Practical Floating-point Divergence Detection*

Wei-Fan Chiang, Ganesh Gopalakrishnan, and Zvonimir Rakamarić

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

Abstract. Reducing floating-point precision allocation in HPC programs is of considerable interest from the point of view of obtaining higher performance. However, this can lead to unexpected behavioral deviations from the programmer's intent. In this paper, we focus on the problem of *divergence detection*: when a given floating-point program exhibits different control flow (or differs in terms of other discrete outputs) with respect to the same program interpreted under reals. This problem has remained open even for everyday programs such as those that compute convex-hulls. We propose a classification of the divergent behaviors exhibited by programs, and propose efficient heuristics to generate inputs causing divergence. Our experimental results demonstrate that our input generation heuristics are far more efficient than random input generation for divergence detection, and can exhibit divergence even for programs with thousands of inputs.

1 Introduction

Almost anyone writing a program involving floating-point data types wonders what precision to allocate (single, double, or higher). There is a great temptation to get away with single precision, as it can yield performance advantage of a factor of 2.5 for CPU codes [20] or even higher for GPU codes [29,21]. Yet, floating-point arithmetic is highly non-intuitive, causing non-reproducible bugs and nighmarish debugging situations [28,25,8,10,24]. For instance, experts in a recent project had to waste several days chasing a Xeon vs. Xeon-Phi floating-point behavioral deviation where identical *source* code running on these machines took different control paths for the same input [22].

Any program in which floating-point results flow into conditional expressions can decide to take different control paths based on floating-point round-off. Also, if a developer banks on a program meeting a specific post-condition, they may find that the truth of the post-condition can depend again on floating-point round-off. Such divergent behaviors ("divergence") have been widely discussed in the literature. Kettner et al. [18] demonstrated that a geometric convex hull construction algorithm can result in non-convex hulls under certain (manually generated) inputs. Problems due to inconsistency in geometric computations are

* Supported in part by NSF awards CCF 1421726 and ACI 1535032, and also performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-669095).

```
1: float8 a, b, c, d, e, f;
2: procedure FOO : → int
3:     if (a+b)+c > (d+e)+f then
4:         return 22;
5:     else
6:         return 33;
7:     end if
8: end procedure
```

(a) Illustrating Divergence and ABS

```
1: float8 x, y;
2: procedure VAR : → float8
3:     var = (x^2+y^2)/2−((x+y)/2)^2;
4:     return var;
5: end procedure
6: Desired Post-condition: var ≥ 0;
```

(b) Illustrating GRT

Fig. 1: Motivating Examples

described in great detail in the context of computer games [12]. The author of [27] suggests the use of "padding constants" followed by "thorough testing" as a practical solution to guard against the nasty surprises of divergence. With the increasing role of geometry in real life (e.g., manufacture of prosthetics using 3D printing, computer gaming, mesh generation, robot motion planning), divergence becomes a life-critical or resource-critical issue, and must be systematically tackled. While allocating higher floating-point precision can reduce the incidence of divergence, a programmer will not go this route (and suffer a slowdown) unless they have at least one piece of evidence (in the form of an input causing divergence – hereafter called *diverging input*) that this measure is necessary. We show that without systematic testing one can fail to find even one divergent behavior for many programs.

The hopelessness of manual reasoning can be highlighted through the program in Fig. 1a where, for simplicity, we consider 8-bit floating-point values. Let *float8* have one sign bit s, four bits of mantissa (or precision) m, and three bits of exponent e representing the value $(-1)^s \cdot 1.m \cdot 2^{e-4}$. One may not be surprised if told that this program may return 33 (under standard IEEE floating-point with round to nearest applied), while returning 22 if we used reals instead of float8. However, it is virtually impossible to manually obtain even one diverging input. [1] Purely random testing is ineffective for exhibiting divergence due to the huge state space it faces. There has been very little prior work on efficiently identifying diverging inputs. In this paper, we propose and evaluate methods to rapidly discover diverging inputs for many useful floating-point primitives. Such primitives are known to be used in many applications—for example, mesh generation.

We focus on the problem of developing efficient heuristics to generate diverging inputs. We assume that the user has identified *discrete features* (e.g., returning 22 or 33 in our example) as one of the key observable results from the program.[2] Our algorithms then generate diverging inputs in a lower precision (say, 32-bit) computation.

[1] $a = e = d = 1.0000 \cdot 2^{-3}$, $b = f = 1.0000 \cdot 2^2$, and $c = 1.0001 \cdot 2^{-3}$ causes divergence.

[2] Real arithmetic is simulated by allocating very high precision. Typically we aim for 64- or 128-bit precision.

While all divergences are attributable to some deviation in conversion of floating-point to discrete value such as deviation in control-flow, it is also well known (e.g., [7]) that many control-flow deviations do not cause divergence. In this paper, we describe divergence detection with respect to the user-given set of discrete features. Clearly, control-flow deviation is an extreme case in our definition: the discrete feature in that case is nothing but the full control-flow path.

The difficulty of identifying diverging inputs is due to (1) the sheer number of input combinations to be considered, (2) non-uniformity of floating-point number distribution, (3) the layers of floating-point operations (e.g., non-linear and transcendental operators and their associated rounding modes, catastrophic cancellations during subtraction [14]) that are involved before a conditional expression's truth value is determined, and (4) poor scalability of symbolic methods since floating-point arithmetic decision procedures are in their infancy. While our previous work [9] helps identify inputs that cause high round-off errors in floating-point functions, such methods cannot be directly used to identify diverging inputs. In this paper, we present an approach that addresses these difficulties by employing empirical search methods to efficiently discover diverging inputs. Ours is the first attempt to classify problems in this area into discernible groups, and provide heuristic approaches for input generation to trigger divergence. Our contributions in this paper are the following:

- Two approaches to trigger divergence in programs of interest to practitioners.
- A classification of programs into two categories, with corresponding new heuristics to trigger divergence in each category.

2 Overview of our Approach

Given a program P and its input i, let $P_R(i)$ indicate the result of running the program on i under real number arithmetic. For simplicity, let vector i capture both the "data input" and "initial program state" of P. Let P_F be the floating-point version of P_R. We are interested in those inputs i under which $P_F(i) \not\equiv P_R(i)$, where $\equiv$ is some coarse equivalence relation since a programmer may not want bit-for-bit equality, but rather something higher level. We define $\equiv$ with the help of an abstract state space $A \subseteq U$ for some universe U, and an abstraction map α that maps into U. Then, a computation is divergent when $\alpha(P_F(i)) \neq \alpha(P_R(i))$.

Example 1: In the example of Fig. 1a, the relevant abstract state space is given by $A = U = \{22, 33\}$; we call the members of A discrete features (or discrete *signatures*). The input $a = e = d = 1.0000 \cdot 2^{-3}$, $b = f = 1.0000 \cdot 2^2$, and $c = 1.0001 \cdot 2^{-3}$ causes divergence. We now introduce our first search method called *abstract binary search* (ABS), which works as follows:

- We first use random testing to generate inputs i_1 and i_2 with signatures S_1 and S_2 under floating-point such that $S_1 \neq S_2$. In Fig. 1a, i_1 may under float8 result in signature 22 and i_2 in signature 33. Suppose i_1 results in 22 and i_2 in 33 under reals as well, and hence this is not a divergent situation.

- We use the discovered pair $\langle i_1, i_2 \rangle$ to bootstrap the binary search part of ABS. We compute the midpoint $mid = (i_1 + i_2)/2$ (taking /2 as a suitable way of finding the midpoint of two N-dimensional points) and proceed recursively with $\langle i_1, mid \rangle$ and $\langle i_2, mid \rangle$ as new pairs of inputs (details in Algorithm 1).
- If/when the floating-point signature output generated for *mid* differs from its real signature, we have located a diverging input and the algorithm terminates.

Example 2: We now introduce our second search method called *guided random testing* (GRT). Fig. 1b computes the variance of x and y in terms of "mean of squares minus square of mean." Variances are non-negative, as captured by the given post-condition. We therefore choose $U = \{T, F\}$ to model Boolean truth, and $A = \{T\}$ to represent when the desired post-condition holds. In more detail, we have observed that for many problems (examples given in §4.2), the desired post-condition is of the form $(e_1 \geq 0) \ \wedge \ (e_2 \geq 0) \ \ldots \ (e_n \geq 0)$, where e_i are expressions. GRT chooses one $e_i \geq 0$ conjunct, and it attempts to generate inputs that falsify it under floating-points (all conjuncts are assumed to be true under reals). In §3.2, we present a heuristic based on *relative errors* that helps find such inputs.

ABS vs. GRT: We recommend the use of GRT whenever a post-condition (always true under reals) has a chance of being violated under floating-points. On the other hand, ABS is recommended whenever such a post-condition does not exist, and one can bootstrap the process by quickly finding input i_1 and i_2 causing unequal signatures S_1 and S_2 under floating-points. The examples in this paper clarify further how we choose between these two search methods. We consider a more involved example next.

Example 3: Let P be a program computing a convex hull for a collection of points i. First, consider a simple case where i consists of five 2D points $\{\langle 0,0 \rangle, \langle C,0 \rangle, \langle C,C \rangle, \langle C,2C \rangle, \langle 0,2C \rangle\}$, where C and $2C$ are representable in floating-points. A convex hull algorithm is correct if the hull it returns is convex and encloses all the points (i.e., no point lies outside). According to this definition, there are two correct answers in this case:

- $\{\langle 0,0 \rangle, \langle C,0 \rangle, \langle C,C \rangle, \langle C,2C \rangle, \langle 0,2C \rangle\}$ or
- $\{\langle 0,0 \rangle, \langle C,0 \rangle, \langle C,2C \rangle, \langle 0,2C \rangle\}$.

In this example, one could either choose an exact listing of coordinates as the signature, or a more abstract signature such as the number of vertices in the convex hull. Whatever be our choice of signatures, we reiterate that in our approach (1) signatures are the only means of observing program behavior, (2) the signature returned by the real-valued computation is taken as the golden truth, and (3) divergence exists when the floating-point computation returns a different signature than the real computation.

Let the chosen signature be the number of vertices in the convex hull. Consider the input $\{\langle 0,0 \rangle, \langle C,0 \rangle, \langle C-\delta,C \rangle, \langle C,2C \rangle, \langle 0,2C \rangle\}$, where δ is very small, but $C-\delta$ is still representable in floating-points. For this input, our convex hull program returns 4 as the signature under reals. However, it may return 5 under

floating-points due to round-off errors. This is an example of a divergent input according to our definition. We now summarize some of our observations:

- Signatures are a mechanism to mimic the "desired output."
- Some signatures (e.g., the count of the number of vertices in a convex hull) are *strong*, in that we observe empirically that one can arrive at diverging inputs fairly quickly.
- One can always choose the entire taken control-flow path as a signature. We empirically show that such signatures are typically *weak*, meaning that they make locating divergences very hard. Our experience shows that a good signature must ideally be a mapping into a small abstract space A.

3 Methodology

Given a program P and its input domain $\mathbb{I}$, we assume that each input $i \in \mathbb{I}$ is a scalar vector and $\mathbb{I}$ is convex. Let $i_X, i_Y \in \mathbb{I}$ be two inputs; then $\mathbb{I}$ is convex if all inputs *in-between* are also in $\mathbb{I}$: $\forall 0 \leq k \leq 1 \,.\, i_X * k + i_Y * (1-k) \in \mathbb{I}$. The *midpoint* of i_X and i_Y is obtained by setting k to 0.5. A *signature*, as mentioned in the previous sections, is a discrete feature (or abstraction) of the concrete program output. Then, a *signature function* α maps program outputs under either floating-point or real arithmetic executions to abstract signature states.

For every input $i \in \mathbb{I}$, the output $P_F(i)$ of an execution under floating-points is supposed to have the same signature as the output $P_R(i)$ of an execution under reals. The *differential contract* of a program, which specifies when a divergence exists, is defined using the following predicate:

$$div(P, i) =_{def} \alpha(P_F(i)) \neq \alpha(P_R(i)). \tag{1}$$

Predicate *div* states that a divergence occurs when the signatures of real and floating-point outputs (i.e., executions) differ.

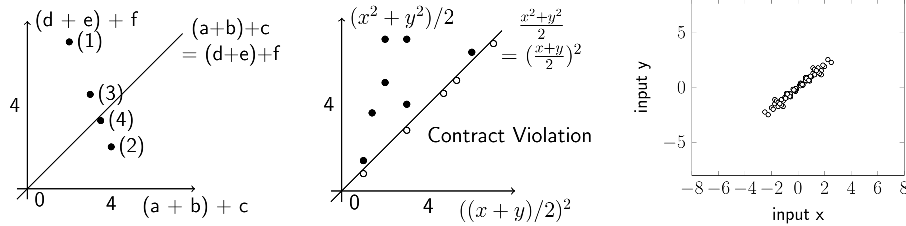
3.1 Abstract Binary Search (ABS)

Fig. 2a illustrates how ABS detects divergence for the program in Fig. 1a. Here, the x-axis shows the values of $(a+b)+c$ and the y-axis the values of $(d+e)+f$. The diagonal separates the abstract signature state spaces of 22 and 33. ABS first finds a vector i_1 of values for inputs $a, \ldots, f$, whose abstract signature is 33 under both reals and floating-points (shown as point (1)). Then, it finds a vector i_2 whose abstract signature is 22 under both reals and floating-points (shown as point (2)). The pair $\langle i_1, i_2 \rangle$ is the input of the subsequent binary search. (Note that ABS is typically not applicable on examples where finding points as above is extremely difficult, as Fig. 2b illustrates.)

Our binary search method successively divides the N-dimensional space between vectors i_1 and i_2 by finding midpoints of these vectors (points (3) and (4) depict this N-dimensional binary search). It is highly unlikely that all points in this search sequence would all evaluate to the same abstract state under reals

Algorithm 1 Abstract Binary Search (ABS)

1: **procedure** ABS(P, α, $\mathbb{I}$)
2: **repeat** ▷ Find starting pair of points
3: $i_A = Random(\mathbb{I})$
4: **if** $div(P, i_A)$ **then return** i_A
5: $i_B = Random(\mathbb{I})$
6: **if** $div(P, i_B)$ **then return** i_B
7: **until** $\alpha(P_F(i_A)) \neq \alpha(P_F(i_B))$
8: $E = \{\langle i_A, i_B \rangle\}$ ▷ Bootstrap binary search between end points
9: **while** $E \neq \emptyset$ **do**
10: $\langle i_X, i_Y \rangle = Select(E)$
11: $E = E \setminus \{\langle i_X, i_Y \rangle\}$
12: **if** $\exists i_M$ midpoint of $\langle i_X, i_Y \rangle$ distinct from i_X, i_Y **then**
13: **if** $div(P, i_M))$ **then return** i_M
14: **if** $\alpha(P_F(i_X)) \neq \alpha(P_F(i_M))$ **then**
15: $E = E \cup \{\langle i_X, i_M \rangle\}$
16: **end if**
17: **if** $\alpha(P_F(i_M)) \neq \alpha(P_F(i_Y))$ **then**
18: $E = E \cup \{\langle i_M, i_Y \rangle\}$
19: **end if**
20: **end if**
21: **end while**
22: **restart search** ▷ Optional restart step
23: **end procedure**



(a) Detecting Divergence in Fig. 1a using ABS

(b) Contract- and Non-violating Points for Fig. 1b

(c) Distribution of Diverging Inputs for Fig. 1b

Fig. 2: Applying ABS and GRT on Examples from Figs. 1a and 1b

and floating-points. This is because it is also unlikely that the evaluations of the constituent expressions in the given program under reals, and their corresponding evaluations under rounding, would track each other perfectly with respect to the chosen discrete features.[3] As a consequence, ABS eventually encounters a point (akin to point (4)) that lies at the borderline of the abstract spaces and causes a divergence.

The efficiency of ABS heavily depends on the chosen signature function. It must be possible to efficiently find — through a few random sampling steps — the initial points i_1 and i_2 that map to distinct abstract states. For example, it must be possible to efficiently locate points (1) and (2) in Fig. 2a.

[3] In practice, we do occasionally encounter a sequence whose discrete signatures match perfectly, and ABS exhausts all possible midpoints. In such cases, ABS is restarted with a different random seed.

Algorithm 1 gives pseudocode of ABS. As input it takes a program P, a signature function α, and the input domain $\mathbb{I}$, and it outputs a divergence-inducing input vector. The first phase of ABS (lines 2–7) finds the initial pair of points $\langle i_A, i_B \rangle$ satisfying $\alpha(P_F(i_A)) \neq \alpha(P_F(i_B))$ by employing random sampling. The second phase (lines 8–21) successively subdivides the space between a pair of points by removing a pair $\langle i_X, i_Y \rangle$ from E and seeking a divergence-inducing midpoint i_M. Under floating-point arithmetic, i_M can be equal to i_X or i_Y, which means we exhausted all midpoints, but could optionally restart. Otherwise, we determine which of the pairs $\langle i_X, i_M \rangle$ or $\langle i_M, i_Y \rangle$ are eligible for further search, and we add them to E. The while-loop of the second phase is guaranteed to terminate because floating-point domains are finite. The ABS procedure either returns a divergence-inducing input vector or timeouts (with the optional restart at line 22 using a different random seed). We rarely encountered timeouts in our empirical evaluation.

3.2 Guided Random Testing (GRT)

We designed GRT based on a key observation: a divergence occurs when one of the expressions in the signature has a high relative error; we now detail why this is so. The relative error of a value v is defined as $|\frac{v_R - v_F}{v_R}|$ [14]. In the example from Fig. 1b, the relative error of *var* must be high, and specifically greater than 1, when a negative variance is returned. Fig. 2b illustrates that this is very rare by showing the space of contract (post-condition) violations and the space where the contract is met. Similarly, Fig. 2c shows all the diverging inputs obtained using a *satisfiability modulo theories* (SMT) solver [11].[4]

Let var_R (resp. var_F) be the variance computed under real (resp. floating-point) execution. Then, a divergence occurs when $(var_R \geq 0) \wedge (var_F < 0)$, which implies $var_R - var_F > var_R$. Thus, the relative error on var must exceed 1, meaning $|\frac{var_R - var_F}{var_R}| > 1$.

We have found that many problems amenable to GRT have a post-condition expressible as a conjunction $(e_1 \geq 0) \wedge (e_2 \geq 0) \wedge \ldots \wedge (e_N \geq 0)$, where e_i is a floating-point expression. Given such a formula, GRT aims to negate one of the conjuncts using guided random search. For this purpose, we employ our S3FP tool that efficiently maximizes a relative error of a given expression [9]. We now detail two approaches we investigated to drive the optimization.

Single-term Objective: Choose one e_i from $e_1 \ldots e_N$ as the objective for triggering high relative error.

Multi-term Objective: Maximize $\sum_{i=1}^{N} err(e_i)$ such that

$$err(e_i) = \begin{cases} |rel_err(e_i)| & : |rel_err(e_i)| < 1 \\ 1 & : otherwise \end{cases},$$

where $rel_err(e_i)$ is the relative error of expression e_i.

[4] Current state-of-the-art SMT solvers work on micro-benchmarks with micro floating-point formats such as the program in Fig. 1b; they still cannot handle realistic floating-point programs used in our work.

Algorithm 2 Guided Random Testing (GRT)

```
1: procedure GRT(P, α, 𝕀)
2:     f_obj = ExtractObjective(α)
3:     while ¬Timeout() do
4:         i = Optimizer(P, 𝕀, f_obj)
5:         if div(P, i) then return i
6:     end while
7: end procedure
```

Algorithm 2 gives the pseudocode of GRT where we assume existence of a suitable function *ExtractObjective* that realizes either the single-term or the multi-term objective. Note that it if often convenient to provide a signature function that only loosely specifies program contracts, and falsifying such a contract does not always imply divergence (an example is provided in §4.2). Hence, each input vector returned by the optimizer (S3FP in our case) has to be checked to establish divergence using predicate *div*.

4 Experimental Results

We have evaluated ABS and GRT on a collection of realistic numerical routines. These routines regularly find applications in implementations of higher level algorithms such as Delaunay triangulation (often used for mesh generation) and other operations in high-performance computing [6]. Divergence detection for all benchmarks is achieved using differential contracts as stated in Equation 1 and defined in §3. The only exception is the approximate sorting benchmark, which invokes an externally specified contract (see §4.2). As defined in §3, a differential contract is a comparison between signatures of outputs computed under reals and floating-points. We use high-precision floating-points to approximate reals, which is a technique employed in many floating-point analysis approaches (e.g., [3,9]). We categorize our benchmarks based on the signature model they follow.

4.1 ABS Benchmarks

Convex Hull: The algorithm takes a set of 2D points as input and outputs a 2D polygon. A coordinate of each point is a floating-point value in the range $[-100, 100)$. The generated (convex) polygon must encompass all input points; we take the polygon vertex-count as our signature. We study four convex hull algorithms: simple [4], incremental [18], quick-hull [5], and Graham's scan [15]. The quick-hull and Graham's scan algorithms were taken from CGAL [6], which is a popular open-source geometric computation library.
Shortest Path: We implemented the well-known Floyd-Warshall shortest path algorithm [13], which calculates all-pair shortest paths for a graph. Our implementation takes a complete directed graph as input, and outputs a boolean value indicating the existence of a negative cycle. The input graph is represented as

a sequence of floating-point edge-weights in the range $[-1, 10)$. The signature is the same as output: a boolean value indicating the existence of a negative cycle.
Intersection Between a 3D Line and Adjacent Triangles: This benchmark checks whether a 3D line intersects with two adjacent triangles. It takes six 3D points as input—four for the adjacent triangles and two for the line. The intersection scenario is one of the following types: the line (1) intersects with a triangle, (2) passes between the two triangles, and (3) neither. The signature indicates whether the intersection scenario is type (2), which is an unexpected scenario as described in related work [12]. This benchmark is taken from CGAL.
Geometric Primitives: These benchmarks, taken from CGAL, involve computing relationships between geometric objects, including a 2D triangle intersection test and several point-orientation tests. The triangle intersection test takes two 2D triangles as input, and determines if they intersect or not. Each point-orientation test takes a 2D/3D point and a 2D/3D shape as input, and determines if the point is inside/above or outside/below the shape. We collected four point-orientation tests: 2D point-to-triangle, 2D point-to-circle, 3D point-to-sphere, and 3D point-to-plane. All geometric primitives take a sequence of floating-point coordinates in the range $[-100, 100)$ as input. Their output is a boolean value indicating the relationship between geometric objects, which is also our chosen signature.

4.2 GRT Benchmarks

Variance Calculation: We implemented the naïve variance calculation, which is known to suffer from catastrophic cancellation effects [23]: $var(X) = E[X^2] - (E[X])^2$. Here, X is a random floating-point variable in the range $[-100, 100)$ and $var(X)$ is its variance. The post-condition states that the computed variance must be non-negative, and is captured with the signature $var(X) \geq 0$.
Exclusive Prefix Sum: The procedure takes an array $X_1, \ldots, X_N$ as input, and outputs a sequence of summations $Y_1, \ldots, Y_N$ such that $Y_1 = 0$ and $Y_i = \sum_{k=1}^{i-1} X_k$ for $2 \leq i \leq N$. If all input values are non-negative, exclusive prefix sum must output a monotonically increasing sequence. We implemented the naïve and two-phase scan [17] algorithms. We provide them with a sequence of floating-point values in the range $[0, 100)$ as input. Given output values $Y_1, \ldots, Y_N$, the post-condition is directly described in the signature function as:

$$(Y_2 \geq Y_1) \wedge (Y_3 \geq Y_2) \wedge \ldots \wedge (Y_N \geq Y_{N-1}). \tag{2}$$

Standard and Approximate Sorting: These benchmarks bubble-sort a sequence of N floating-point values obtained using a procedure that introduces round-off errors. More specifically, we generate each value in the sequence to be sorted by summing over N floating-point values in the range $[-100, 100)$. Standard sorting judges the output sequence as correct when it is strictly non-decreasing, whereas approximate sorting allows for a bounded degree of misorderings, defined as follows. Given an unsorted input $X = X_1, \ldots, X_N$ and a sorted output $Y = Y_1, \ldots, Y_N$, let $Z = Z_1, \ldots, Z_N$ be the permutation vector.

Benchmark	ISize	SRate	Samples	Restarts	RT
Conv. hull simple	200	10/10	3.21e+2	0.2	0
	2000	6/10	3.66e+2	0	N/A
Conv. hull simple (1 hr.)	2000	9/10	4.61e+2	0	N/A
Conv. hull simple (2 hr.)	2000	10/10	5.16e+2	0	N/A
Conv. hull incremental	200	10/10	2.65e+2	0.1	0
	2000	10/10	5.60e+2	0.1	0
Conv. hull quick-hull	200	10/10	3.03e+2	0.1	0
	2000	10/10	4.68e+2	0.2	0
Conv. hull Graham	200	10/10	2.26e+2	0.0	0
	2000	10/10	6.09e+2	0.2	1
Shortest path	90	10/10	2.43e+2	5.7	0
	2450	0/10	N/A	N/A	0
Shortest path with manual hint	2450	10/10	1.27e+2	2.4	0
Line × Adjacent Triangles	18	10/10	8.19e+2	15.7	0
Line × Adjacent Triangles with manual hint	18	10/10	6.24e+2	5.5	0
Tri. intersection	12	10/10	3.86e+1	0.3	0
Pt. triangle	8	10/10	8.43e+1	1.2	0
Pt. plane (3x3)	12	10/10	5.02e+1	0.7	0
Pt. plane (4x4)	12	10/10	6.11e+1	1.0	1
Pt. circle (3x3)	8	10/10	2.64e+1	0	0
Pt. circle (4x4)	8	10/10	3.70e+1	0.3	0
Pt. sphere (4x4)	15	10/10	3.05e+1	0.1	1
Pt. sphere (5x5)	15	10/10	3.33e+1	0.2	1

(a) Experimental Results for ABS

Benchmark	ISize	SRate	Samples	RT
Variance est.	1000	10/10	1.28e+3	0
	10000	10/10	6.10e+2	0
Naïve scan (single)	1024	10/10	2.55e+3	0
	8192	10/10	1.03e+3	0
Naïve scan (multi)	1024	0/10	N/A	0
	8192	0/10	N/A	0
Two-phase scan (single)	1024	0/10	N/A	0
	8192	0/10	N/A	0
Two-phase scan (multi)	1024	0/10	N/A	0
	8192	0/10	N/A	0
Standard sorting (single)	4096	10/10	7.25e+2	70
	10000	10/10	2.42e+2	220
Standard sorting (multi)	4096	10/10	5.08e+2	70
	10000	10/10	1.19e+2	220
Approx. sorting (single)	4096	9/10	2.15e+4	0
	10000	7/10	2.06e+4	0
Approx. sorting (multi)	4096	10/10	4.63e+3	0
	10000	10/10	1.89e+3	0

(b) Experimental Results for GRT. The two-phase scan is divergence-free. The results of the naïve scan and sorting show the difference between selecting the single-term (*single*) and the multi-term (*multi*) objectives for optimization. The random testing results are the same for the two objectives.

Table 1: Experimental Results. *ISize* is the input size (i.e., the number of input floating-point values); *SRate* is the number of divergences detected in ten runs (each run either finds a divergence or timeouts after 30 minutes); *Samples* is the average number of inputs enumerated to trigger the first divergence, computed over runs that successfully found one (*N/A* denotes experiments that fail in all runs); *RT* is the number of divergences triggered using 1 million random inputs; *Restarts* is the average number of restarts over 10 runs of ABS.

For example, if $X = \langle 7, 6, 8, 5\rangle$ and $Y = \langle 5, 6, 7, 8\rangle$, then $Z = \langle 3, 2, 4, 1\rangle$. Let Z_F be the permutation vector under floating-points and Z_R under reals. We define the degree of misorderings d_{mis} as the mean-square of $Z_R - Z_F$. Then, our post-condition for approximate sorting is $d_{mis} \leq \sqrt{N}$. For standard sorting, our post-condition is $Y_1 \leq Y_2 \leq \ldots \leq Y_N$. We define a common signature function as Equation 2.

For both types of sorting we use the above conjunctive signature. Hence, signature violations do not necessarily lead to post-condition violations for approximate sorting. Thus, an additional divergence check $d_{mis} < \sqrt{N}$ is required to confirm the inputs violating the differential contract. We call this additional divergence check an *externally specified contract*.

4.3 ABS Results

Table 1a shows our experimental results for ABS. Each run of ABS can restart multiple times to find an initial pair of points. All our experiments were performed on a machine with 12 Intel Xeon 2.40GHz CPUs and 48GB RAM. (We currently use only one processor of this multi-processor machine; parallelizing ABS and GRT is future work.) We measure the efficiency of ABS using the number of inputs enumerated to trigger the first divergence within 30 minutes. To measure scalability, we experiment with large program inputs (thousands of input variables). Our experiments show that ABS efficiently detects divergences by enumerating just a few hundreds of inputs even for large input sizes. Furthermore, ABS usually restarts only a few times to find initial end points.

Discussion: As expected of dynamic analysis techniques that need to repeatedly execute programs, practical efficiency of our divergence detection methods is related to execution times of programs under test. For example, simple convex hull is an $O(N^3)$ algorithm, and its execution time becomes very long for large input sizes. Hence, ABS detected only 6 divergences over 10 runs for Conv. hull simple with 2000 input variables. When given extra time, more runs of ABS successfully detect divergences: Conv. hull simple (1 hr.)/(2 hr.) in Table 1a denotes the result of running ABS for 1/2 hour(s).

ABS uses random search to find initial end points (see Algorithm 1), but programmers can provide hints to facilitate search. For our shortest path benchmark with input size of 2450, ABS failed to detect divergences in all runs since it failed to find initial end points: all randomly sampled inputs contained negative cycles. However, it is easy to manually provide an input which does not contain a negative cycle by assigning positive values to all edges' weights. Using this simple hint, ABS successfully detected divergences even in this case (see *Shortest path with manual hint* in Table 1a). Applying manual hints also improves ABS's efficiency of divergence detection. For our intersection check benchmark, we provided ABS with a manual hint causing different triangles to be intersected by the line (see *Line* $\times$ *Adjacent Triangles with manual hint* in Table 1a). Compared to the result that uses random search (see *Line* $\times$ *Adjacent Triangles*), with the manual hint ABS spent fewer enumerations to detect divergences.

Weak Signature Functions: Both ABS and GRT expect signature functions that satisfy Equation 1. However, both methods could work even with signature functions that do not satisfy this equation. For example, for the convex hull benchmarks we used a signature function that generates a boolean vector recording the point-orientation decisions made in the process of generating a convex hull. We call such signature functions *weak signature functions*, while those satisfying the equation are *strong signature functions*. Fig. 3 shows the comparison between using the two types of signatures. The black bars indicate the usage of a strong signature function (the number of the output hull vertices), and the white bars the usage of a weak signature function (the decision sequence). The shorter the bar, the fewer inputs enumerated by ABS were required to trigger the first divergence, implying better efficiency of divergence detection. Our results show

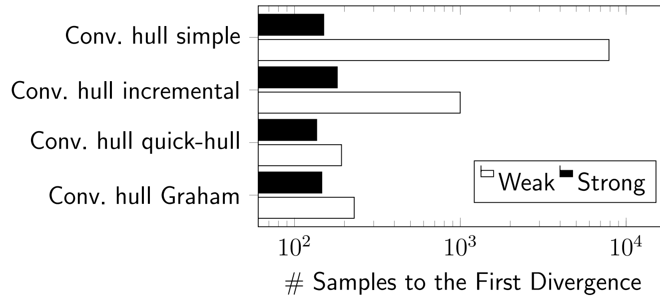


Fig. 3: Comparison of Using Strong and Weak Signature Functions with ABS

that ABS can work with weak signature functions, but the efficiency is lower than when using the strong ones.

4.4 GRT Results

Table 1b shows our experimental results for GRT. Benchmarks labeled with *single/multi* denote the single-/multi-term objective applied to our optimizer (as described in §3.2). The S3FP optimizer we use is an efficient tool for triggering high round-off errors [9]. Note that the random testing results under both objectives are the same because objective selection and random testing are independent. GRT detects divergences in all our benchmarks except the two-phase exclusive scan, which is expected since it is a divergence-free benchmark. The results also suggest that GRT is scalable since it can handle large input sizes. It is more efficient than random testing even for standard sorting. For example, for the standard sorting with 256 input variables, GRT enumerates 5590 inputs on average to trigger a divergence, while random testing needs over 300, 000 inputs.
Discussion: While our dynamic analysis methods can precisely detect diverging inputs, they cannot establish divergence-freedom when divergence is not possible in any feasible execution. For example, even though two-phase scan is a divergence-free algorithm when inputs are non-negative, we cannot infer that solely because GRT did not detect a divergence. (We omit the proof in this paper; it can be done by a simple induction.) Automatically proving divergence-freedom can be done using static analysis techniques [7], which is complementary to our work.

The results of the naïve scan and approximate sorting benchmarks indicate that the GRT's efficiency can be affected by optimization objective selection. Applying the single-term objective to the naïve scan can successfully detect divergences in all runs. On the other hand, applying the multi-term objective resulted in no detected divergences. However, the results for the approximate sorting show the opposite: applying the multi-term objective found more divergences than the single-term objective (29 over 30 runs versus 20 over 30 runs). As future work, we plan to explore heuristics for choosing a good optimization objective.

4.5 Random Testing

To demonstrate efficiency, we compare our methods with random testing, which is, to the best of our knowledge, the only divergence detection approach available to today's designers. In all our experiments, we randomly generated one million inputs for each input size, and column *RT* in Table 1 gives the number of divergences detected. At most one divergence was triggered in most of the benchmarks except the standard sorting. The results for *Conv. hull simple* with the input size of 2000 are not available because the execution time is very high (one million executions can take more than a week to finish). Our random testing results suggest that divergence is very difficult to detect without applying good search strategies such as ABS and GRT.

5 Related Work

In [7], the authors propose a verifier that attempts to prove that a set of user-specified axioms (e.g., Knuth's axioms for convex hulls [19]) cannot be violated by any control-flow path in a piece of code. Their work does not address floating-point directly; in fact, they treat all conditionals as non-deterministic selection statements, which can be unrealistic. Also, devising axioms for new problems is non-trivially hard. The scalability of their symbolic decision procedure is also in doubt (tool unavailable), and it can also generate false alarms (our method does not generate false alarms). Our approach is more practical, as it requires users to provide discrete features, and not an axiomatic specification.

Runtime instability detection could also be used to detect divergence [1]. This work does not address the task of generating inputs that can quickly induce divergence.

The authors of [16] propose a method for witnessing branch deviations across platforms. Their targeting problem is similar to the problem described in [22], and it is different from our targeting problem: witnessing discrete feature deviations between floating-point and real computations (which is called *divergence* in this paper). The key idea of their method is firstly using a SMT solver to find a candidate input, and then searching close inputs around the candidate and checking if any of them triggering a deviation. Our approach can address many practical scalability issues such as handling non-linear operations, and can be applied with equal ease even when source codes are unavailable.

White-box Sampling: White-box sampling [2] was proposed for finding discontinuous program behaviors. In this work, a program is seen as a composition of continuous functions which have disjoint input domains. White-box sampling tries to discover all continuous functions by finding at least one input for each of their input domains. The approach of checking whether two inputs belong to the same continuous function's domain is by comparing the decision sequences generated in the executions. A decision sequence is composed with floating-point-decided discrete values, called discrete factors, like branch decision and float-to-int type casting. Such discrete factor sequence can be one of the weak

signatures adopted by ABS (demonstrated in §4.3). Extracting discrete factor sequences from executions requires program instrumentation which is difficult to apply to large-scale programs (e.g. programs invoke dynamic linked libraries). However, ABS is not restricted to using discrete factor sequence as signature. ABS can treat programs as black boxes and adopt signature functions which directly observe program outputs.

Floating-point Program Testing Methods and Dynamic Precision Analysis: Both our divergence detection method and dynamic round-off error estimation [9] are methods for testing floating-point programs. However, dynamic round-off error estimation merely triggers high error on a given expression while ABS and GRT automatically find inputs that cause divergence. We can see the both divergence detection and round-off error estimation are two methods for finding inputs triggering floating-point imprecision scenarios. The inputs are important for dynamic floating-point analyses to avoid overly under-approximating floating-point imprecision. Examples of dynamic floating-point analyses which use concrete inputs to profile precision are catastrophic cancellation detection [3], instability detection [1], auto-tuning [25], and synthesis [26].

6 Concluding Remarks

With the increasing pressure to reduce data movement, reducing floating-point precision allocation is a necessity. Also, the increasing platform heterogeneity is likely to increase the proclivity for program divergence — a definite impediment to achieving execution reproducibility. In this paper, we offer the first in-depth study of divergence. Our experimental results suggest that our new heuristics, namely ABS and GRT, are capable of handling many practical examples with well over 1000 inputs by quickly guiding input generation to locate divergence. For our future work, we plan to study heterogeneity-induced divergence scenarios.

References

1. T. Bao and X. Zhang. On-the-fly detection of instability problems in floating-point program execution. In *OOPSLA*, pages 817–832, 2013.
2. T. Bao, Y. Zheng, and X. Zhang. White box sampling in uncertain data processing enabled by program analysis. In *OOPSLA*, pages 897–914, 2012.
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. M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 3rd ed. edition, 2008.
5. A. Bykat. Convex hull of a finite set of points in two dimensions. *Information Processing Letters*, pages 296–298, 1978.
6. CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.
7. S. Chaudhuri, A. Farzan, and Z. Kincaid. Consistency analysis of decision-making programs. In *POPL*, pages 555–567, 2014.

8. W.-F. Chiang, G. Gopalakrishnan, and Z. Rakamarić. Unsafe floating-point to unsigned integer casting check for GPU programs. In *NSV*, 2015.
9. W.-F. Chiang, G. Gopalakrishnan, Z. Rakamarić, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *PPOPP*, pages 43–52, 2014.
10. E. Darulova and V. Kuncak. Sound compilation of reals. In *POPL*, pages 235–248, 2014.
11. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
12. C. Ericson. *Real-time collision detection*, volume 14. Elsevier, 2005.
13. R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 1962.
14. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23:5–48, 1991.
15. R. L. Graham. An efficient algorith for determining the convex hull of a finite planar set. *Information processing letters*, pages 132–133, 1972.
16. Y. Gu, T. Wahl, M. Bayati, and M. Leeser. Behavioral non-portability in scientific numeric computing. In *Euro-Par*, pages 558–569, 2015.
17. M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, pages 851–876, 2007.
18. L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry Theory Applications*, pages 61–78, 2008.
19. D. E. Knuth. *Axioms and hulls*. Lecture Notes in Computer Science, LNCS 606, 1992. ISBN 3-540-55611-7.
20. M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *ICS*, pages 369–378, 2013.
21. 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 *CGO*, pages 230–237, 2010.
22. Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. Preliminary experiences with the uintah framework on intel Xeon phi and stampede. In *XSEDE*, pages 48:1–48:8, 2013.
23. G. Paganelli and W. Ahrendt. Verifying (in-)stability in floating-point programs by increasing precision, using smt solving. In *SYNASC*, pages 209–216, 2013.
24. P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI*, pages 1–11, 2015.
25. 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*, pages 27:1–27:12, 2013.
26. E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, pages 53–64, 2014.
27. J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, pages 305–363, 1997.
28. A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM*, pages 532–550, 2015.
29. M. Taufer, O. Padron, P. Saponaro, and S. Patel. Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs. In *IPDPS*, pages 1–9, Apr. 2010.