4. A method for proving robustness and continuity of programs is presented in [4] and a black box testing approach for establishing Lipschitz continuity is described in [18]. Unfortunately, many high-performance kernels such as $\texttt{exp}(\cdot)$ are not Lipschitz continuous, and functions such as $\texttt{tan}(\cdot)$ are not even continuous. Thus the applicability to STOKE is limited. Interactive theorem provers such as [9] may be applicable to the non-trivial optimizations discovered by STOKE, but the result would be far from a fully automatic system.

Random interpretations [23] can provide strong statistical guarantees of program behavior. However, the number of samples required to produce these guarantees depends crucially on the particular operators used by a program. Unfortunately, bit-extraction operators for example, require an intractable number of samples. Additional testing infrastructures for floating-point code that have no statistical guarantees include [3, 20]. Monte Carlo testing approaches for checking numerical stability, which is a concern orthogonal to the correctness of optimizations, include [19, 33, 34]. MCMC sampling has also been applied to a related program analysis task, the generation of test cases which obtain good program coverage [28].

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 [5]. A brute-force approach to replacing double-precision instructions with their single-precision equivalents appears in [21], and a randomized technique for producing floating-point narrowing conversions at the source code level is discussed in [26]. 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 [13] and a critique of using execution time for measuring improvement in performance is discussed in [7]. 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 hence 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 which 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 [5].

8. Conclusion and Future Work

We have described a new approach to the optimization of high-performance floating-point kernels that 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 which can be applied to the verification of all of the optimizations produced by the prototype implementation of our system. To address this limitation, we show a general-purpose randomized validation technique which can be used to establish strong evidence for the correctness of floating-point programs. Although our technique is not applicable to application domains which 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.

Our system is the first implementation of a stochastic optimizer which has been successfully applied to full programs. Our system achieves 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 still remain, including extensions which allow for optimizations based on the reorganization of program-wide data structures.

Acknowledgments

This material is based on research sponsored by a fellowship from VMWare and DARPA under agreement number FA84750-14-2-0006. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied of DARPA or the U.S. Government.

References

[1] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[2] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, pages 198–209, 2010.

[3] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *PLDI*, pages 453–462, 2012.

[4] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity and robustness of programs. *Commun. ACM*, 55(8):107–115, 2012.

[5] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. PPoPP '14, pages 43–52, 2014.

[6] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *EuroSys*, pages 315–328, 2011.

[7] C. Curtsinger and E. D. Berger. Stabilizer: Statistically sound performance evaluation. In *ASPLOS*, pages 219–228, 2013.

[8] E. Darulova and V. Kuncak. Sound compilation of reals. In *POPL*, pages 235–248, 2014.

[9] F. de Dinechin, C. Q. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using GAPPA. *IEEE Trans. Computers*, 60(2):242–253, 2011.

[10] L. M. de Moura, N. Bjørner, and C. M. Wintersteiger. Z3. `http://z3.codeplex.com/`.

[11] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, pages 53–69, 2009.

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

[13] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[14] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[15] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, pages 131–140, 2012.

[16] P. S. Heckbert, editor. *Graphics gems IV*. Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[17] F. Ivancic, M. K. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*, pages 49–58, 2010.

[18] M. Jha and S. Raskhodnikova. Testing and reconstruction of lipschitz functions with applications to data privacy. *Electronic Colloquium on Computational Complexity (ECCC)*, pages 11–57, 2011.

[19] F. Jzquel and J. M. Chesneaux. CADNA: A library for estimating round-off error propagation. *Computer Physics Communications*, 178 (12):933–955, 2008.

[20] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. FloPSy - Search-based floating-point constraint solving for symbolic execution. In *ICTSS*, pages 142–157, 2010.

[21] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre. Automatically adapting programs for mixed-precision floating-point computation. ICS '13, pages 369–378, 2013.

[22] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *SAS*, pages 316–333, 2011.

[23] G. C. Necula and S. Gulwani. Randomized algorithms for program analysis and verification. In *CAV*, 2005.

[24] A. Podgurski. Reliability, sampling, and algorithmic randomness. TAV4, pages 11–20, New York, NY, USA, 1991. ACM.

[25] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. `http://www.R-project.org`.

[26] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *SC*, 2013.

[27] P. Rümmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *SMT*, 2010.

[28] S. Sankaranarayanan, R. M. Chang, G. Jiang, and F. Ivancic. State space exploration using feedback constraint generation and Monte Carlo sampling. In *ESEC/SIGSOFT FSE*, pages 321–330, 2007.

[29] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, pages 305–316, 2013.

[30] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken. Data-driven equivalence checking. In *OOPSLA*, pages 391–406, 2013.

[31] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[32] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[33] 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):58–68, 2000.

[34] E. Tang, E. T. Barr, X. Li, and Z. Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *ISSTA*, pages 131–142, 2010.

[35] C. W. Ueberhuber. *Numerical computation: Methods, software and analysis*. Springer-Verlag, 1997.

[36] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. C. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454, 2012.