DOI:
10.1145/2483760.2483791

Finding rare numerical stability errors in concurrent computations

Conference Paper · July 2013

DOI: 10.1145/2483760.2483791

CITATIONS
2

READS
46

3 authors:

Hana Chockler
IBM
62 PUBLICATIONS **797** CITATIONS
SEE PROFILE

Karine Even Mendoza
King's College London
9 PUBLICATIONS **11** CITATIONS
SEE PROFILE

Eran Yahav
Technion - Israel Institute of Technology
142 PUBLICATIONS **3,062** CITATIONS
SEE PROFILE

Finding Rare Numerical Stability Errors in Concurrent Computations

Hana Chockler
IBM Research, Haifa, Israel
hanac@il.ibm.com

Karine Even
Technion, Haifa, Israel
karinee@technion.ac.il

Eran Yahav
Technion, Haifa, Israel
yahave@technion.ac.il

ABSTRACT

A numerical algorithm is called stable if an error, in all possible executions of the algorithm, does not exceed a predefined bound. Introduction of concurrency to numerical algorithms results in a significant increase in the number of possible computations of the same result, due to different possible interleavings of the concurrent threads. This can lead to instability of previously stable algorithms, since rounding combined with the possibility of different interleavings of threads can result in a larger error than expected for some interleavings. Such errors can be very rare, since the particular combination of rounding can occur in only a very small percentage of these interleavings. In this paper, we suggest to adapt the cross-entropy method – a generic approach to rare event simulation and combinatorial optimization – to detection of rare numerical instability in concurrent programs. The cross-entropy method iteratively samples a small number of executions and adjusts the probability distribution of possible scheduling decisions to increase the probability of encountering an error in a subsequent iteration. We demonstrate the effectiveness of our approach on implementations of several numerical algorithms with concurrency and rounding by truncation of intermediate computations. We describe several abstraction algorithms on top of the implementation of the cross-entropy method and show that with abstraction, our algorithms successfully find rare errors in programs with hundreds of threads. In fact, some of our abstractions lead to a state space whose size does not depend on the number of threads at all. We compare our approach to several existing testing algorithms and argue that its performance is superior to other techniques.

1. INTRODUCTION

Numerical algorithms use numeric approximation for solving hard computational problems. These algorithms are used in all fields of engineering and the physical sciences for complex and critical calculations. The notion of *numerical stability* is an important criterion for numerical algorithms: an algorithm is called *stable* if an error, in all possible executions of the algorithm, does not exceed a predefined bound (which is regarded as an acceptable rounding error) [11]. Stability of a numerical algorithm is an important factor in its analysis, since an error exceeding the bound can lead to disastrous results: for example, the Patriot missile failure in 1991 occurred due to rounding errors [7], explosion of the Ariane 5 rocket in 1996 was a result of an overflow error [24], and sinking of the Sleipner A offshore platform was caused by a combination of a numerical error and a physical error [25].

Introduction of concurrency to numerical algorithms results in a significant increase in the number of possible computations of the same result, due to different possible schedules of the concurrent threads. This can lead to instability of previously stable algorithms, since rounding combined with the possibility of different interleaving of threads can result in a larger error than expected for some interleaving. Such errors can be very rare, since the particular combination of rounding can occur in only a very small percentage of interleaving, making the detection of such errors an especially challenging task.

In this paper, we suggest to adapt the *cross-entropy method* to detection of rare numerical instability in concurrent programs. The cross-entropy method is a generic approach to rare event simulation and combinatorial optimization [27]. It derives its name from the cross-entropy or the Kullback-Leibler distance, which is a fundamental concept of modern information theory [16]. Roughly speaking, the cross-entropy method is an iterative approach based on minimizing the cross-entropy (or the Kullback-Leibler distance) between the current probability distribution and an optimal probability distribution on a given probability space. An optimal probability distribution is a distribution that maximizes the probability of elements with the highest value of a predefined performance function (see Figure 1). In numerical algorithms, natural performance functions are derived from estimations of the error bounds, as described in [5]. We demonstrate our approach on implementations of several numerical algorithms using different methods of summation (see [10] for the description of summation methods in numerical algorithms) on floating point numbers and truncation of results. We also address the issue of scalability of the cross-entropy method by

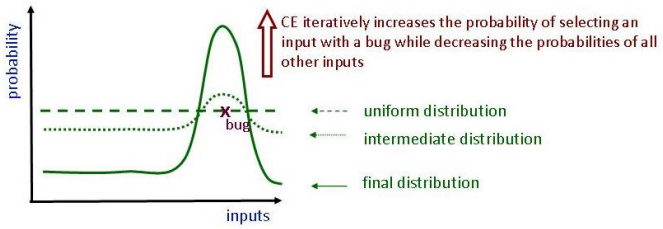

Figure 1: Changes in the probability distribution as a result of using the cross-entropy method

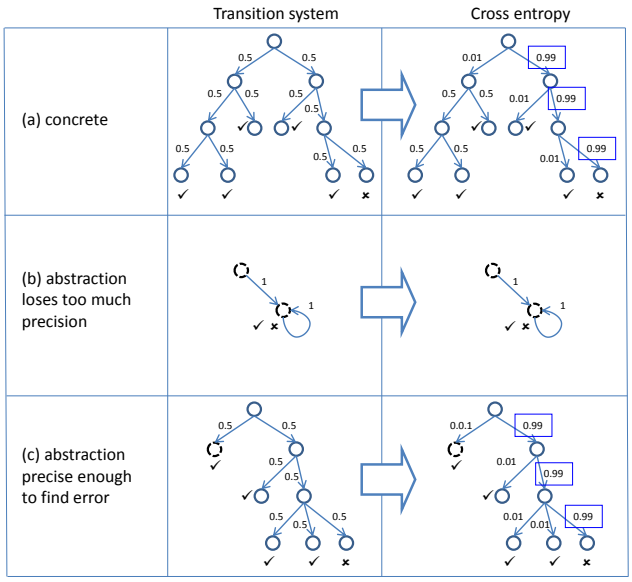

Figure 2: Cross-entropy with Abstraction.

introducing abstraction of the recorded executions. We show that with abstraction, our implementation successfully finds rare numerical instability in programs with up to 1000 threads. We compare our approach to several existing testing algorithms and show that our implementation can find errors that are not detected by other methods.

Main Contributions. The contributions of this paper are:

- We present a framework for cross-entropy based testing of concurrent numerical programs with a large number of threads. The framework can be used to detect rare numerical stability errors in concurrent computations. Our framework combines ideas from cross-entropy based testing, together with abstractions that make these techniques effective for programs with large number of threads.

- We instantiate the framework with: (i) a performance function for the common case of round-off errors due to a series of additions and subtractions of floating point numbers, and (ii) a number of abstractions that push the scalability limits of cross-entropy based testing, and make it applicable to programs with a large number of threads.

- We have implemented our approach in a tool ACE and applied it to test the numerical stability of several concurrent numerical algorithms from the literature. We show that with certain precision requirements, the concurrent algorithms are no longer stable, but that this instability cannot be detected by random testing. We show that ACE is effective in finding these numerical stability errors.

2. OVERVIEW

In this section we present an overview of our approach. We illustrate the concept of numerical instability on the motivational toy example of a concurrent computation of an approximate sum of floating point numbers and explain how the instability can be found using the cross-entropy method.

2.1 Motivating Example - Array Sum

Our toy example *ArraySum* is a concurrent computation of an approximate sum of floating point numbers. In this example, an array of 10 input numbers is given to the algorithm as an input, and the algorithm performs concurrent additions, where each addition is done by a different thread atomically (hence we have 10 threads running concurrently). A pseudocode of the toy example is shown in Figure 3, where $finalSum$ (the result of the algorithm), and $input\_decimal\_array$ (the input array) are both shared data variables. While there is actually no "bug" in this algorithm, it can

$$
\begin{array}{l}
Thread(i): \\
\quad local = input\_decimal\_array[i]; \\
\quad atomic\{ \\
\quad\quad finalSum += local; \\
\quad \}
\end{array}
$$

Figure 3: *Array Sum*

give different results on the same inputs depending on the order of operations and the rounding or truncation operations it performs. For example, consider the following input array with 10 floating point numbers of three digits (of course, our method is not limited to three-digits numbers; we are just using this here for clarity of presentation):

$$[0.25, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.75, 6.0],$$

and assume that the output is a floating point number of three digits as well (overflows are truncated). Then, summing the numbers in the order they are given results in

$$0.25+0.5+0.5+0.5+0.5+0.5+0.5+0.5+0.75+6.0 = 10.5,$$

which is correct. On the other hand, the order

$$6.0 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.75 + 0.25$$

loses 0.05 at the addition of 0.75 to the intermediate result (the accurate result is 10.25, and the truncated result is 10.2), and then loses 0.05 again at the last addition of 0.25, hence giving an inaccurate result 10.4.

In Section 5, we consider the *ArraySum* example with input arrays of up to 1000 numbers as well as other numerical algorithms performing additions and subtractions of floating point numbers. In these cases, a numerical error of the final result can be a result of the accumulation of several minor numerical errors in the intermediate additions. A single numerical error might be small enough to still fall within the bounds of acceptable approximation, and often these errors do not accumulate. We show that numerical instability of a concurrent algorithm can occur with exponentially small probability and hence is almost impossible to detect using existing testing approaches.

2.2 Our Approach

The type of error described in Section 2.1 results not from one erroneous computation step, but from a specific combination of legal operations. Thus these errors are particularly hard to find, especially when the orders resulting in an error are very rare (see Section 6 for the discussion on related work).

The cross-entropy method. We use the cross-entropy method to detect rare numerical stability errors in concurrent programs, as illustrated by the *ArraySum* example. The cross-entropy method is based on gradual improvement of a sample, and can be viewed as a guided search towards the order of operations that leads to an error.

In our setting, the probability space is the space of all possible executions viewed as a graph, with the set of nodes being the state space of a given program and edges corresponding to transitions between states in legal executions. The probability distribution is derived from probabilities defined on edges, and the performance function – the function that defines a direction of convergence of the cross-entropy method – is based on the bounds for an absolute error in the result as defined in [5] and is likely to have higher values on executions with higher probability of having an error due to rounding or truncation. Essentially, our performance function gives higher values to executions in which the intermediate sums are expected to have many digits, and it shows a good correlation with the occurrences of truncation errors in the final result. This is shown schematically in Figure 2 (a), where the initial transition system has weights allocated with a uniform distribution (left), and after the application of cross-entropy, the path leading to the buggy state (marked by $\times$) is given higher probability (right).

Cross-entropy with Abstraction. The cross-entropy method applied over the transition system of a concurrent program has an inherent scalability limitation, as the size of the transition system is exponential in the size of the program and in the number of threads. This severely limited scalability of past applications cross-entropy to the domain of concurrent programs [3]. In this paper, we use abstraction to construct an *abstract transition system* which can be significantly smaller than the original program's transition system (in fact, the original transition system need not even be finite), and use the abstract transition system as basis for cross-entropy based-testing.

Note that our use of abstraction is limited to the transition system that guides the concrete exploration. The actual executions of the program are always concrete, and whenever a bug is detected it is guaranteed to be a real bug of the program. However, the abstraction controls the granularity at which cross-entropy probabilities are maintained. An imprecise abstraction (as schematically shown in Figure 2 (b)), would hinder effective guidance. A precise-enough abstraction (schematically shown in Figure 2 (c)) is abstract enough to hide parts of the transition system, but keeps enough information for cross-entropy guidance to be effective.

Our implementation is based on the tool ConCEnter [3], where the cross-entropy method is used for testing multi-threaded programs. In our implementation, we add several layers of abstraction – of the number of threads and of data, significantly increasing the scalability of the tool while still keeping enough information to allow convergence. Essentially, abstraction reduces the size of each stored node in the execution and also the size of the stored graph of previously seen execution by keeping less information in each node. Our implementation successfully analyses programs with up to 1000 threads, and thus is more scalable than even the traditional random testing. Since absolute bounds for an error defined in [5] can serve as natural performance functions for the numerical stability algorithms, and, on the other hand, numerical instability can be very rare and hard to find using traditional methods, we foresee wide applications for our algorithms in concurrent versions of numerical computations.

3. PRELIMINARIES

3.1 Programs as transition systems

All definitions refer to a finite multi-threaded program P. For simplicity, we assume unique locations, that is, there is one-to-one correspondence between the program counter and a line in the code. For loops, the loop counter is considered a part of the program counter (and similarly for for functions), to avoid unwinding of loops and inlining of all function calls (see also [3] for a discussion on abstraction that avoids unwinding of loops). For readability, we omit P from notations.

Let $PID = \{1, \ldots, n\}$ be the set of *thread identifiers* in the program, where n stands for the number of threads. We assume that a thread identifier is unique and constant through the executions. The executions of P are monitored by keeping track of a subset of program locations, specified by the line numbers in the code of P (roughly speaking, these are locations that influence the scheduling or a result of the program's execution). We denote this subset by *label*.

A *state* of P is formally defined below as a combination of the program locations of each thread in P together with the values of data. The program locations are determined by the values of labels (from *label*) and, since we focus on numeric programs, we assume that data values are in $\mathbb{R}$. We denote by $Lvar$ the set of local variables and by $Svar$ the set of shared variables of P.

Definition 1 (Program state) *A* program state $\sigma \in \Sigma$ *is a tuple* $\langle PID, pc_\sigma, Ldata_\sigma, Sdata_\sigma \rangle$*, where PID is the set of thread identifiers as defined above,* $pc_\sigma : PID \to Label$ *defines the current location of each thread,* $Ldata_\sigma : PID \to (Lvar \to \mathbb{R})$ *defines the values of local variables, and* $Sdata_\sigma : Svar \to \mathbb{R}$ *defines the values of shared variables of* P *in the current state.*

Transitions describe possible execution steps from the current program state to a possible next state. There is a transition between two states σ and σ', if there exists an execution of P that visits σ' in the next step after visiting σ. In particular, this means that one thread changes its location from σ to σ' and the other threads stay in the same locations, the values of shared variables and local variables to the thread that executed a statement may change as a result of executing this statement, and the values of local variables of other threads do not change.

Definition 2 (Program transition) *A* program transition $t \in \Sigma \times \Sigma$ *is a pair* $\langle \sigma, \sigma' \rangle$*, where* $\sigma = \langle PID, pc_\sigma, Ldata_\sigma, Sdata_\sigma \rangle \in \Sigma$ *is a source state,* $\sigma' = \langle PID, pc_{\sigma'}, Ldata_{\sigma'}, Sdata_{\sigma'} \rangle \in \Sigma$ *is a destination state, and there exists* $i \in \{1, \ldots, n\}$ *such that:*
(1) $pc_\sigma|_{PID \setminus \{i\}} = pc_{\sigma'}|_{PID \setminus \{i\}}$ *(that is, the states* σ *and* σ' *are identical except for the thread* i*);*
(2) *let* st_i *be the statement in the current location of the thread* i *in* σ*; then,* $\sigma' \in [\![st_i]\!](\sigma)$*, where* $[\![st_i]\!](\sigma)$ *is the set of possible states of the program when executing the statement* st_i *in state* σ*;*
(3) *the only changes of the values of data between* σ *and* σ' *are in shared data and thread* i*'s local data variables.*

We denote the set of all possible transitions by T. An *execution* of P is a sequence $\sigma_0, \sigma_1, \ldots \sigma_j$ such that for all $0 \leq i < j$, we have $\langle \sigma_i, \sigma_{i+1} \rangle \in T$.

We are now ready to define a program as a transition system.

Definition 3 (Transition system) *For a program* P*, we define a* transition system representing P *as a tuple* $TS = \langle \Sigma, T, \Sigma_0 \rangle$*, where* Σ *is the set of all states of* P*,* T *is the set of all transitions of* P*, and* $\Sigma_0 \subseteq \Sigma$ *is the non-empty set of initial states of* P*.*

A *probability distribution* on transition systems is defined as follows.

Definition 4 (Probability distribution) *For a transition system* $TS = \langle \Sigma, T, \Sigma_0 \rangle$*, a probability distribution* $PF : T \to \mathbb{R}$ *on the transitions of* TS *is a function, such that* $0 \leq PF(t) \leq 1$ *for all* $t \in T$*, and for every state* $\sigma \in \Sigma$*, we have* $\sum_{\langle \sigma, \sigma' \rangle \in T} PF(\langle \sigma, \sigma' \rangle) = 1$*. In other words, the probability is always between* 0 *and* 1*, and for each state, the sum of probabilities of its outgoing transitions is* 1*.*

The transition system TS of a program P can be viewed as a graph $G = \langle V, E \rangle$, with the set of states Σ being the set of nodes V, and the set of transitions T being the set of edges E of G. The probability distribution PF over TS then becomes the probability distribution over the edges of G. In the next section, we give a brief overview of the cross-entropy method viewed as an optimization problem on graphs, and in Section 4 we adapt this framework to finding rare numerical stability errors in concurrent computations.

3.2 Cross-Entropy Method Over Graphs

The *Cross-Entropy (CE) method* was developed in order to efficiently estimate probabilities of rare events, where an event e is considered a rare event if its probability is very small, say, smaller than 10^{-5}. In this method, we are given a very large probability space and a function S from this space to $\mathbb{R}^+$, and we say that e occurs on an input X (from the probability space) if $S(X) > \gamma$ for some predefined value $\gamma \in \mathbb{R}^+$. Since the space is very large, it is infeasible to search it exhaustively, therefore the estimation of the probability l of e is made by sampling. A straightforward way to estimate l is to draw a random sample according to the given probability distribution f on inputs, and then estimate l by examining the sample. The problem is, that when e is a rare event, the sample might have to be very large in order to estimate l accurately. A better way is to draw the sample according to some other probability distribution g that raises the probability of e. The ideal probability distribution here would be g_l, which gives the probability 0 to inputs that do not contain e. The CE method attempts to approximate g_l. The distance between two distributions. used in this approximation is the *Kullback-Leibler "distance"* (also called *cross-entropy*). The Kullback-Leibler "distance" between g and h defined as:

$$\mathcal{D}(g,h) = E_g \ln \frac{g(X)}{h(X)} = \int g(x) \ln g(x) dx - \int g(x) \ln h(x) dx.$$

Note that this is not a distance in the formal sense, since in general it is not symmetric. Since g_l is unknown, the approximation is done iteratively, where in iteration i we draw a random sample according to the current probability distribution f_i and compute the (approximate) cross-entropy between the f_i and g_l based on this sample. Then we construct f_{i+1} by updating f_i based on the cross-entropy result. The reader is referred to the book on cross-entropy for the complete description of the method[27].

Our method of finding rare numerical stability errors in concurrent programs is based on the application of the CE method to *graph optimization* problems. In these problems, we are given a (possibly weighted) graph $G = \langle V, E \rangle$, and the probability space is the set of paths in G represented by the sets of traversed vertices. A *performance function* S is defined over the paths of the graph such that it is smooth enough to ensure convergence and reaches its maximum on inputs we are searching for. This setting matches, for example, the definitions of the traveling salesman problem and the Hamiltonian path problem in the context of CE method, where the performance function reaches its maximum on the solution for the traveling salesman problem and on the Hamiltonian path in the graph, respectively.

This is also the setting which we use in our approach. Our goal is to find executions of the concurrent program which result in a wrong output of a numeric computation. Section 3.1 formally defines concurrent programs as graphs, with a probability distribution on edges corresponding to scheduling decisions. Paths in this graph correspond to executions of the program. We discuss the choice of the performance function S in Section 4.1, where we show that for numerical stability problems there are natural pre-defined performance functions based on the bounds of the absolute error in computations.

The CE method for graph optimization problems is shown more formal terms in Figure 4, where f_i is the *probability distribution* on the edges of G in the i-th iteration, S is a *performance function*, $q \ll 1$ is a parameter indicating the size of the best samples subset (of size is $\lceil qN \rceil$), and N is the number of elements in the sample per iteration. The elements $\pi_1, \ldots, \pi_N$, are paths on the graph,

Input: Graph $G = \langle V, E \rangle$, probability distribution f_0, performance function S, and $q \ll 1$.

On each iteration i do:
Generate a sample $\Pi_i = \{\pi_1, \ldots, \pi_N\}$ of size N according to f_i
Sort Π_i according to S
Let $Q(\Pi_i) = \{\pi_{\lfloor (1-q)N \rfloor}, \pi_{\lfloor (1-q)N+1 \rfloor}, \ldots, X_N\}$
Update f_{i+1}

Figure 4: The pseudo-code of the CE-Method for graph optimization problems

which are, in our case, executions of the program according to the current probability distribution, in the order of increasing value of S. $Q(\Pi_i)$ is the best q-quantile of the sample on i-th iteration, and it is used for generating the updated probability distribution for the next iteration. The probability distribution update is based on the formula

$$f_{i+1}(e) = \frac{|Q(\Pi_i)(e)|}{|Q(\Pi_i)(v)|}, \tag{1}$$

where $e = \langle v, w \rangle \in E$ is an outgoing edge from v, $Q(\Pi_i)(v)$ is the set of paths in the $Q(\Pi_i)$ that visit v, and $Q(\Pi_i)(e)$ is the set of paths in $Q(\Pi_i)$ that use e. Intuitively, the edge e "competes" with other outgoing edges from the same vertex, and the updated probability for e depends on the fraction of paths in $Q(\Pi_i)$ that use e out of the paths in $Q(\Pi_i)$ that visit v. We continue in the next iteration with the updated probability distribution f_{i+1}. The procedure terminates when a sample has a relative standard deviation below a predefined threshold parameter (usually between 1% and 5%).

Remark 1 (Smoothed updating) *In optimization problems involving discrete random variables, such as graph optimization problems, the following equation is used in updating the probability function instead of Equation 3.2:*

$$f'_{i+1}(e) = \alpha f_{i+1}(e) + (1-\alpha) f_i(e), \tag{2}$$

where $0 < \alpha \leq 1$ *is the* smoothing parameter. *Clearly, for* $\alpha = 1$ *we have the original updating equation. Usually, a value of* α *between* 0.9 *and* 0.95 *is used. The main reason why the smoothed updating performs better than non-smoothed is that it prevents losing a good element of the sample forever (if one of its edges is assigned* 0 *in one of the iterations).*

4. CROSS-ENTROPY BASED TESTING FOR NUMERICAL STABILITY PROBLEMS

In this section, we show how to apply CE-Based testing to parallel numerical programs with a large number of threads. Testing of large concurrent programs is often infeasible due to space or time limitations. Exhaustive techniques face the "the state explosion problem", whereas non-exhaustive techniques can miss bugs

altogether. The CE-method searches the state space of a program dynamically, and yet stores all paths of the current iteration, together with the information on the (currently known) program's states as the performance function's entries.

The size of a concrete program state directly depends on the number of threads, since each thread's location and data is stored in the state. This makes storing the concrete information about executions infeasible for programs with many threads. At the same time, since the reachable state space of a concurrent program is exponential in the number of threads, small samples cannot obtain enough information to guide convergence of the cross-entropy method. To overcome these problems, we introduce *abstraction* to reduce the size of stored states and to "condence" the state space without losing information necessary for convergence.

We start by presenting a performance function for numerical algorithms using addition and subtraction (Section 4.1), and then in Section 4.2, we show how to apply abstraction to CE-based testing.

4.1 Performance Function for Floating Point Summation and Subtraction Algorithms

In numerical programs, a program is considered *stable* if all its executions on the same input give the same result up to the allowed error margins, due to truncation or rounding. A program is *numerically unstable* if there exists an execution that outputs a result that deviates from the result of the sequential version by more than the allowed error margin. In other words, the sequential version is considered correct, and the correctness of the concurrent version is assessed with respect to the sequential one. Errors in numerical programs occur when the number of digits of the output is not sufficient for the result of the calculation, and thus some digits are truncated (and the result can be either truncated or rounded to the next value). In summation, these errors occur when the result is too big fit in the allocated number of digits of the output. In subtraction, the error is caused by *subtractive cancellation* (or *loss of significance*), which happens when an operation on two numbers increases relative error substantially more than it increases absolute error, for example in subtracting two nearly equal numbers.

The naive performance function for concurrent numeric programs ranks the executions by the distance of their output from the output of the sequential version. This function, however, is not suitable for the cross-entropy method, because it requires the correct value to be known in advance.

Instead, we use a performance function that corresponds to the likelihood of having a rounding, truncation, or cancellation error in each operation. For addition, the larger the distance between the two operands is, the more likely it is that the addition operation will result in a truncation or rounding error; in subtraction, the situation is reversed: subtracting numbers that are close together is more likely to produce a subtractive cancellation error [10].

The performance function of a path (an execution of the program) is the *sum* of the values of the performance function of all operations on this path, assuming the operations are of the same type, as defined in [5]. For programs that perform both addition and subtraction, we define the performance function to be a pair of the value of the performance function on the addition part of the execution and the value of the performance function on the subtraction part of the same execution. More formally, for a prefix π_i of a path π on the graph G corresponding to the program P, the value of the performance function for addition S^+ on π_i is

$$S^+(\pi_i) = \sum_{i=0}^{y} \left(\begin{array}{l} |exp\,(r_i) - exp\,(e_i)| + digits\,(r_i) + \\ +digits\,(e_i) + digits\,(max\,(r_i, e_i)) \end{array} \right), \tag{3}$$

where

- e_i is the current added element (that is, e_i participates in the last operation in π_i;
- $r_i = \sum_{j=0}^{i-1} e_j$ is the intermediate sum of all elements in π_i except e_i;
- $exp(x)$ is the exponent in the standard representation of the floating point x;
- $digits(x)$ is the number of digits in the standard representation of x.

It is easy to see that performance function for additions of floating point numbers gives larger values to operations where the operands are far from each other (and thus potentially resulting in numbers which will be rounded or truncated).

The performance function S^- for subtraction of floating point numbers gives larger values to executions with subtractions of close numbers. The formula is similar to the formula for S^+, with maximum reached on close numbers:

$$S^-(\pi_i) = \sum_{i=0}^{y} \left(\begin{array}{l} |exp(|r_i - e_i|) - \\ -exp(\max(|r_i|, |e_i|))| + \\ +digits(|r_i - e_i|) + \\ +digits(\max(|r_i|, |e_i|)) + \\ +digits(\max(|r_i - e_i|, max(|r_i|, |e_i|))) \end{array} \right), \tag{4}$$

where e_i, r_i, and $digits$ are defined as for S^+. It is easy to see that the value S^- directly depends on the difference between the values of its operands and the difference between these values; that is, the closer the operands to each other, the larger the value of S^-. Since in subtraction, errors occur when the result is small compared to the operands, this function has a heuristic correspondence to the likelihood of an error in a series of subtractions.

For algorithms with both addition and subtraction, we define the performance function S^{+-} as an ordered pair of the addition part and of the subtraction part separately, that is, $S^{+-} = \langle S^+, S^- \rangle$. Then, for paths π_1 and π_2, if $S^+(\pi_1) > S^+(\pi_2)$ and $S^-(\pi_1) < S^-(\pi_2)$, we say that $S^{+-}(\pi_1) > S^{+-}(\pi_2)$ in the first k iterations, and $S^{+-}(\pi_1) < S^{+-}(\pi_2)$ in the next k iterations of the algorithm. Intuitively, we use the addition part to guide convergence for several iterations, then switch to using the subtraction part. The parameter k is, of course, tunable: if the algorithm does not converge within the first two k iterations, we can either increase k or continue alternating between the addition and subtraction part. In our benchmarks, we used 2 and 3 for the value of k.

4.2 CE-Based Testing with Abstraction

As we mention in the beginning of this section, the size of each concrete state directly depends on the number of threads, as it stored the information about each thread's location in the code and its data. This results in very large states for programs with many concurrent threads. Moreover, theoretically, the size of the state space of a concurrent program is exponential in the number of threads. While we never construct the whole state space of a program, its size slows down the convergence of the cross-entropy method, since many executions need to be examined in order to find the direction of improvement of the performance function. We

aim to avoid the dependency on the number of threads altogether by replacing the concrete executions with their abstract counterparts.

We present a simple family of abstractions that allows us to use cross-entropy based testing on concurrent numerical programs with a large number of threads. Abstraction is used to construct an abstract transition system, which results in smaller recorded executions and a smaller overall state space. The abstraction allows us to reduce the recorded information in two dimensions:

- *reduced number of entries*: abstraction of data, thread identities, and counters; and
- *reduced size of each entry*: abstraction of data and the concrete location of each thread.

As we describe more formally below, abstraction of data either abstracts away all data completely or replaces the full data with $\log(data)$. This is possible, of course, since all data in numeric programs is numeric, hence the $\log$ is well defined. Abstraction of concrete thread locations is done by replacing the vector of thread locations for each thread by the vector of program locations with the number, for each program location, of the number of threads currently in this location. This can be further abstracted by only distinguishing between the cases of 0 threads, 1 thread, or more than one thread for each program location. Clearly, this abstraction reduces the size of the state space when there are many threads and the code of the program is relatively small. The transitions in abstract transition systems are computed from the concrete transition system using *existential abstraction*: there is a transition between two abstract states if there exists a pair of concrete states corresponding to this abstract states with a transition between them.

We note that we still execute the conrete program, and abstraction is used only to reduce the recorded information. Therefore, bugs found using our method are *real bugs in the program* (and not spurious bugs resulting from overly coarse abstraction). In order to obtain a concrete execution exhibiting the bug, we only need to execute the last iteration of the algorithm once, as this is the iteration which gives the highest probabilities to executions with interleavings leading to an error.

Abstraction of thread identifiers. In this type of abstraction, we do not assign a unique identifier to each thread, but rather identify sets of threads with common characteristics (such as program location, values of data, etc.)

Thread counter abstraction. In this type of abstraction, instead of keeping a vector of threads and their program locations, we keep a vector of program locations, where, for each location, we store an abstraction of the actual number of threads with the same abstract thread identifier and the same (possibly abstract) data in this program location. In our test cases we used two abstractions: $0-1$ (either there are no threads at this location or there is at least one thread of this type) and $0-1-2$ (no threads at this location, one thread, or more than one thread). Clearly, this abstraction can be refined as required, but it is sufficient in our test cases.

Data abstraction. Instead of storing the full data, we can store some abstract representation of it, thus minimizing the size of each state. We use $\log$ data abstraction (where we store the $\log$ of data instead of its actual value) and full data abstraction, where we do not record the data values at all.

Abstraction of the program location. In large programs or programs with large loops (or functions), there can be many different program locations, many of which refer to locations that are either similar to each other (for example, the same place in the loop in subsequent iterations) or to non-interesting locations in the code (where different interleavings do not affect the final result). In the numerical algorithms we study in this paper, the is relatively small, and the complexity results from the large size of the input and the large number of threads, hence we do not abstract the program locations. In larger programs, the abstraction of the program location can be done by replacing the program counter with its value *modulo* some smaller number (in particular, this is a good abstraction for loops - see [3]) and by giving equivalent program locations the same value of the counter.

An abstraction of a program P defines an *transition system* TS_{abs} in a natural way and an abstract graph $G_{abs} = \langle V_{abs}, E_{abs} \rangle$ similarly to the definitions of the concrete transition system and the corresponding graph in Section 3.1. The values of the probability distribution of each abstract transition are computed using the same formula as in concrete transition systems. Clearly, abstract transitions do not correspond to threads' interleavings directly.

These abstractions can also be used in conjunction with each other – for example, we can abstract the thread counters and data at the same time. The obvious advantage of using abstractions is the reduction in the size of each state and hence in the overall space required for storing the recorded executions. A more subtle advantage is that abstraction "condences" the overall state space of the program, making it easier to converge to the maximum of the performance function. On the other hand, a too coarse abstraction might map two executions – one with a bug and one without a bug – to one abstract execution, thus interfering with convergence. As we show in Section 5, sometimes a finer abstraction ends up to be more efficient than a coarser one.

5. EVALUATION

In this section, we describe some details of our prototype implementation, and evaluate our implementation on a number of concurrent numerical programs with a large number of threads.

5.1 Implementation

We have implemented our method in a tool named ACE that is based on an existing implementation of the CE method [3, 4].

Data Abstraction.: ACE includes an implementation of two possible data abstractions, a full data abstraction, and a log data abstraction (by taking the log value of data). Other abstractions can be added easily.

Threads' Identification. In a concrete transition system of a program with at most n threads, with a set of thread identifiers $PID = \{1, \ldots, n\}$, a thread has the same thread identifier in all executions. A thread can be recognized by its type, spawn location or first-seen location, and fields attributes. With regards to Java programs, we use the thread's class, start or run location and local data; in our implementation, we set the *thread identity* (denoted as *thread-id*) to be a tuple of the thread's type, spawn or first-seen location, and local data. In the implementation of the concrete transition system, we use the same mapping $ID \mapsto PID$ in all executions to assure that a thread always has the same thread identifier; *ID* is a set of all *thread-id*'s of the program under test. Several instances of the same thread are distinguished by the number of instance, that is, we refer to i-th instance of the same *thread-id* as the same thread for all executions.

In abstract transition systems, thread identification is done by a tuple of thread type, spawn location, first-seen location and local data under some abstraction. For our benchmarks, we use abstract thread identifiers under a full data abstraction; that is, all concrete threads of the same type that have the same spawn location or first-seen location have the same abstract thread id, regardless of their

order of creation.

The values of the probability distribution PF are stored as a table, which maps each state of the program's transition system to its probability vector. In a concurrent transition system of a program with at most n threads, the probability vector of a concrete state $\sigma \in \Sigma$ is an n-dimensional vector, consisting of the probabilities of outgoing transitions of σ (where each outgoing transition represents a thread). The i-th element of this vector is greater than 0 if the i-th thread is enabled in σ and is 0 otherwise. The probability distribution table is updated at the end of each iteration. To improve the running time of our algorithm and reduce the amount of stored data, we only maintain states with non-uniform distribution over transitions (where the uniform distribution is the initial distribution over transitions). The comparison of the current state to the stored states during the execution is done by comparing the hash function values of states. The update of the probability distribution table, done at the end of each iteration off-line, compares the full states. In our experiments, we encountered only very few collisions of the hash function values on the majority of the states, and these collisions occurred only at the first iteration of the tool (since after that, each state has a different vector of probabilities of its outgoing transitions).

5.2 Benchmarks

Higham [10] identifies five different *summation methods* for computing the sum of an array of floating point numbers, four of which have concurrent algorithms as well: *recursive summation*, *pairwise summation*, $S + /S-$ *method*, and *compensated summation* (also known as *Kahan summation method*).

To evaluate our approach, we apply ACE to the following variants of the summation methods:

- *ArraySum* computes the sum of an array of floating point non-negative numbers using the recursive summation method;
- *ArrayDiff* computes the sum of an array of floating point numbers with mixed signs using the recursive summation method;
- *Tree Schema ArraySum* and *PrefixSum* are two algorithms computing the sum of an array of non-negative floating point numbers using the pairwise summation method;
- *SplusSminus* computes the sum of an array of floating point numbers with mixed signs using $S+/S-$ summation method;
- *Kahan* computes the sum of an array of non-negative floating point numbers using the using compensated summation method.

We elaborate more on each of these algorithms later in this section.

5.3 Methodology

We evaluate ACE by checking its ability to detect rare numerical stability bugs. Our benchmarks use decimal numbers with a certain number of digits in finite precision floating-point format: a Java float data type, a Java double data type, or a custom size decimal number (for instance, by using Java BigDecimal class). The input to all algorithms is an array of n-digits numbers drawn from a (given) discrete distribution.

The summation algorithms we use as benchmarks have a certain error resulting from rounding, truncation, and cancellation. We can define an error margin under which the result of the algorithm is still acceptable. Given an error margin, our goal is to check whether there are any executions where the error is more than the acceptable margin.

5.3.1 Description of benchmarks

Recursive summation method is defined as a summation of a set of real numbers in any order. The sequential algorithm sums a given input array of n numbers $\{x_i\}$ $(1 \leq i \leq n)$ in a given order (see Figure 5.3.1). The concurrent version of this algorithm

$finalSum = 0;$
$for\ all\ i \in \{1 \ldots n\}\ do:$
$\{$
  $finalSum+ = input\_decimal\_array[i];$
$\}$

Figure 5: Recursive summation method, sequential version.

is presented in Figure 3 (our toy example in Section 2).

We consider two variants of recursive summation: *ArraySum* over input arrays of non-negative numbers, and *ArrayDiff* over input arrays of positive and negative numbers. An input array is a set of floating point numbers up to a given precision (number of digits). All floating point operations are performed atomically, and thus, different results can result only from different thread schedules. We search for numerical instability of the final result of *ArraySum* and *ArrayDiff* by using the performance functions for floating points numbers described in subsection 4.1, where in *ArrayDiff*, the performance function is a tuple $\langle S+, S^- \rangle$. There are no subtractions operation *Array Sum*, as it sums only non-negative numbers, hence its performance function is S^+.

The pairwise summation method, also known as the cascade summation method, sums the set of given numbers in pairs (cf. [10, 1, 19]); the basic version of pairwise summation is an algorithm that sums an array of n numbers $\{x_i\}$ $(1 \leq i \leq n)$ into a new array of y_i $(1 \leq i \leq [\frac{n}{2}])$ recursively, as shown in Figure 6. The concurrent version of this algorithm has $[log_2 n]$ stages [12],

Input:
$decimal\ numbers\ sequence\ \{array_i\}, i \in [1, n]$

Algorithm:
$int\ n =$ size of $array;$
$finalSum = call\ PairwiseSum(array, n);$

$function\ decimal\ PairwiseSum(sequence\ of\ decimals\ :$
  $x, int : n)$
$\{$
  $if\ (n == 1)$
    $return\ x_1;$
  $n' = [\frac{n}{2}];$
  $Create\ initialized\ sequence\ size\ n' \rightarrow y;$
  $for\ all\ i \in \{1 \ldots n'\}\ do:$
    $y_i = x_{2i} + x_{2i-1};$
  $if\ (n\ is\ odd)$
    $y_{n'} = x_n;$
  $return\ call\ PairwiseSum(y, n');$
$\}$

Figure 6: Pairwise summation method, recursive version.

calculating the items of $\{y_i\}$ in parallel during each stage. It is based on the basic algorithm of Figure 6, when each call to *PairwiseSum* function is done by a different thread, and not in a loop. In our implementation of this algorithm, there are no barriers between stages, that is, a thread can add an item of a sequence to

the next sequence, once it finishes calculating it, without waiting for the creation of the whole sequence. For the pairwise summation method, we present two benchmarks: *Tree Schema ArraySum*, which is the concurrent version of the basic algorithm, and *Prefix-Sum* that returns all the partial sums along the way in addition to the final result. We explain these benchmarks in more details below.

Tree Schema Array Sum creates n threads, where each thread executes the code shown in Figure 7. Each thread atomically up-

Input:
$decimal\ numbers\ sequence\ \{x_i\}, i \in [1,n]$

Algorithm:
$Create\ initialized\ sequence\ size\ n \rightarrow z;$
$Copy\ sequence\ \{x_i\}\ into\ new\ sequence\ \{y_i\};$

$each\ thread(i)\ i \in \{2 \dots n\}\ performs:$
{
  $int\ l = max_{\{j \in [1, log_2(n)] \wedge ((i-1) mod (2^j) == 0)\}}(j);$
  $while\ (z_i < l);$
  $atomically\ perform: \{$
    $y_{i-2^l} += y_i;$
    $++ z_{i-2^l};$
  }
}

$thread(1)\ performs:$
{
  $int\ l = log_2(n);$
  $while\ (z_0 < l);$
  $output: y_1;$
}

Figure 7: Pairwise summation method, concurrent version.

dates an element from the sequence $\{y_i\}$. There is no barrier that forces a unique order of summation the elements, as shown in the example in Figure 8. The value of each element of the sequence $\{y_i\}$ is affected by the order of summation of the elements.

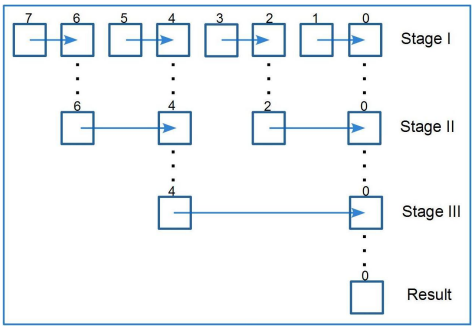

Figure 8: Example of Concurrent Summation of Eight Elements, Pairwise Summation Method. In this example, the order of adding cells 1,2,4 to cell 0, can be different in each execution, depends on the scheduling, and so can result in different outcomes.

Prefix Sum computes, for an input array of n numbers $\{x_i\}$, a sequence of prefixes of the sum of all elements: $\{y_i\}$, $y_i = \sum_0^i x_i$ $(0 \leq i < n)$. The parallel version of the algorithm [17] can be informally described as follows:

- Calculate $log(n)$ temporary sequences, $\{z_i^l\}$ $(0 \leq l \leq log(n))$, where $z_i^l = z_i^{l-1} + z_{i-2^{l-1}}^{l-1}$ if $i+1$ divided in 2^l, otherwise the value stays the same (i.e., $z_i^l = z_i^{l-1}$). The first sequence (where $l = 0$) is the input sequence.
- Calculate another temporary set of sequences, $\{w_i^l\}$, where $w_{i+2^{l-1}}^{l-1} = w_{i+2^{l-1}}^l + w_i^l$ if $i+1$ divided in 2^l, otherwise the value stays the same, that is, $w_i^{log(n)} = z_i^{log(n)}$.
- The output sequence is $\{w_i^0\}$.

All elements of the sequence are computed concurrently. A sequence depends on the previous sequences, thus extracting elements of different sequences in parallel is not possible. As a result, all addition operations, which are related to an element on the output sequence, are done in a fixed order;and so, all executions output the same sequence.

We modified the algorithm to allow extracting elements of different w^l sequences if all the elements that are required by an element we wish to extract are available. That is, we removed the barrier that forces finishing the computation of a w^l sequence before starting the computation of the next sequence. The w^l sequence is set to be a single sequence, $\{w_i\}$, which is the output sequence, where $w_i = \sum_{j=0}^{\lfloor log(i+1) \rfloor} a_{i,j}$, and $a_{i,j} = z_{i-(2^j-1)}^j$ if $((i+1) \wedge (2^j)) = (2^j)$ else $a_{i,j} = 0$. The z^l sequences calculation stays the same. Extracting an element of w sequence can be done in many different orders of addition operations, hence the computation can accumulate the rounding or truncation bugs, resulting in an inaccurate output even when all operations are done atomically. For simplicity, we search for an error only in the computation of the total sum of the input sequence, that is, y_{n-2}.

$S+/S-$ **summation method** was suggested in [18] as a method to overcome inaccurate results due to rounding errors in the solution of an optimization problem. This method suggests to sum all numbers of the same sign together into two different intermediate results S_+ and S_-, and compute the output $S_+ + S_-$ as the last operation. As always, all additions to S_+ or to S_- are done atomically. The result of an execution can be inaccurate due to rounding errors in the intermediate additions or due to a cancellation error when computing $S_+ + S_-$. Since there is a single instruction with a possible cancellation error and many with possible rounding errors (due to truncation of the intermediate result values in S_+ or S_-), and since the summation of S_+ and S_- is not done concurrently, we search for a numerical instability of the calculation of S_+ and S_-, not taking into account the effect of these errors on the final result. Since we ignore the effect of the last operation, there is only partial correlation of the performance function with the existence of an error in the execution of $S+/S-$. We avoid missing the actual error by reducing the value of the smoothing parameter compared to other implementations, thus slowing the convergence process.

Compensated summation method (or Kahan summation) was first described by Kahan [15] for floating point numbers and by Møller [21] for chopped floating point arithmetics. Roughly speaking, this method accumulates an estimation of the error on each arithmetic operation and adds it to the final result (see pseudocode in Figure 9). In the concurrent version of the algorithm in Figure 9, we set $finalSum, error$ and the input array $input\_decimal\_array$ to be shared data variables. All updates of shared data variables are done atomically. The threads of the concurrent algorithm perform the code of the inner *for* loop atomically; that is, the i-th thread adds the i-th element of the input array, including the last error estimation. The error here is the same as in $ArraySum$ code examples,

$finalSum = 0;$
$error = 0;$
$for\ all\ i \in \{1 \dots n\}\ do:$
$\{$
$\quad item\_with\_error =$
$\qquad input\_decimal\_array[i] + error;$
$\quad error = item\_with\_error - ((finalsum +$
$\qquad item\_with\_error) - finalsum);$
$\quad finalSum += item\_with\_error;$
$\}$

Figure 9: *Møller's Serial Step-wise Version [20].*

when the order of summation the elements affects the final result. We search for an instability in the code as we do in $ArraySum$ code example, ignoring the current error element. In order to estimate the effect of such a correction we use a large register/more precise numbers during the performance function estimation (that is, if the Kahan summation algorithm uses Java float numbers, we can use Java double numbers).

5.3.2 Evaluation setting

Our evaluation compares the results of random testing with the results of executing ACE in different configurations on our benchmarks as follows:

- concrete CE-based testing with concrete data (CONCRETE);
- four types of abstractions implemented in ACE: (i) counter $0-1-2$ abstraction with log abstraction of data (ABS1 LOG-3VAL); (ii) counter $0-1$ abstraction with log abstraction of data (ABS2 LOG-BINARY); (iii) counter $0-1-2$ abstraction with full abstraction of data (ABS3 FULL-3VAL); (iv) counter $0-1$ abstraction with full abstraction of data (ABS4 FULL-BINARY).

The experiments with ACE were conducted on a laptop, with Intel Core $i7-2640MCPU@2.80GHz4$, running Ubuntu 11.10 64 bit, 8GB memory. The random testing experiments were conducted on Linux 64 bit-arc, running Red Hat Enterprise Linux Client release 5.6 (Tikanga), memory 18479676 kB, eight CPUs Intel(R) Xeon(R), $E5440@2.83$GHz, which is a much more powerful machine than the one used for ACE experiments, making the results of ACE even more impressive in comparison with the random testing.

The code of ACE is available from the authors on request.

5.4 Experimental results

We present the results of evaluation of ACE and comparison between different configurations as described in Section 5.3 on the benchmarks described in Section 5.3.1. Our approach regards the input array as a part of the setting, that is, we are searching for an error for a given input array. We compare ACE with executing random testing in the same setting. One could ask whether randomizing the inputs can make the detection of the numerical stability error easier for the randomized testing. We show that the answer to this question is *no*: randomized testing is unable to detect the numerical stability errors in our benchmarks even in the setting where we draw many input arrays from natural distributions: uniform and binomial.

Evaluation of ACE. Table 1 shows the results of running ACE on our benchmarks. The table lists the number of executions needed for ACE to find the bug (the total number of executions and the number of executions with a bug), and the size of the performance function table (the number of nodes and their average size on the last iteration). The total number of executions of the code is the total number of iterations multiplied by the number of executions in one iteration, and the performance function table is the table of the first iteration in which the bug was found or in which ACE terminates (because the bug becomes common).

Table 2: Results of random testing. Random testing failed to find even a single bug despite often running for more than 24 hours.

Benchmark	Input size	Output Precision	Uniform Distr.		Binomial Dist.	
			Time	Number Sampled	Time	Number Sampled
Array Sum	100	5	13.25h	100	13h	600
		6	12.75h	100	13h	200
	150	5	15.5h	100	15.5h	400
		6	15.75h	100	16h	200
	250	5	18.5h	100	18.75h	200
		6	18.25h	100	18.5h	100
	500	5	22h	100	23h	300
		6	22.5h	100	23.5h	100
	1000	5	29.75h	100	30.75h	300
		6	29.75h	100	30.5h	100
Prefix Sum	128	5	33.5h	100	33.5h	100
		6	33.5h	100	34h	100
Tree Schema Sum	128	5	34h	100	34.25h	100
		6	34.5h	100	32h	100

We use OOM to denote cases where execution ran out of memory without detecting a bug, and T-O to denote a timeout. In most examples, when running random testing we used a generous timeout of 30 hours. If ACE did not find a bug after 500 executions using cross-entropy based testing, we consider that to be a timeout as well. We include the results for random testing on a given input array and testing with the uniform distribution over transitions. Each configuration of the ACE has two columns. The first column is a pair of numbers, where the first number is the total number of iterations until the bug is found, and the second is the number of buggy executions found. The second column shows the total number of nodes in the table, giving the average size of a single node in brackets.

The *ArraySum* implementation was executed with 100, 150, 250, 500, and 1000 threads, and *ArrayDiff* was executed with 100 and 150 threads; the size of the input array is the same as the number of threads in the benchmark. *TreeArraySum* and *PrefixSum* were executed with 128 threads on two different input arrays. *S+/S-* algorithm was executed with 100 and with 150 threads, and Kahan algorithm was executed with 150 threads.

Random testing. Table 2 shows the results of executing our benchmarks randomly on at least 100 input arrays drawn from uniform and from binomial distributions (as we consider these distributions to be the most natural ones). Each benchmark was executed at least 3000 times on each input array. All operations in our benchmarks were done with 7 digit precision. The required precision of the output is shown in terms of number of digits in the table (either 5 or 6). We sample the input values from the domain [0.01,111.]; The parameters of the uniform distribution are:

$$(n, a, b) = (11099, 111.0, 0.01),$$

and of the binomial distribution: $(n, p) = (11099, 0.5)$. The $ArraySum100$ algorithm was also executed with a different domain.

As can be seen from the table, several settings of the random testing were left to run for more than 24 hours. No errors were detected during any of these executions, supporting our initial hy-

Table 1: *Effectiveness of bug detection using different methods*: (1) Random testing, (2) execution of code under a uniform distribution over edges of the graph representing the code(Uniform over edges), and, (3) Concrete CrossEntropy method, (4) CrossEntropy with different abstractions.

Benchmark	Random	Uniform over edges	Cross-Entropy Based									
			Concrete		Abs1 Log-3Val		Abs2 Log-Binary		Abs3 Full-3Val		Abs4 Full-Binary	
ArraySum 10	31.67%	9.3%	10/3	89(10)	10/2	49(3)	10/1	54(3)	5/1	54(3)	5/2	42(2)
ArraySum 100	0.01%	T-O	T-O	44089(67)	240/1	5044(4)	300/1	4529(4)	300/1	5891(4)	T-O	5023(5)
ArraySum 150	0.05%	0.133%	320/1	33389(100)	260/1	10485(5)	390/1	8646(5)	140/1	6544(4)	130/1	4623(4)
ArraySum 250	0.11%	T-O	OOM	81365(165)	180/1	11886(4)	120/1	6971(4)	T-O	23494(5)	300/1	13363(5)
ArraySum 500	0.08%	0.0497%	OOM	-	T-O	54968(5)	195/1	18499(4)	T-O	51959(5)	180/1	18198(4)
ArraySum 1000	0.04%	0.106%	OOM	-	T-O	141601(5)	150/1	45107(5)	T-O	150044(5)	350/1	83789(5)
ArrayDiff 100	0.03%	0.1%	T-O	32183(67)	240/1	4400(4)	300/1	4594(4)	360/1	6488(1)	T-O	5940(5)
ArrayDiff 150	0.08%	0.233%	T-O	60940(100)	300/1	8585(5)	400/1	8324(5)	240/1	11652(5)	200/1	8024(5)
TreeArraySum 128	2.71%	0.1%	140/2	12688(128)	120/1	12694(26)	200/1	21286(25)	180/1	16603(24)	150/2	17964(27)
TreeArraySum 128	0.9%	1.7%	150/3	14767(113)	80/1	9663(23)	90/1	12405(24)	80/1	9593(24)	150/1	16266(24)
PrefixSum 128	0.0067%	2.0013%	T-O	17510(74)	120/4	26120(79)	150/4	36526(80)	120/6	25776(81)	150/5	24729(78)
PrefixSum 128	0.4%	3.776%	90/6	29479(221)	80/7	26026(77)	80/6	12596(76)	80/4	25916(81)	120/10	36521(80)
$S+/S-$ 100	0.04%	0.1%	T-O	39929(67)	T-O	13969(5)	100/1	4493(5)	150/1	5002(5)	300/1	9530(5)
$S+/S-$ 150	0.05%	T-O	T-O	67850(96)	300/1	16966(5)	T-O	23119(5)	T-O	28871(5)	300/1	20603(5)
Kahan 150	0.33%	0.0637%	480/1	61006(112)	120/1	4618(4)	120/1	3448(4)	120/1	4773(4)	120/1	3651(4)

pothesis that the rounding/truncation errors are too rare to be found by randomized testing, even if the inputs are chosen at random.

For each benchmark, the table lists: (i) input size; (ii) the required precision of the output, in terms of the number of digits; (iii) time and number of sampled arrays for discrete uniform distribution; (iv) time and number of sampled arrays for binomial distribution.

6. RELATED WORK

In general, concurrent programs present a challenge for all testing approaches dues to the sheer number of possible executions and difficulty to direct the scheduler to interesting (potentially buggy) executions. Exhaustive testing is clearly infeasible for even medium-size programs, since the number of possible threads-interleaving makes the state space of the program prohibitively large. *Random testing* in general tends to miss rare bugs, focusing instead of uniform coverage of the state space of the program [6, 29]. As our experiments show (see Section 5), repeated execution of a program either fully randomly or under natural probability distributions over possible transitions (we checked uniform and binomial distributions) does not lead to discovery of numerical instability, since the instability manifests itself on only a very small fraction of possible executions. Hence, in particular, the systematic approach for generating test cases for all feasible paths of a program using *concolic* techniques (see, for example, [8, 28]) would not scale to numerical algorithms in our examples.

The approach of creating uncommon or non-trivial scheduling of threads by interfering with scheduling decisions is used in several testing tools, which either enforce a specific scheduling [2, 14, 13, 26], or introduce a random interference (noise) in the scheduler with the goal of creating uncommon behaviors (see [29] or ConTest [6]). For the problems of numerical instability, these methods do not perform better than random testing, since the specific scheduling leading to a bug is not known in advance.

Methods in which heuristics or program analysis techniques are used to guide testing [2, 3, 13, 23, 28], usually perform more efficiently than random or exhaustive approaches. However, dictated by the nature of a program, one method can be more successful in finding a failure than others. Often, a successful method is good only for a specific problem, such as bugs that appear only when the system is heavily loaded [9](i.e., *stress testing*). In this case, not finding errors cannot guarantee the correctness of all of the possible behaviors of a given program.

Existing race-detection analysis based on *context bounding* essentially exploits the fact that most races occur even if we limit the number of context switches in the program, thus drastically reducing its state space (see, for example, CHESS [22]). This approach is not suited for finding instability in numerical algorithms because the existence of an error depends on the number of context switches – the rounding error is amplified with each context switch and reaches unacceptable values only on executions with a very high number of context switches.

Finally, perhaps the most relevant to this paper is the recent research on using cross-entropy for testing and replay of concurrent programs [3, 4]. The implementation described in [3] is the basis for our implementation of the cross-entropy based testing of numerical instability. The algorithm in [3] does not scale beyond programs with very few threads (the examples in the paper are of programs with two threads), and it searches for errors with no predefined performance functions, thus limiting this approach to problems with natural performance functions – the paper examines only the problem of buffer overflow, where the size of the buffer is the natural performance function. Using cross-entropy for replay, described in [4], is complementary to our algorithm and can be used to verify that the bug was fixed (by trying to replay a buggy execution).

7. CONCLUSIONS AND DISCUSSION

Theoretical error analysis of floating point summation methods shows that no method is numerically more stable than others and suggests guidelines for choosing a suitable summation method for each specific case [10]. For example, pairwise summation and compensated summation tend to work better for large arrays, and recursive summation with decreasing order of summation is recommended for sums with heavy cancellation. So it is of little surprise that we found numerical instabilities in all summation methods we checked. At the same time, as we demonstrated in the paper, existing testing methods are ill-suited for finding this type of errors, and hence they remain undetected even in mission-critical applications.

Most of the errors we found resulted from one specific scenario (which is also a reason why they were so hard to detect), which can be avoided by introducing several synchronization blocks, effectively blocking this particular scenario. In mission-critical applications, the addition of synchronization resulting from our error analysis might be a small price to pay for stability.

8. REFERENCES

[1] I. Babuska. Numerical stability in mathematical analysis. In *IFIP Congress (1)'68*, pages 11–23, 1968.

[2] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 167–178, New York, NY, USA, 2010. ACM.

[3] H. Chockler, E. Farchi, B. Godlin, and S. Novikov. Cross-entropy based testing. In *Proceedings of the Formal Methods in Computer Aided Design*, pages 101–108, Washington, DC, USA, 2007. IEEE Computer Society.

[4] H. Chockler, E. Farchi, B. Godlin, and S. Novikov. Cross-entropy-based replay of concurrent programs. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 201–215, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] G. Dahlquist and Å. Björck. *Numerical Methods*. Dover Books on Mathematics Series. Dover, 1974.

[6] O. Edelstein, E. Farchi, Y. Nir, G. Ratzaby, and S. Ur. Multithreaded java program test generation. 41(3):111–125, 2002.

[7] Patriot Missile failure. http://www.ima.umn.edu/ arnold/disasters/patriot.html, 1991.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[9] A. Hartman, A. Kirshin, and K. Nagin. A test execution environment running abstract tests for distributed software. In *SEA*, Proceeding (374) Software Engineering and Applications - 2002. Acta Press, 2002.

[10] N. J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput.*, 14(4):783–799, July 1993.

[11] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, second edition, 2002.

[12] R.W. Hockney and C.R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger, 1981.

[13] P. Joshi, M. Naik, C.-S. Park, and K. Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 675–681, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 110–120, New York, NY, USA, 2009. ACM.

[15] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, January 1965.

[16] S. Kullback and R.A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22:79–86, 1951.

[17] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980.

[18] L. S. Lasdon, J. Plummer, B. Buehler, and A D. Waren. Optimal design of efficient acoustic antenna arrays. *Math. Program.*, 39(2):131–155, November 1987.

[19] P. Linz. Accurate floating-point summation. *Commun. ACM*, 13(6):361–362, June 1970.

[20] O. Møller. Note on quasi double-precision. *BIT*.

[21] O. Møller. Quasi double-precision in floating point addition. *BIT*, 5(1):37–50, March 1965.

[22] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[23] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[24] Explosion of Ariane 5. http://www.ima.umn.edu/ arnold/disasters/ariane.html, 1996.

[25] Sinking of the Sleipner A offshore platform. http://www.ima.umn.edu/ arnold/disasters/sleipner.html, 1991.

[26] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT FSE*, pages 135–145, 2008.

[27] R.Y. Rubinstein and D.P. Kroese. *The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-carlo Simulation (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[28] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[29] S.D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.