Efficient Search for Inputs Causing High Floating-point Errors

Wei-Fan Chiang Ganesh Gopalakrishnan Zvonimir Rakamarić Alexey Solovyev

School of Computing,
University of Utah,
Salt Lake City, UT 84112, USA
{wfchiang,ganesh,zvonimir,monad}@cs.utah.edu

Abstract

Tools for floating-point error estimation are fundamental to program understanding and optimization. In this paper, we focus on tools for determining the input settings to a floating point routine that maximizes its result error. Such tools can help support activities such as precision allocation, performance optimization, and auto-tuning. We benchmark current abstraction-based precision analysis methods, and show that they often do not work at scale, or generate highly pessimistic error estimates, often caused by non-linear operators or complex input constraints that define the set of legal inputs. We show that while concrete-testing-based error estimation methods based on maintaining shadow values at higher precision can search out higher error-inducing inputs, suitable heuristic search guidance is key to finding higher errors. We develop a heuristic search algorithm called Binary Guided Random Testing (BGRT). In 45 of the 48 total benchmarks, including many real-world routines, BGRT returns higher guaranteed errors. We also evaluate BGRT against two other heuristic search methods called ILS and PSO, obtaining better results.

Keywords Sequential and parallel programming; floating-point error estimation methods; guided search.

1. Introduction

Computational errors caused by limited precision implementations of floating-point routines are a central concern in high-performance computing at all levels of scale ranging from high-end supercomputers through hand-held electronics. Researchers often allocate limited precision to gain higher performance, and study this trade-off using existing tools (e.g., [12, 31, 36]). Others have studied the effect of parallelization strategy selection in the limited-precision context, given that it has a direct impact on the effective shape of expression trees (e.g., [2, 3, 9, 15, 16]).

There are many challenges in developing a general-purpose tool for computing the worst error-causing inputs. The program structure as well as the operations employed can span a huge variety. Consequently, the output can exhibit high sensitivity to input values as well as internal loss of precision. Closed-form solutions for errors, or broadly applicable error-compensation techniques, are virtually impossible to develop except in special domains (e.g., [25]). Short of exhaustive search, there are no obvious ways to find the *worst error-causing inputs*. However, since realistic programs are quite complex and large, exhaustive search is infeasible. The novel approach we propose is based on *heuristic-guided search*, and we provide a new tool S^3FP (input-space Sour-Spot Searcher for Floating-Point) in support of this work. S^3FP can generate *guaranteed lower-bounds on imprecision*. To the best of our knowledge, this is the first tool of its kind, which computes guaranteed high lower bounds for many real-world programs, including library functions within Magma [34], implementations of FFT [39], benchmark suites such as Parboil [39], and also components of active projects (e.g., Uintah [33], where, after significant performance tuning, the developers sought our help to check for precision loss). S^3FP may also be used, for example, as an assistant while publishing floating-point routines in a library—by tagging it with the *guaranteed lower-bound* on relative error that a programmer can expect. Furthermore, it may find uses in auto-tuning compilers where the search (for algorithms or implementations) may be based both on performance and precision [2].

Automatic determination of inputs that cause high errors[1] can be useful in many settings. In Precimonious [36], the allocation of precision is based on manually provided training inputs; the work in this paper can help automate this aspect. In recent work [17], the authors report their initial efforts on proving safe separation zones for aircrafts using the PVS theorem prover, and how these proofs were re-done for finite-precision implementations in C. The authors employed the static analysis tool Gappa [12] for estimating bounds on variable values. We show in this paper that tools such as Gappa can often generate pessimistic results, and therefore, it is possible that the separation proofs will not carry over.[2] In these cases, S^3FP can help confirm at least some of the Gappa findings as genuine, allowing designers to probe further and refine their algorithm.

Related Work. Many existing tool-based approaches for precision estimation are based on static analysis, as supported by tools such as Gappa [12] or SmartFloat [11]. While the use of Satisfiability Modulo Theories (SMT) based tools is possible for more precise estimation, we have not come across any such tools. Ideas combining symbolic techniques and heuristics have been studied, but not applied to the domain of imprecision analysis. (We later study in §3.5 one of the algorithms used in previous work called PSO [38].) Static analysis based precision estimation can yield pessimistic results (very high values of estimated relative error). This problem becomes worse when non-linear operations are present, or

[1] In §2, we define the term *relative error* and explain why we choose this as our uniform metric (barring a few exceptions) for benchmarking.

[2] The authors of [17] reported no such failure of proof carry-over, although in personal conversation they agreed it was possible.

UU IR Author Manuscript

Benchmark	Operators Used
Microbenchmarks	$\{+, -, /\}$
Reductions (BR,IBR,IBR^K)	$\{+, -\}$
DQMOM	$\{+, -, /\}$ (exp unrolled)
FFT	$\{sin, cos, +, -, *, /\}$
LU and QR decomp, matrix mult.	$\{+, -, *, /\}$

Table 1: Operators Used in our Benchmarks

when certain inputs are semantically related (we demonstrate these through microbenchmarks in §2).

We believe that a practical way to handle large problem sizes, complex code structures, as well as non-linear operators is to employ some kind of *search* over input configurations—mappings of inputs to real-number intervals (e.g., for a two-input function, a configuration may be: $i_1 \mapsto [0.5, 0.6], i_2 \mapsto [1.5, 2.5]$). Our main contribution is a search method called Binary Guided Random Testing (BGRT) for locating inputs that (heuristically) cause the highest floating-point errors ("worst inputs"). For comparison, we also implemented two other guided search methods to locate the (inputs causing the) highest error: the first is based on Iterated Local Search (ILS, [32]) and the second on Particle Swarm Optimization (PSO, [26]). Our results are now summarized over 48 experiments (Table 1 lists our benchmarks and the operators used in them):

- Unguided Random Testing (URT) found the highest error in 3 of the 48 experiments.
- BGRT found the highest error in 39 of the 48 experiments.
- ILS found the highest error in 5 of the 48 experiments.
- PSO found the highest error in 1 of the 48 experiments.
- Also, compared to URT:
  - BGRT found higher error in 45 of the 48 experiments.
  - ILS found higher error in 32 of the 48 experiments.
  - PSO found higher error in 22 of the 48 experiments.

To eliminate biases in terms of implementation, we confirmed that BGRT produces these higher relative error values *both* for the same overall runtime as well as for the same number of search steps (Tables 6, 7, 8, and 10). Clearly, much like other search based algorithms (e.g., Boolean satisfiability solvers), there is ample opportunity to further tune the heuristics of BGRT, now that we have proposed one choice that appears to outperform previous methods. In summary, our main contributions are these:

- We evaluate static analysis and SMT-based approaches for precision, and for the first time bring out their pros and cons.
- We offer BGRT as our current best choice for guided random search, and evaluate it on real-world benchmarks (e.g., Parboil), libraries (e.g., Magma), and sequential applications (e.g., linear solver of DQMOM).
- We release the S^3FP tool on our website [37] to help parallel programmers conduct error analysis in practice.

2. Background and Microbenchmarking

We introduce error metrics commonly used in floating-point error estimation followed by a microbenchmark-based study of various existing tools, and finally our precision estimation techniques based on shadow values.

The two most common indicators of floating-point error are *absolute error* and *relative error*. For a program P whose output is O, we use $O_{\mathbb{R}}$ to denote P's output which is calculated under infinite precision. We use $O_{\mathbb{F}}$ to denote P's output which is calculated under finite precision (such as 32-bit floating-point arithmetic). The absolute error of P on its output O is then $|O_{\mathbb{F}} - O_{\mathbb{R}}|$. The relative error on the output O is $|(O_{\mathbb{F}} - O_{\mathbb{R}})/O_{\mathbb{R}}|$. When estimating relative error, the case of $O_{\mathbb{R}} = 0$ (which causes the relative error to become undefined) must be properly handled. In this paper, we employ a padding constant, defining relative error as $|(O_{\mathbb{F}} - O_{\mathbb{R}})/max(O_{\mathbb{R}}, \delta)|$ similar to that employed in [5]. We choose $\delta = 10^{-3}$, keeping it sufficiently away from 0 while being relatively small with respects to the magnitudes of outputs in our experiments.

Tool	Benchmark 1	Benchmark 2	Benchmark 3
Gappa	Inf	7.7548e-16	NA
SmartFloat	1.0362e-15	NA	NA
SMT	4.9960e-15	Timeout	2.4367e-14

Table 2: Experimental Results for Microbenchmarks. 'NA' stands for the tool not handling the case correctly (see text). Timeout is set to one hour.

We observe that floating-point precision has been approached either through the metric of absolute error or relative error. For example, SmartFloat reports relative errors by default. On the other hand, Precimonious [36], an automatic floating-point bit-width allocation tool, measures absolute error by default. In our work, we select relative error as our default metric of floating-point error because, compared to absolute error, the scale of relative error is not related to the scales of the precise ($O_{\mathbb{R}}$) and the imprecise ($O_{\mathbb{F}}$) values. This property of relative error is helpful in pinpointing and isolating those instructions that introduce the most floating-point error in a program.

Recently, some floating-point precision estimation efforts have focused on detecting specific phenomena such as *catastrophic cancellation* [5, 29]. BGRT can complement such efforts.

There are three main approaches for abstract analysis of floating-point precision: interval arithmetic, affine arithmetic, and satisfiability modulo theories (SMT). Abstract analysis over-approximates floating-point error. It is generally used when verifying safety criteria such as separation proofs for aircraft [17].

Interval arithmetic tools [12, 14] are usually combined with proof assistants such as Coq [10] to work as lemma generators [6, 7]. Interval arithmetic based approaches tend to produce pessimistic results when computed values are interdependent (e.g., when one value is generated as the sum of two inputs $x + y$ and the other as $x - y$, their sum must be recognized as $x + x$). This situation is illustrated in Fig. 1a and further explained later in this section.

Some interval arithmetic tools such as Gappa employ expression re-writing to improve this situation. Affine arithmetic tools [11, 31] handle input dependencies better. However, when applied to precision analysis of programs, they cannot smoothly handle path conditions. While SMT [24, 27] based approaches[3] do not have these limitations, they are, however, limited by scalability. While SMT has recently been used for floating-point exception detection [4], their approach does not help with error estimation.

We use four benchmarks, three microbenchmarks and one real-world benchmark, to illustrate the limitations of abstract analysis.

- We selected one tool from each abstract analysis approach to measure the worst-case relative errors on the outputs of the four benchmarks.
- For interval arithmetic approach, we selected Gappa [12].
- For affine arithmetic approach, we selected SmartFloat [11].

[3] An approach to encode floating-point round-off errors was published in [24], and the method of using binary search for bounding floating-point numbers was published in [27]. A single tool combining these is what we implemented; we have not encountered such a tool by others, yet.

UU IR Author Manuscript

1: double $x_0 \leftarrow [1.0, 2.0]$, $x_1 \leftarrow [1.0, 2.0]$, $x_2 \leftarrow [1.0, 2.0]$
2: double $p_0 \leftarrow (x_0 + x_1) - x_2$
3: double $p_1 \leftarrow (x_1 + x_2) - x_0$
4: double $p_2 \leftarrow (x_2 + x_0) - x_1$
5: double $sum \leftarrow (p_0 + p_1) + p_2$
6: **compute error:** sum // $(x_0 + x_1) + x_2$

(a) Microbenchmark 1

1: double $sum \leftarrow 0.0$, $x_0 \leftarrow [1.0, 3.0]$, ..., $x_7 \leftarrow [1.0, 3.0]$
2: **for** $i = 0$ *to* 7 **do**
3: $sum \leftarrow sum + x_i$
4: **end for**
5: **assume** $1.0 \leq x_0 \ldots x_7 \leq 2.0$
6: **compute error:** sum

(b) Microbenchmark 2

1: double $x \leftarrow [-1.0, 1.0]$, $y \leftarrow [-1.0, 1.0]$
2: **assume** $x \geq y + 0.1 \wedge y \geq 0.0$ // $(x - y) > 0$
3: double $z = (x+y)/(x-y)$ // divide-by-zero is impossible
4: **compute error:** z

(c) Microbenchmark 3

Figure 1: Microbenchmarks

Tool	$O_{\mathbb{R}}$	$O_{\mathbb{F}}$	Rel. Error
Gappa	$[-9e+10, 9e+10]$	$[-9e+10, 9e+10]$	7.3186e+09
SmartFloat	$[-9e+10, 9e+10]$	$[-9e+10, 9e+10]$	2.2207e-11
SMT	$[-9e+05, 9e+05]$	$[-9e+05, 9e+05]$	Timeout

Table 3: Results Returned by Abstract Analysis Tools on DQMOM. Timeout is set to one hour.

- For SMT approach, we built our own tool that uses Z3 [13] as its underlying solver. (The ideas of building this SMT based tool are proposed in [24, 27]).
- Table 2 shows the worst-case relative errors detected by tools for our microbenchmarks.
- Table 3 shows output ranges and the worst-case relative errors detected by tools for our real-world benchmark.

Microbenchmark 1 (see Fig. 1a) illustrates the scenario that interval arithmetic approaches could return pessimistic results. Without thoroughly considering the dependencies among p_0, p_1, and p_2, the range of sum is $[0, 9]$. (The true range is $[3, 6]$.) When calculating the relative error of sum, 0 is considered to be a possible value of the precise sum. Thus, interval arithmetic tools, which cannot precisely reason about variable dependencies, could report infinite as the bound of the sum's relative error.

Microbenchmark 2 (see Fig. 1b) illustrates the scalability problem of SMT (timeout in Table 2). Also, this microbenchmark contains a condition, encoded using the **assume** statement, which SmartFloat cannot handle. In particular, note that SmartFloat provides predicate functions, called *possibly* and *certainly*, to check whether a condition is possibly true or certainly true. In the microbenchmark, SmartFloat could check that $1.0 \leq x_0 \ldots x_7 \leq 2.0$ is true at line 5. However, SmartFloat cannot take the condition (on line 5) into account when computing floating-point error. Thus we report 'NA' as SmartFloat's result for this microbenchmark.

Microbenchmark 3 (see Fig. 1c) illustrates the scenario that mixes conditions and non-linear operations. Both Gappa and SmartFloat were unable to handle this microbenchmark. To the best of our knowledge, Gappa, a state-of-the-art interval arithmetic tool, relies on saturation based solving to handle additional conditions in measuring floating-point error. In our experiments with this microbenchmark, Gappa failed to find the bounds of $x - y$. On the other hand, given its precision, SMT successfully handled such scenario.

Table 3 shows the results of using abstract analysis tools to measure the range of the output values (computed both under finite and infinite precision) and the relative error of DQMOM. (We introduce DQMOM in §4.1.) We can observe that both interval and affine arithmetic tools, namely Gappa and SmartFloat, returned overly pessimistic results. We manually calculated the range of the outputs and verified that SMT's results were equal to our manually generated answers. Thus we can conclude that SMT has potential to precisely measure floating-point error. However, when estimating the relative error for DQMOM, SMT was limited by its scalability.

To summarize, we conclude that the interval-arithmetic-based approach for floating-point precision measurement cannot handle input dependencies well (illustrated in Fig. 1a). The affine-arithmetic-based approach cannot estimate floating-point error while considering additional conditions (illustrated in Figs. 1b and 1c). On the other hand, the SMT-based approach could overcome both of these limitations. However, it is limited by its scalability (illustrated in Fig. 1c and Table 3). Therefore, when measuring floating-point precision for parallel programs, we consider none of the three approaches is a suitable solution.

P	A program
$I_1 \ldots I_n$	Input variables of P
$R_1 \ldots R_n$	Value ranges (provided by users)
${R_i}^{p/q}$ $p, q \in \mathbb{N}, 0 \leq p < q$	If $R_i = [x, y]$ then ${R_i}^{p/q} = \left[x + \frac{(y-x)*p}{q}, x + \frac{(y-x)*(p+1)}{q}\right]$
$C_1 \ldots C_n$	Configurations: functions from input vars to their ranges
$Eval : P \times C \times \mathbb{N} \mapsto \mathbb{F}$	Evaluation of a program under a configuration
URT	Unguided Random Testing
BGRT	Binary Guided Random Testing
ILS	Iterated Local Search
PSO	Particle Swarm Optimization
RT	Random Testing; one of {URT, BGRT, ILS, PSO}
GRT	Guided Random Testing; one of {BGRT, ILS, PSO}

Table 4: Terminology and Notations

3. Guided Random Testing

3.1 Terminology and Notations

We now introduce terminology and notations common to all our guided-search discussions (see Table 4).

Shadow Value Execution. Given a program P and its legal (intended) range of inputs, we execute P under the given precision setting, and also conduct a higher ("infinite") precision execution.[4] Given these outputs of P, we can compute the absolute/relative errors, as described in §2.

[4] Given the high difficulty of correctly modifying all external benchmarks to the same higher precision, we sometimes have ended up choosing 64 bits and sometimes 128 bits for our "infinite" precision. However, within each comparative study, we employ only one high-precision allocation.

$$\left\{\begin{array}{l} I_0 \mapsto [0.0, 1.0] \\ I_1 \mapsto [0.0, 1.0] \\ I_2 \mapsto [0.0, 1.0] \\ I_3 \mapsto [0.0, 1.0] \\ I_4 \mapsto [0.0, 1.0] \\ I_5 \mapsto [0.0, 1.0] \\ I_6 \mapsto [-1.0, 1.0] \\ I_7 \mapsto [-1.0, 1.0] \\ I_8 \mapsto [-1.0, 1.0] \end{array}\right\}$$

(a) Original Configuration

k	Rel. Error
10^1	3.7270e-07
10^2	1.6205e-06
10^3	1.4213e-05
10^4	1.6553e-04
10^5	2.7246e-04
10^6	8.6022e-04

(b) Evaluation Results

Figure 2: Original Configuration of DQMOM and its Evaluation Results

Valid Input. Here we describe how a *valid* input is found. For an input variable I_i of a program P, there is a corresponding (floating-point) value range R_i that contains all possible values of I_i. If I_1 is the only input of P, a valid input is a random value in R_1. If P has multiple input variables $I_1 \ldots I_n$, a valid input is a random sampling, one from each range $R_1 \ldots R_n$. We often denote R_i by $[x, y]$ where $x \leq y$. A sub-range $R_i{}^{p/q}$, where $0 \leq p < q$, is a subset of R_i defined by

$$\left[x + \frac{(y-x)*p}{q}, x + \frac{(y-x)*(p+1)}{q}\right]$$

Configuration. A *configuration* is a function from program inputs to ranges of values. For two configurations, c_x and c_y, c_y is *tighter* than c_x if they have the same domain (input variables) and for every i in this domain, $c_y(i)$ is contained in $c_x(i)$. In our work, we assume that users supply the initial configurations, starting from which the search proceeds seeking the tightest configuration that maximizes error (within resource limits, and perhaps finding a local maximum).

Evaluation of a Configuration. During GRT, an initial configuration will be recursively divided into tighter configurations. (The intuition and the algorithm are introduced in §3.4.) We need to *evaluate* (tighter) configurations and compare them through shadow value computations. More specifically, for a configuration C, we sample k valid inputs from C and also perform shadow value execution k times on the program P. From the k measured floating-point errors, we use the highest one as the evaluation result. Function $Eval$ takes a program, a configuration, and an integer as its arguments. For example, $Eval(P, C, k)$ evaluates the given configuration C by performing shadow value execution k times on P, and reports the highest measured floating-point error. Finally, one configuration is chosen as "better" based on such comparisons.

3.2 Intuition behind Guided Random Testing

In this section, we give an intuitive description of GRT methods, using the Direct Quadrature Method of Moments benchmark (DQMOM, detailed in §4.1) as an illustrative example. First, Fig. 2a shows the initial (user-given) configuration of DQMOM mapping 9 program inputs $I_0 \ldots I_8$ into their ranges. Fig. 2b shows its evaluation results (highest relative error obtained) after the shown number of evaluations (trials) going from $10^1 \ldots 10^6$. As is to be expected, the higher the k parameter in $Eval(P, C, k)$, the higher is the generated relative error. Fig. 3 illustrates the intuition further with two experiments: Exp_1 and Exp_2. For each experiment, we find a "good" and "bad" configuration to replace the initial configuration, and then we evaluate both. Note that both good and bad configurations are tighter than the initial configuration. Figs. 3a and 3b show the bad and good configurations of Exp_1 and Exp_2, respectively. Fig. 3c gives the highest computed relative errors for these four configurations. Figs. 3d and 3e plot the comparison of the computed relative errors for the original configuration, and the bad and good configurations of the two experiments. From these two plots we can observe the following:

- With higher number of iterations k, the exhibited error increases.
- The choice of the initial configuration matters.
- After only a few iterations (i.e., 10–100), it is possible to select a better/worse error inducing configuration.

The search heuristics we have experimented with attempt to capitalize on these observations.

3.3 Unguided Random Testing

Algorithm 1 Unguided Random Testing

```
1: Input: P, C_init
2: Output: Computed highest floating-point error
3: WorstErr ← 0.0
4: while has resources do
5:     CurrErr ← Eval(P, C_init, 1)
6:     if CurrErr > WorstErr then
7:         WorstErr ← CurrErr
8:     end if
9: end while
10: return WorstErr
```

Unguided random testing takes as input a program to measure floating-point error for and its initial configuration. Then it just repeatedly samples inputs from the initial configuration and computes the highest floating-point error by performing shadow value executions. Algo. 1 shows the pseudocode of unguided random testing. It basically repeatedly calls the evaluation function $Eval$ until running out of the given resource limit (runtime or the total number of shadow value executions).

3.4 Binary Guided Random Testing

In this section, we describe our custom binary guided random testing (BGRT) algorithm, which is the best guiding heuristic we discovered so far. BGRT takes the initial configuration, and iteratively zooms into tighter configurations that result in higher error. Each BGRT iteration starts with an configuration (not necessary the initial), enumerates some of its tighter configurations, and selects the one that (locally) maximizes the error. The selected tighter configuration starts the next BGRT iteration. With a certain probability, we also allow restarts to the user-given initial configuration so that we prevent getting stuck in a local maximum.

To explain how BGRT enumerates tighter configurations, we first introduce some helper routines. $PartConf$ randomly partitions a configuration into two non-empty configurations such that the domains are mutually exclusive and exhaustive with respect to the domain of the incoming configuration. For example, c_q and c_r could be one possible result of $PartConf(c_p)$:

$$c_p : \left\{\begin{array}{l} I_0 \mapsto [-1, 1] \\ I_1 \mapsto [0.1, 0.2] \\ I_2 \mapsto [-0.2, -0.1] \end{array}\right\} \quad c_q : \left\{\begin{array}{l} I_0 \mapsto [-1, 1] \\ I_2 \mapsto [-0.2, -0.1] \end{array}\right\} \quad c_r : \left\{\begin{array}{l} I_1 \mapsto [0.1, 0.2] \end{array}\right\}$$

We define the *upper half* and the *lower half* configurations of an incoming configuration c as $\overbrace{c}$ and $\underbrace{c}$, respectively. The upper half is obtained by changing each range (say R_i) of the incoming configuration to its upper half sub-range (i.e., $R_i{}^{1/2}$). Examples of the upper and the lower halves of the configuration c_p are as follows:

$$\overbrace{c_p} : \left\{\begin{array}{l} I_0 \mapsto [0, 1] \\ I_1 \mapsto [0.15, 0.2] \\ I_2 \mapsto [-0.15, -0.1] \end{array}\right\} \quad \underbrace{c_p} : \left\{\begin{array}{l} I_0 \mapsto [-1, 0] \\ I_1 \mapsto [0.1, 0.15] \\ I_2 \mapsto [-0.2, -0.15] \end{array}\right\}$$

Bad Config.	Good Config.
$I_0 \mapsto [0.3, 0.4]$	$I_0 \mapsto [0.6, 0.7]$
$I_1 \mapsto [0.2, 0.3]$	$I_1 \mapsto [0.4, 0.5]$
$I_2 \mapsto [0.2, 0.3]$	$I_2 \mapsto [0.9, 1.0]$
$I_3 \mapsto [0.3, 0.4]$	$I_3 \mapsto [0.9, 1.0]$
$I_4 \mapsto [0.2, 0.3]$	$I_4 \mapsto [0.9, 1.0]$
$I_5 \mapsto [0.2, 1.3]$	$I_5 \mapsto [0.4, 0.5]$
$I_6 \mapsto [-0.5, -0.4]$	$I_6 \mapsto [0.9, 1.0]$
$I_7 \mapsto [-0.4, -0.3]$	$I_7 \mapsto [-0.7, -0.6]$
$I_8 \mapsto [-0.3, -0.2]$	$I_8 \mapsto [-0.1, 0.0]$

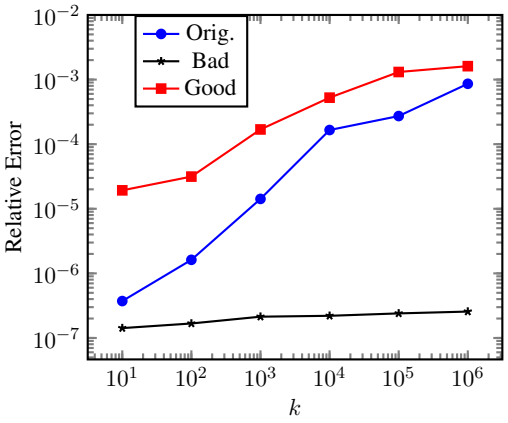
(a) Bad and Good Configurations of Exp_1

Bad Config.	Good Config.
$I_0 \mapsto [0.1, 0.2]$	$I_0 \mapsto [0.0, 0.1]$
$I_1 \mapsto [0.0, 0.1]$	$I_1 \mapsto [0.8, 0.9]$
$I_2 \mapsto [0.0, 0.1]$	$I_2 \mapsto [0.0, 0.1]$
$I_3 \mapsto [0.0, 0.1]$	$I_3 \mapsto [0.8, 0.9]$
$I_4 \mapsto [0.1, 0.2]$	$I_4 \mapsto [0.1, 0.2]$
$I_5 \mapsto [0.0, 0.1]$	$I_5 \mapsto [0.6, 0.7]$
$I_6 \mapsto [-0.8, -0.7]$	$I_6 \mapsto [0.3, 0.4]$
$I_7 \mapsto [-0.9, -0.8]$	$I_7 \mapsto [-0.9, -0.8]$
$I_8 \mapsto [-0.8, -0.7]$	$I_8 \mapsto [-0.9, -0.8]$

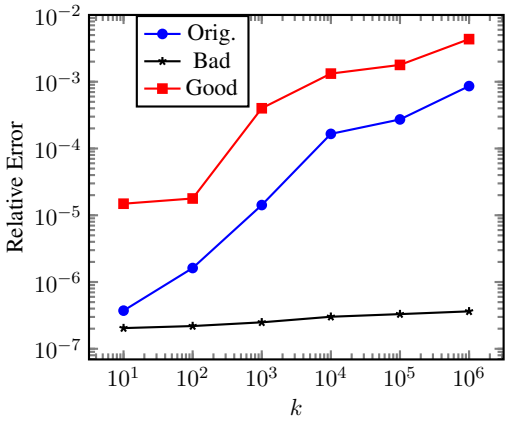
(b) Bad and Good Configurations of Exp_2

k	Exp_1		Exp_2	
	Bad Config.	Good Config.	Bad Config.	Good Config.
10^1	1.4218e-07	1.9292e-05	2.0404e-07	1.4894e-05
10^2	1.6768e-07	3.1482e-05	2.1905e-07	1.7829e-05
10^3	2.1361e-07	1.6898e-04	2.4907e-07	4.0045e-04
10^4	2.2047e-07	5.2483e-04	3.0191e-07	1.3234e-03
10^5	2.4011e-07	1.3093e-03	3.3055e-07	1.7865e-03
10^6	2.5629e-07	1.6118e-03	3.6362e-07	4.3456e-03

(c) Evaluation Results of Bad and Good Configurations



(d) Relative Errors for Exp_1



(e) Relative Errors for Exp_2

Figure 3: Two Experiments Illustrating the Intuition behind GRT

These are both tighter configurations with respect to the incoming configuration.

NextGen (in Algo. 2) is a function that takes an configuration (*conf*) and enumerates a set of its tighter configurations (held by *nextg*) to explore. *NextGen* first partitions *conf* into two configurations c_x and c_y. Then it finds the upper and lower halves of the two partitions and permutes them. Such divide-and-permute will be repeated N_{part} times, which is one of the parameters of BGRT. Note that $\overbrace{c_x} \cup \overbrace{c_y} = \overbrace{conf}$. Thus, we add the upper and the lower halves of *conf* into *nextg* once in the beginning of *NextGen*. In *NextGen*, we use $\cup$ to denote the union of the two tighter configurations (with disjoint domains) into one configuration. We use $\uplus$ to denote forming a set of configurations (discriminated union).

Algo. 2 shows our BGRT algorithm. A BGRT iteration is shown from lines 16–30. The starting configuration of a BGRT iteration is *LocalConf*, and it is used to enumerate tighter configurations (*NextConfs*) at line 17. From the enumerated tighter configurations, we choose the one whose evaluation results in the highest error (lines 18–24). The number of shadow value executions k is also a parameter of BGRT. Lines 25–27 keep track of the all-time highest floating-point error in *WorstErr*. Line 29 resets the starting configuration to the user-given initial configuration if BGRT decides to restart.

3.5 Other Guided Random Search Strategies

Beside BGRT, we have also implemented and explored two other GRT strategies: iterated local search (ILS) and particle swarm optimization (PSO). However, our experimental results currently show that ILS and PSO do not usually find higher floating-point errors than BGRT. The reason why we explored ILS and PSO is that they have been successful in finding (heuristically) optimal solutions for other problems. ILS is used in a framework called ParamILS [22] that automatically searches for optimal parameter settings of algorithms. It defines neighborhood relationships between models in a search space, and searches for the optimal model by iteratively evaluating selected models and their neighbors. PSO is used in a randomized constraint solver called CORAL [38] that is capable of finding floating-point models for given floating-point constraints. It keeps track of a group of models with their evaluation results, finds the next group to explore based on the current group, and reports the optimal among all explored models. In this paper, we omit the details of our implementations of ILS and PSO; those details can be found on our website [37].

	Sequential			Parallel (GPU)			
	BR	IBR	DQMOM	FFT	LU	QR	MM
Original	32	32	32	32	32	32	32
High Prec.	128	128	128	128	64	64	64

Table 5: Benchmark Precisions Used in Shadow Value Execution

4. Experimental Results

To assess the effectiveness of the four proposed random search techniques, namely URT, BGRT, ILS, and PSO, we implemented them in our prototype tool S^3FP and compared on our benchmark suite. The benchmark suite includes both sequential and parallel programs. Our sequential benchmarks comprise of balanced reduction (BR), imbalanced reduction (IBR), and direct quadrature method of moments (DQMOM). Although DQMOM itself is sequential, we extracted it from a parallel computational framework [33]. Our parallel benchmarks are well-known GPU primitives including fast Fourier transform (FFT), LU decomposition (LU), QR decomposition (QR), and matrix multiplication (MM). FFT is taken from Parboil parallel benchmark set (v0.2) [39]; LU, QR, and MM are from Magma library (v1.4.0) [34].

All examples in our benchmark suite perform 32-bit floating-point arithmetic. To be able to perform shadow value execution, we created higher precision versions employing 64- or 128-bit

Algorithm 2 Binary Guided Random Testing

```
1: Input: P, C_init, k, N_part
2: Output: Computed highest floating-point error
3:
4: procedure NextGen(conf)
5:     nextg ← conf ⊎ conf
6:     for i = 1 to N_part do
7:         (c_x, c_y) = PartConf(conf)
8:         nextg ← nextg ⊎ (c_x ∪ c_y) ⊎ (c_x ∪ c_y)
9:     end for
10:    return nextg
11: end procedure
12:
13: WorstErr, LocalErr ← 0.0
14: LocalConf ← C_init
15: while has resources do
16:     LocalErr ← 0.0
17:     NextConfs ← NextGen(LocalConf)
18:     for c ∈ NextConfs do
19:         err ← Eval(P, c, k)
20:         if err > LocalErr then
21:             LocalErr ← err
22:             LocalConf ← c
23:         end if
24:     end for
25:     if LocalErr > WorstErr then
26:         WorstErr ← LocalErr
27:     end if
28:     if random restart then
29:         LocalConf ← C_init
30:     end if
31: end while
32: return WorstErr
```

floating-point arithmetic. Table 5 summarizes the floating-point bit-width of the original benchmarks and their higher precision versions. Although various versions of the benchmarks compute with different bit-widths, S^3FP generates only 32-bit floating-point numbers as inputs. Using exclusively 32-bit floating-point inputs guarantees that, when doing shadow value execution, the original and higher precision version are executed on identical inputs. If we used higher bit-width (i.e., 64- or 128-bit) inputs instead, we would immediately lose precision when type casting to 32-bit inputs for the original benchmarks, and the two executions would potentially diverge.

Outputs of most of our benchmarks are arrays or matrices, and hence defining relative errors for such benchmarks is not trivial. Our strategy for measuring relative errors for such benchmarks is to select one of the output array/matrix elements as a representative output, and measure only its relative error. This manual selection process is based on our study of the benchmarks: we chose the elements which are calculated using the highest number of floating-point operations since these are likely to exhibit the largest relative errors.

To perform a meaningful comparison of our random testing methods, we relied on measuring (or limiting) two different resources. The first resource is elapsed time, where we would limit the allocated running time (i.e., set a time out) to, for example, one hour. This gives us a good estimate of how well the proposed random testing techniques work in practice. However, it includes potential algorithm and implementation overheads. The second resource we measure is the total number of times a shadow value execution is repeated. This demonstrates the power of RT techniques under the assumption that all have the same algorithm/implementation overhead.

	Algo.	IBRK (2048)	BR (2048)	IBR (2048)	DQMOM (960)
Exp_1	URT^p	3.6151e-03	1.4106e-02	1.1035e-01	8.8729e-03
	BGRT^p	**2.7132e-01**	**9.6636e-01**	**4.4229e+01**	**6.0333e+00**
	ILS^p	2.5134e-02	4.3401e-01	5.0068e-01	4.6705e-02
	PSO^p	8.6183e-03	1.4833e-01	5.2374e-02	1.0133e-02
Exp_2	URT^p	3.1396e-02	3.5851e-01	3.2051e-01	2.4357e-03
	BGRT^p	**2.9659e-01**	**8.0504e-01**	**1.3488e+01**	**1.8198e+00**
	ILS^p	2.1614e-02	7.9974e-02	5.2502e-02	7.6498e-03
	PSO^p	3.1449e-02	2.7312e-01	9.4350e-01	5.9729e-03

Table 7: Relative Errors for Sequential Benchmarks. The number of allowed shadow value executions is set to 10^6.

We have implemented in S^3FP both sequential and parallel versions of all four runtime testing methods. In our tables with results, URT, BGRT, ILS, and PSO denote single-core implementations, while URT^p, BGRT^p, ILS^p, and PSO^p denote multi-core implementations. We parallelized URT^p by allowing simultaneous calls of $Eval(P, c_i, 1)$. BGRT^p, ILS^p, and PSO^p are parallelized by allowing a call of $Eval(P, c_i, k)$ to simultaneously perform shadow value executions. However, our current multi-core implementations can only be applied to sequential benchmarks. We will explore more parallelism in GRT implementations and apply them to parallel benchmarks in our future work.

4.1 Results for Sequential Benchmarks

Our benchmark suite contains four sequential programs. The sequential balanced reduction (BR) simulates parallel reduction [18], while the sequential imbalanced reduction (IBR) simulates multiple threads accumulating values using atomic operations. Since we have not fully investigated how non-deterministic thread interleavings affect floating-point precision, we converted parallel reductions into sequential versions to perform this comparison. This allows us to compare IBR^K, IBR, and BR. These examples are further discussed in §5. IBR^K is a variant of IBR that employs the more precise Kahan summation [25]. In our experiments, all BR and IBR input variables are initially mapped to ranges $[-100, 100]$.

The direct quadrature method of moments (DQMOM) is a core function of a combustion simulation component of Uintah computational framework [33]. DQMOM models the transform of particle density between moments, converts the transform as a linear system $AX = B$, and solves the system. In our initial configuration of DQMOM, the first 3/5 input variables are mapped to $[0.0, 1.0]$ and the last 2/5 input variables are mapped to $[-1.0, 1.0]$.

Table 6 gives the (highest) relative errors discovered by the four random testing methods on our sequential benchmarks within one hour time budget. In the table, "# SVE" denotes the total number of shadow value executions. (A call to $Eval(P, c_i, k)$ is counted as k shadow value executions.) The input size (the number of input variables) of each benchmark is given in parentheses next to the benchmark name. For each benchmark, we compared the four methods twice (denoted as Exp_1 and Exp_2), and highlighted the highest relative error using bold font. Table 7 shows the results of running RT methods with the same number of shadow value executions (10^6). From Tables 6 and 7, we can observe that BGRT usually returns the highest errors. Fig. 4 plots the results in Table 6.

4.2 Results for Parallel Benchmarks

For our parallel benchmarks, we have verified that their computational results (and hence, our error estimation) are unaffected by thread schedules, based on the following lines of reasoning. First, we assume the absence of data races (checking for races using tools

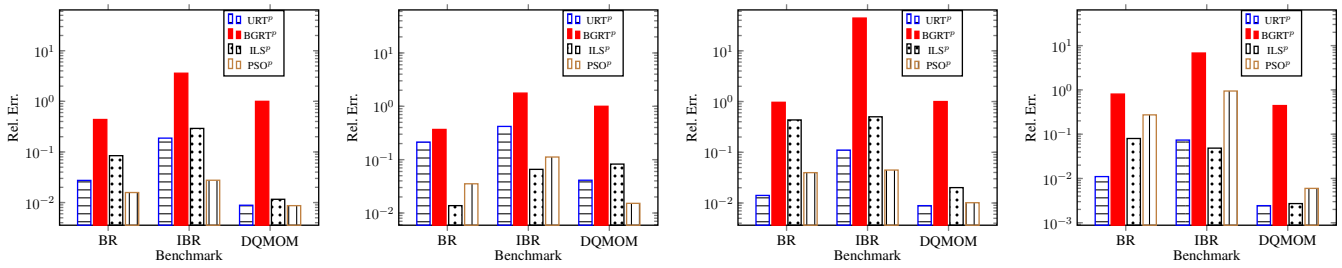
	Algo.	BR (512)		IBR (512)		DQMOM (240)	
		rel. error	# SVE	rel. error	# SVE	rel. error	# SVE
Exp_1	URT^p	2.7416e-02	1.2*10^6	1.8757e-01	1.2*10^6	8.8585e-03	1.2*10^6
	BGRT^p	**4.3774e-01**	1.1*10^6	**3.5947e+00**	1.2*10^6	**1.0000e+00**	1.2*10^6
	ILS^p	8.4400e-02	8.8*10^5	2.9171e-01	8.7*10^5	1.1611e-02	1.1*10^6
	PSO^p	1.5789e-02	1.0*10^6	2.7611e-02	1.0*10^6	8.6954e-03	1.1*10^6
Exp_2	URT^p	2.1344e-01	1.2*10^6	4.1944e-01	1.2*10^6	4.1150e-02	1.2*10^6
	BGRT^p	**3.6824e-01**	1.1*10^6	**1.7659e+00**	1.1*10^6	**1.0000e+00**	1.2*10^6
	ILS^p	1.3739e-02	8.8*10^5	6.5749e-02	8.7*10^5	8.2554e-02	1.1*10^6
	PSO^p	3.5282e-02	1.0*10^6	1.1199e-01	1.0*10^6	1.5171e-02	1.1*10^6

(a) Small Input Size

	Algo.	BR (2048)		IBR (2048)		DQMOM (960)	
		rel. error	# SVE	rel. error	# SVE	rel. error	# SVE
Exp_1	URT^p	1.4106e-02	6.4*10^5	1.1035e-01	6.5*10^5	8.8723e-03	6.6*10^5
	BGRT^p	**9.6636e-01**	6.1*10^5	**4.4229e+01**	6.1*10^5	**1.0000e+00**	6.4*10^5
	ILS^p	4.3401e-01	2.2*10^5	5.0068e-01	2.2*10^5	2.0105e-02	4.3*10^5
	PSO^p	3.9677e-02	4.6*10^5	4.4608e-02	4.6*10^5	1.0133e-02	5.9*10^5
Exp_2	URT^p	1.1031e-02	6.4*10^5	7.3680e-02	6.4*10^5	2.4357e-03	6.6*10^5
	BGRT^p	**8.0504e-01**	6.0*10^5	**6.8056e+00**	6.1*10^5	**4.4318e-01**	6.4*10^5
	ILS^p	7.9974e-02	2.2*10^5	4.8452e-02	2.0*10^5	2.7101e-03	4.3*10^5
	PSO^p	2.7312e-01	4.6*10^5	9.4350e-01	4.6*10^5	5.9729e-03	5.9*10^5

(b) Large Input Size

Table 6: Relative Errors for Sequential Benchmarks. Time budget is set to one hour.



(a) Small Input Size (Exp_1) (b) Small Input Size (Exp_2) (c) Large Input Size (Exp_1) (d) Large Input Size (Exp_2)

Figure 4: Relative Errors for Sequential Benchmarks. Time budget is set to one hour.

	Algo.	FFT (512)		LU (256)		QR (256)		MM (770)	
		rel. error	# SVE	rel. error	# SVE	rel. error	# SVE	rel. error	# SVE
$Exp.1$	URT	1.9481e-02	8.0*10^4	6.8969e-03	4.8*10^4	**3.5228e-02**	4.8*10^4	3.5777e-04	4.8*10^4
	BGRT	**2.3552e-02**	8.0*10^4	**1.7481e-02**	4.9*10^4	3.0875e-02	4.5*10^4	**1.2014e+01**	4.8*10^4
	ILS	3.8490e-03	7.9*10^4	9.3943e-03	4.9*10^4	3.9905e-04	4.8*10^4	2.0271e-02	4.8*10^4
	PSO	4.7028e-03	8.0*10^4	1.2302e-03	4.8*10^4	1.2890e-02	4.2*10^4	7.0844e-04	4.7*10^4
$Exp.2$	URT	7.0919e-03	8.0*10^4	6.2447e-04	4.9*10^4	5.0282e-03	4.8*10^4	2.1496e-03	4.9*10^4
	BGRT	2.0740e-02	8.0*10^4	**1.8670e-02**	4.9*10^4	6.1168e-02	4.7*10^4	**2.5339e+00**	4.8*10^4
	ILS	**2.8143e-02**	8.1*10^4	5.3179e-03	4.9*10^4	2.2136e-03	4.8*10^4	1.0493e-03	4.8*10^4
	PSO	1.2055e-02	8.0*10^4	1.3823e-03	4.8*10^4	**2.0000e+00**	4.7*10^4	1.9052e-03	4.8*10^4

(a) Small Input Size

	Algo.	FFT (2048)		LU (1024)		QR (1024)		MM (3074)	
		rel. error	# SVE	rel. error	# SVE	rel. error	# SVE	rel. error	# SVE
$Exp.1$	URT	9.9671e-03	7.7*10^4	1.1942e-03	4.8*10^4	3.2723e-02	4.6*10^4	1.0016e-02	4.6*10^4
	BGRT	3.4312e-02	7.6*10^4	**2.6197e-02**	4.8*10^4	**1.9540e-01**	4.6*10^4	**3.1161e+00**	4.6*10^4
	ILS	**6.8418e-02**	7.5*10^4	3.3736e-03	4.7*10^4	2.1083e-02	4.7*10^4	1.6710e-01	4.4*10^4
	PSO	3.5419e-03	7.6*10^4	2.8987e-03	4.7*10^4	4.3618e-02	4.1*10^4	8.6908e-04	4.1*10^4
$Exp.2$	URT	1.9560e-03	7.7*10^4	1.1742e-03	4.8*10^4	**1.6825e-01**	4.7*10^4	1.5422e-02	4.6*10^4
	BGRT	1.2580e-02	7.6*10^4	**2.5969e-02**	4.8*10^4	1.0213e-01	4.6*10^4	**1.7881e-01**	4.6*10^4
	ILS	**4.4445e-02**	7.5*10^4	7.9298e-03	4.8*10^4	3.9839e-02	4.7*10^4	7.6199e-03	4.4*10^4
	PSO	1.4056e-02	7.5*10^4	9.3751e-03	4.7*10^4	8.1161e-02	4.6*10^4	3.2531e-03	4.5*10^4

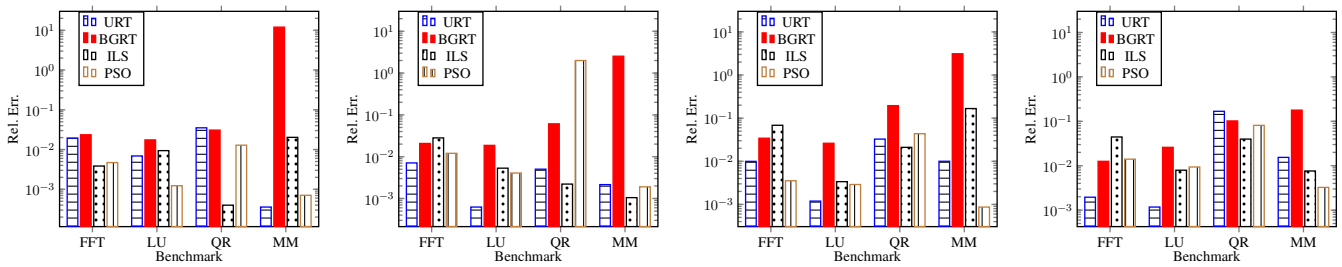
(b) Large Input Size

Table 8: Relative Errors for Parallel Benchmarks. Time budget is set to one hour.

such as [30] is future work). Second, we have manually verified that there are no schedule-dependent reduction operations (automation of these checks is future work).

All input variables of our parallel (GPU) benchmarks are initially mapped to $[-100, 100]$. Table 8 shows the computed relative errors for our parallel benchmarks given one hour time limit. We can observe that BGRT returns the highest relative errors in most of our experiments with parallel benchmarks. When limiting the number of shadow value executions as resource, BGRT still works the best among all four RT techniques (see Table 10). Fig. 5 plots the results from Table 8.

In Table 8, the largest input matrix size is 32×32. However, we observed that if the LU and QR benchmarks are called with these sizes, they directly call LAPACK [1] routines to *sequentially* compute the results. Table 9 show the results of applying larger input matrix size (1200×1200) that triggers GPU (paral-



(a) Small Input Size (Exp_1) (b) Small Input Size (Exp_2) (c) Large Input Size (Exp_1) (d) Large Input Size (Exp_2)

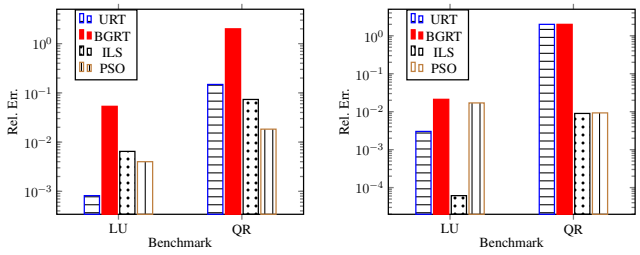
Figure 5: Relative Errors for Parallel Benchmarks. Time budget is set to one hour.

	Algo.	LU (1440000)		QR (1440000)	
		rel. error	# SVE	rel. error	# SVE
$Exp.1$	URT	8.1670e-04	1630	1.4829e-01	2600
	BGRT	**5.3323e-02**	1440	**2.0008e+00**	2130
	ILS	6.4743e-03	1210	7.3494e-02	1670
	PSO	3.9983e-03	1160	1.8321e-02	1580
$Exp.2$	URT	3.0200e-03	1630	1.9998e+00	2600
	BGRT	**2.1163e-02**	1440	**2.0009e+00**	2140
	ILS	6.1338e-05	1210	8.9875e-03	1670
	PSO	1.7010e-02	1160	9.2871e-03	1570

Table 9: Relative Errors for LU and QR with Large Input Matrices. Time budget is set to one hour.

		FFT (2048)	LU (1024)	QR (1024)	MM (3074)
Exp_1	URT	9.9671e-03	1.1942e-03	3.2723e-02	1.0016e-02
	BGRT	3.4312e-02	**2.6197e-02**	**1.9540e-01**	**3.1161e+00**
	ILS	**4.0819e-02**	3.3736e-03	3.6521e-02	1.6710e-01
	PSO	3.5419e-03	3.1306e-03	4.3618e-02	8.6908e-04
Exp_2	URT	1.9560e-03	1.1742e-03	**1.6825e-01**	1.5422e-02
	BGRT	3.1750e-03	**2.5969e-02**	1.0213e-01	**1.7881e-01**
	ILS	**3.8323e-02**	7.9298e-03	3.9839e-02	2.7150e-02
	PSO	1.4056e-02	9.3751e-03	8.1161e-02	3.2531e-03

Table 10: Relative Errors for Parallel Benchmarks. The number of allowed shadow value executions is set to $5 * 10^4$.



(a) Large Input Size (Exp_1) (b) Large Input Size (Exp_2)

Figure 6: Relative Errors for Parallel Benchmarks with 1200×1200 Input Matrices. Time budget is set to one hour.

lel) computations in LU and QR. Fig. 6 plots the results from Table 9. For the LU benchmark which performs LU decomposition, $A = P \times L \times U$, we calculate the product of P, L, and U, and measure the precision on one of the elements in the produced matrix. This precaution is required because the permutation matrices (Ps) computed by different precision versions may be different.

Note that in our experiments in §4.1 and §4.2 we only compute relative errors. However, we also performed limited experiments showing that BGRT also successfully generates higher absolute errors than URT. Table 11 gives the computed absolute errors for our parallel benchmarks with time out set to one hour. BGRT still returns the higher errors than URT in most of the experiments.

4.3 Discussion

From our experimental results we can observe that BGRT found the highest floating-point error 39 times out of 48 experiments total. The intuition behind such success of BGRT is that it could often "zoom in" to a certain configuration that potentially causes high floating-point error. By observing Figs. 3d and 3e, we could make a hypothesis: bad inputs (inputs that result in low errors) and good inputs (inputs that result in high errors) are not equally spaced in the input domain. In other words, for a configuration, there may exist a tighter configuration that contains many good inputs. On the other hand, there may exist a tighter configuration that contains many bad inputs. BGRT iteratively applies this hypothesis and iteratively searches for "better" configurations Our experimental results empirically show that the intuition is useful for measuring floating-point error in practice. However, ILS and PSO do not follow this hypothesis: when they encounter a configuration that is evaluated to high error, they try to find a similar, but not a tighter configuration, to explore. (Two configurations are similar if they have the same domain and map most of the input variables to the same ranges. But for those input variables which are mapped to different ranges, those ranges are mutually exclusive.)

Finally, note that the algorithm tuning parameters of BGRT, ILS, and PSO in our experiments were manually explored (e.g., k and N_{part} in BGRT, see Algo. 2). In particular, we manually varied parameter settings for the GRT techniques and chose the best one for our experiments. Admittedly, applying different parameter settings to different GRT techniques could give them different capabilities of generating floating-point errors. We plan to investigate automatic methods for tuning GRT parameters in our future work.

5. Applications

Combining Random Testing with Automatic Performance Tuning. Automatic tuning tools (e.g., [28, 36]) increase program performance by lowering floating-point bit-width of input variables or instruction operands while at the same time satisfying a user-provided precision constraint. The constraint is usually in the form of an upper bound on floating-point error. A common approach to checking whether a tuned program satisfies the precision constraint is to concretely execute it with user-given or randomly generated inputs. However, such inputs may not result in highly imprecise outputs of the program. Consequently, the tuned programs may frequently violate the expected precision constraint in practice. Our

	Algo.	FFT (512) abs. error	LU (256) abs. error	QR (256) abs. error	MM (770) abs. error
Exp_1	NRT	9.8124e-05	2.7483e-05	2.0435e-03	5.7346e-01
	BGRT	**1.2366e-04**	**2.9065e-04**	**1.3575e-02**	**3.3573e+01**
Exp_2	NRT	**1.0980e-04**	2.6175e-05	5.0654e-03	6.8767e-01
	BGRT	1.0325e-04	**1.3049e-03**	**2.8785e-02**	**3.1175e+00**

(a) Small Input Size

	Algo.	FFT (2048) abs. error	LU (1024) abs. error	QR (1024) abs. error	MM (3074) abs. error
Exp_1	NRT	1.5593e-04	**7.3125e-04**	**4.1454e-02**	1.6142e+00
	BGRT	**1.6951e-04**	3.5286e-04	1.1190e-02	**8.6679e+00**
Exp_2	NRT	1.5268e-04	9.5654e-05	1.9666e-03	1.2067e+00
	BGRT	**1.5341e-04**	**1.2441e+00**	**5.7901e-03**	**8.0260e+00**

(b) Large Input Size

Table 11: Absolute Error for Parallel Benchmarks. Time budget is set to one hour.

GRT approach could provide inputs that result in highly imprecise outputs to such automatic performance tuning tools. Hence, using GRT the final tuned programs are expected to violate the precision constraint in practice much less frequently.

Input Generation for Floating-point Error Detection. Recently proposed tools [5, 29] pin-point the critical instructions of a program that cause high floating-point errors. They employ shadow value execution and keep track of each value's shadow value. By comparing the original and shadow values they can find the source of floating-point imprecision. However, such technique highly depends on provided inputs: if the inputs result in precise outputs, the tools may not accurately point out the imprecision-causing instructions. Our GRT approach could enumerate inputs resulting in highly imprecise outputs, thereby increasing precision of floating-point error detection.

Algorithm Comparison. Mathematically analyzing numerical errors of algorithms is a tedious process, and it is not feasible that algorithm designers perform it for their every new algorithm. We believe that our random testing methods would provide the designers with solid empirical data about the numerical error of their algorithms. Publishing such data with an algorithm would in turn help programmers to select algorithms based on the numerical precision needs of their software. For example, Table 7 shows a comparison of three reductions: balanced (BR), imbalanced (IBR), and Kahan summation (IBR^K). The reductions calculate the summation of a set of 32-bit floating-point values. However, Kahan summation uses an additional 32-bit floating-point number to hold round-off value which "compensates" floating-point error. Such compensated summation is suggested to be a more precise way to summarize floating-point values [20]. Indeed, in Table 7 all random testing methods reported that Kahan summation has lower relative floating-point error than the other two. A programmer could rely on such data to make a more informative choice when exploring various trade-offs involving floating-point precision. We plan to investigate how to use GRT to more accurately compare numerical precision among various classes of algorithms.

6. Future Work

Combining Random Testing with Input Constraint Generation. A weakness of our current GRT methods, including BGRT, ILS, and PSO, is that they do not guarantee path coverage. In other words, given a set of randomly generated inputs, there may exist some program paths which are not traversed using these inputs. However, if there exists a path which could generate highly imprecise results, it is highly likely to be traversed sufficiently many times by our methods. Otherwise, the final floating-point error of the program may be heavily under-approximated. (Figs. 3d and 3e suggest that the fewer traversals of a path, the lower the floating-point error detected from it will be.)

A potential approach to improving path coverage of GRT methods is combining them with weakest precondition [21] inference, and related symbolic input generation techniques [8]. These techniques can help GRT methods generate equal number of inputs for each path, or even bias toward some designated paths. For example, one can generate input constraints that help reach specific program points. The difficulty of reasoning about floating-point values (not well-supported by current SMT tools) may be overcome by upcoming progress in SMT, and perhaps also by approximating the reasoning (perhaps sufficient to bias search).

Fair Floating-point Range Division. BGRT, ILS, and PSO divide floating-point ranges into sub-ranges with various granularity. For example, when generating the "upper half" of a configuration (see §3.4), we divide each range in half. For example, we divide $[2.0, 32.0]$ into $[2.0, 17.0]$ and $[17.0, 32.0]$. However, based on IEEE-754 standard [23], floating-point numbers are not equally spaced between the largest and the smallest floating-point numbers. Hence, the number of floating-point numbers in $[2.0, 17.0]$ is not equal to the number of floating-point numbers in $[17.0, 32.0]$. (In fact, $[2.0, 17.0]$ contains more floating-point numbers.) A side-effect of such unfair division is that we may bias some floating-point numbers to be chosen as input values. A *fair* division on $[2.0, 32.0]$ is $[2.0, 8.0]$ and $[8.0, 32.0]$ such that the two sub-ranges contain the same amount of floating-point numbers. Our preliminary experiments are not conclusive to show that such fair division could result in finding higher floating-point error—we are planning to explore this direction further.

Compositional Random Testing. Random testing could sometimes be expensive, in particular when even one run of the program under test takes a long time. In addition, input size could be very large as well. For example, Table 12 shows our experimental results for the parallel FFT from the NAS Parallel Benchmarks (NPB), MPI version [35]. The input of this benchmark contains about 0.5 million floating-point numbers, while the number of shadow value executions performed in an hour is about $8.5 * 10^2$. Hence, only a tiny fraction of the input space can be covered in an hour, which is sometimes not enough to find a satisfactory floating-point error. These are some challenges for GRT to detect high floating-point error for large-scale programs. A potential solution is to extract critical functions using a custom static analysis, and then focus RT only on those. For example, a large FFT program (Cooley-Tukey FFT [19]) recursively breaks down discrete Fourier transforms (DFTs) into smaller DFTs, and then gathers their results. We could potentially precisely measure the floating-point error for DFT, and leverage it to compositionally estimate the floating-point error of the whole program. We will further investigate such compositional GRT approaches for handling large scale programs.

Acknowledgements

Thanks to Alan Humphrey, Brandon Gibson and Martin Berzins for providing us the DQMOM benchmark from the Uintah Framework. Chiang was supported in part by an Nvidia Graduate Fellowship and Gopalakrishnan by the SUPER (super-scidac.org). Additional support was provided by NSF grants ACI 1148127, CCF 1255776, CCF 1302449 and CCF 1346756.

	Algo.	FFT (524288)			
		rel. err.	# SVE	|abs. err.|	# SVE
Exp_1	URT	**4.9803e-07**	870	8.8437e+00	880
	BGRT	4.6859e-07	830	**1.6545e+01**	790
Exp_2	URT	**5.6224e-07**	870	**9.9878e+00**	870
	BGRT	5.3524e-07	840	9.6160e+00	760

Table 12: Experimental Results on a Large Scale Program: Parallel FFT of the NAS Parallel Benchmarks (NPB)

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (3rd Ed.)*. SIAM: Society for Industrial and Applied Mathematics, 1999.

[2] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 38–49, 2009.

[3] D. H. Bailey and J. M. Borwein. Highly parallel, high-precision numerical integration, 2005.

[4] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 549–560, 2013.

[5] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 453–462, 2012.

[6] S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *IEEE Symposium on Computer Arithmetic*, pages 243–252, 2011.

[7] S. Boldo, J.-C. Filliâtre, and G. Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *Intelligent Computer Mathematics*, pages 59–74. Springer-Verlag, 2009.

[8] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, Feb. 2013.

[9] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamarić, D. H. Ahn, and G. L. Lee. Determinism and reproducibility in large-scale HPC systems. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2013.

[10] Coq. The Coq proof assistant. `http://coq.inria.fr/`.

[11] E. Darulova and V. Kuncak. Trustworthy numerical computation in Scala. In *Conference on Object-Oriented Programming Systems (OOPSLA)*, pages 325–344, 2011.

[12] F. de Dinechin, C. Q. Lauter, and G. Melquiond. Assised verification of elementary functions using Gappa. In *ACM Symposium on Applied computing (SAC)*, pages 1318–1322, 2006.

[13] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[14] 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 *Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, pages 53–69, 2009.

[15] J. Demmel and H. D. Nguyen. Numerical reproducibility and accuracy at exascale. In *IEEE Symposium on Computer Arithmetic*, pages 235–237, 2013.

[16] J. W. Demmel. Trading off parallelism and numerical stability. Technical Report UCB/CSD-92-702, EECS Department, University of California, Berkeley, 1992.

[17] A. Goodloe, C. Muñoz, F. Kirchner, and L. Correnson. Verification of numerical programs: From real numbers to floating point numbers. In *NASA Formal Methods Symposium (NFM)*, pages 441–446, 2013.

[18] M. Harris. *Optimizing parallel reduction in CUDA*. NVIDIA Developer Technology, 2007.

[19] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast Fourier transform. *ASSP Magazine, IEEE*, 1(4):14–21, 1984.

[20] N. J. Higham. *Accuracy and Stability of Numerical Algorithms (2nd Ed.)*. SIAM: Society for Industrial and Applied Mathematics, 2002.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[22] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *National Conference on Artificial Intelligence*, pages 1152–1157, 2007.

[23] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, 1985.

[24] F. Ivancic, M. K. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *IEEE/ACM Intl. Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 49–58, 2010.

[25] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Communications of the ACM*, 8(1):40–, Jan. 1965.

[26] J. Kennedy. Particle swarm optimization. In *Encyclopedia of Machine Learning*, pages 760–766. Springer-Verlag, 2010.

[27] A. B. Kinsman and N. Nicolici. Finite precision bit-width allocation using SAT-modulo theory. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 1106–1111, 2009.

[28] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *International Conference on Supercomputing (ICS)*, pages 369–378, 2013.

[29] M. O. Lam, J. K. Hollingsworth, and G. W. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39(3):146–155, 2013.

[30] P. Li, G. Li, and G. Gopalakrishnan. Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 29:1–29:10, 2012.

[31] M. D. Linderman, M. Ho, D. L. Dill, T. H. Y. Meng, and G. P. Nolan. Towards program optimization through automated analysis of numerical precision. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 230–237, 2010.

[32] H. R. Lourenço, O. C. Martin, and T. Stutzle. Iterated local search. *arXiv preprint math/0102188*, 2001.

[33] J. Luitjens, B. Worthen, M. Berzins, and T. C. Henderson. *Petascale Computing Algorithms and Applications*, chapter Scalable parallel AMR for the Uintah multiphysics code, pages 67–82. Chapman and Hall/CRC, 2007.

[34] MAGMA. The MAGMA library. `http://icl.cs.utk.edu/magma/index.html`.

[35] NPB. The NAS parallel benchmarks (NPB). `http://www.nas.nasa.gov/publications/npb.html`.

[36] 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 *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 27:1–27:12, 2013.

[37] S3FP. Guided random testing for floating-point error estimation. `https://sites.google.com/site/grt4fperror/`.

[38] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu. CORAL: Solving complex constraints for symbolic Pathfinder. In *NASA Formal Methods Symposium (NFM)*, pages 359–374, 2011.

[39] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.