Finding Rare Numerical Stability Errors in Concurrent Computations

Karine Even
Finding Rare Numerical Stability Errors in Concurrent Computations

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Karine Even

Submitted to the Senate of the Technion — Israel Institute of Technology
Iyar 5773 Haifa April 2013
This thesis was completed under the supervision of Assistant Prof. Eran Yahav (Department of Computer Science, Technion) and Dr. Hana Chockler (IBM Haifa Research Labs) in the Department of Computer Science.

The generous financial support of the Technion is gratefully acknowledged.
# Contents

Abstract 1

1 Introduction 2

2 Overview 6

3 Preliminaries 12
   3.1 Programs as Transition Systems  12
   3.2 Cross-Entropy Method  14
   3.3 Cross-Entropy Method in Optimization Problems  17
   3.4 Cross-Entropy Method Over Graphs  18

4 Cross-Entropy Based Testing for Numerical Stability Problems 22
   4.1 Performance Function for Floating Point Summation and Subtraction Algorithms  23
   4.2 CE-Based Testing with Abstraction  25

5 Evaluation 31
   5.1 Implementation  31
   5.2 Benchmarks  32
   5.3 Methodology  33
      5.3.1 Description of Benchmarks  33
      5.3.2 Generating Benchmarks Inputs  39
      5.3.3 Numerical Stability of Benchmarks  40
      5.3.4 Evaluation Setting  41
   5.4 Experimental Results  43

6 Related Work 46
   6.1 Numerical Stability  46
   6.2 Existing Approaches for Faults Detection  48
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Conclusions and Future Work</td>
<td>52</td>
</tr>
<tr>
<td>7.1 Refine Abstraction on the Fly</td>
<td>53</td>
</tr>
<tr>
<td>7.2 Performance Functions of Elementary Arithmetic</td>
<td>54</td>
</tr>
<tr>
<td>7.3 Removing Detected Numerical Instabilities</td>
<td>55</td>
</tr>
<tr>
<td>8 Appendix</td>
<td>65</td>
</tr>
<tr>
<td>8.1 Upper Bound Probability of Sampling (at least) Two Numbers From Interval Edges</td>
<td>65</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 *Changes in the probability distribution as a result of using the cross-entropy method.* .............................................. 3

2.1 *Array Sum: Pseudo code of a thread* ................................. 6
2.2 *Cross-entropy with Abstraction.* ...................................... 9

3.1 *The pseudo-code of the CE-Method for optimization problems* ... 18
3.2 *The pseudo-code of the CE-Method for graph optimization problems* 19

5.1 *Recursive summation method, sequential version* ................. 34
5.2 *Pairwise summation method, recursive version* .................... 35
5.3 *Pairwise summation method, concurrent version* .................. 36
5.4 *An Example of Concurrent Summation of Eight Elements, Pairwise Summation Method.* ........................................... 37
5.5 *Møller’s serial step-wise version [49]* ............................... 38
5.6 *Random testing of a benchmark P with generation of many inputs from a given distribution f* ............................... 40

7.1 *A high-level description of our future additional functionality to ACE* 56
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 concurrent threads. This can lead to instability of previously stable algorithms, since rounding 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 small fraction of interleavings. In this thesis, we apply the cross-entropy method—a generic approach to rare event simulation and combinatorial optimization—to detect 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.
Chapter 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) [35]. 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 [5], explosion of the Ariane 5 rocket in 1996 was a result of an overflow error [7], sinking of the Sleipner A offshore platform was caused by a combination of a numerical error and a physical error [6], Vancouver Stock Exchange in 1983, an exchange’s index value had dropped due to truncation errors [57], a publication of an incorrect inflation in UK during 1986-1987 was a result of rounding errors and costs DHSS 100 million pounds [69], and the German parliament makeup in 1992 changed after the revealing of rounding errors during the elections to the parliament, which affected the results [71].

Introduction of concurrency to numerical algorithms results in a significant increase in the number of possible computations of the same result, due to different schedules of concurrent threads. This can lead to instability of previously stable algorithms, as rounding 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 small fraction of interleaving, making the detection of such errors an especially challenging task. Avoiding numerical instabilities by sticking to sequential numerical algorithms is not always possible, as explained by Higham in
Five summation methods and their variations are analysed here. The accuracy of the methods is compared using rounding error analysis and numerical experiments. . . . No one method is uniformly more accurate than others, but some guidelines are given on the choice of the method in particular cases.

That is to say, using a method may ensure tighter error bounds but cannot guarantee the reduction of the actual error.

In this thesis, we use the cross-entropy method to detect rare numerical instability in concurrent programs. The cross-entropy method is a generic approach to rare event simulation and combinatorial optimization [62]. It derives its name from the cross-entropy or the Kullback-Leibler distance, which is a fundamental concept of modern information theory [42]. The cross-entropy method has been successfully applied in many problems, including buffer allocation [3], queueing models of telecommunication systems [11, 13], combinatorial auctions [18], neural computation [26], DNA sequence alignment [40], clustering analysis [14, 15, 41], scheduling [46], and graph problems [61]. 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 1.1). The iterations

![Figure 1.1: Changes in the probability distribution as a result of using the cross-entropy method.](image)
in the cross-entropy method consist of two phases:

1. Generate a random data sample according to a specified mechanism.

2. Update the parameters of the random mechanism based on the data to produce a “better” sample in the next iteration.

The cross-entropy method uses the predefined performance function to evaluate a sample. The procedure terminates when the “best” sample, that is, a sample with the maximal value of the performance function (or with a sufficiently small relative deviation, if the global maximum is unknown in advance), is generated. In rare faults detection, we are given a program and an input to search for a numerical instability or other bugs (see [20] for buffer overflow detection with cross-entropy). The cross-entropy based testing searches for executions with faults (e.g., an unstable result, a buffer overflow, a race, an atomic violation, etc.) by directing an execution to areas in which faults can occur, via a predefined performance function. In numerical algorithms, natural performance functions are derived from estimations of the error bounds, as described in [25].

We also address the issue of scalability of the cross-entropy method by 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. Once our method finds a numerical instability in the code, it retrieves a partial trace of the faulty concrete execution with additional information for restoring such faulty executions. Without abstraction our method will not scale for large concurrent programs. The size of the samples and the size of the updated information depend on the number of threads and on the size of the code. Adding abstraction to the method assures smaller samples and adds less information at the end of each iteration. As a result, the information we keep through the iterations is reduced, the search for a bug becomes more efficient and it is less likely the method will run out of memory. We refer to this problem in more details in Chapter 4 for our method and in Chapter 6 for other methods.

We demonstrate our approach on implementations of several numerical algorithms using different methods of summation (see [34] for the description of summation methods in numerical algorithms) on floating point numbers and truncation of results. We compare our approach to several existing testing algorithms and show that our implementation can find errors that are not detected by other methods. Summation methods are part of many algorithms calculating as part of
their steps: norms, means, variance, covariance, inner-product and non-linear functions. For instance, Fast Fourier Transform (FFT) performs cascade summation as a main part of its algorithm for improving the numerical accuracy of the result [17]; counting sort uses prefix-sum in order to determine the starting position in the output array of elements of a certain key [24, 70]. We consider these methods as an important building block of many numerical algorithms, and as a first step for constructing a complete method of rare numerical instabilities detection for many more numerical algorithms.

**Main Contributions.** The contributions of this thesis 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.
Chapter 2

Overview

In this chapter 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, show real life examples of parallel and concurrent numerical algorithms, and explain how the instability can be found using the cross-entropy method.

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 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 2.1, where finalSum (the result of the algorithm), and input_decimal_array (the input array) are both shared variables. While there is actually no “bug” in this algorithm,

\[
\text{Thread}(i): \\
\text{local} = \text{input}_{\text{decimal}}\text{array}[i]; \\
\text{atomic} \{ \\
\text{finalSum} + = \text{local}; \\
\} \\
\]

Figure 2.1: Array Sum: Pseudo code of a thread

it can give different results on the same input 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 3-digit numbers; we are just using this here for clarity of presentation):

\[0.25, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.75, 6.0],\]

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

\[0.25 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.75 + 6.0 = 10.5,\]

which is correct. On the other hand, the order

\[6.0 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.75 + 0.25\]

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 Chapter 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.

Real-Life Algorithms

The computational complexity of real numerical problems is much higher than the complexity of our toy-example ArraySum. Concurrent or parallel algorithms are considered as a plausible solution for large scale numerical problems, but not all sequential algorithms can be parallelized –in many cases concurrency introduces or uncovers numerical instability.
In summation methods, instability of an algorithm is commonly a result of non-determinism related to the input. Following from the rounding error analysis of these methods, no method is numerically more stable than the others and guidelines are used for choosing a suitable summation method [34]; as the suggestion to use the pairwise summation method or the compensated summation method for large arrays summation, or to use the recursive summation method with decreasing order of summation for sums with heavy cancellation. The nature of an algorithm and the unknown in advanced characteristics of the input, both can affect the accuracy of the output (e.g., ascending or descending order of the input, input of opposite sign numbers, the input domain, and the way the input is generated). Concurrency is another source of non-determinism, and thus can affect the numerical stability of a method.

With regards to other numerical algorithms, either there can be a proper translation from sequential to parallel code (as done in [48]) or there is a natural way to write the code in parallel (e.g., pairwise summation and prefix sum). We refer two examples, the first one is QuickStep, a system for parallelizing sequential programs, and the second is Fast Fourier Transform (FFT) algorithms, which use pairwise summation method. QuickStep can be applied on numerical algorithms, as shown in the example of a scientific simulation of pairwise interactions between water molecules (i.e., the simulation calculates the total energy of a system, kinetic and potential energies, by using only interactions between two molecules), while keeping the simulation numerically stable by adding corrections to the basic translation [48]. The Fast Fourier Transform (FFT) calculates the Discrete Fourier Transform (DFT) by using cascade summation, reducing the time complexity from $O(n^2)$ to $O(n \log(n))$ in compare to the naive algorithm. The very small round-off errors of FFT algorithms are due to the fact it sums via cascade summation, reducing the round off error from $O(n)$ to $\log(n)$ in the worst case, and from $O(\sqrt{n})$ to $\sqrt{\log(n)}$ in the average case (with an input size $n$ and a finite-precision floating-point arithmetic, for both cases) [17]. Following from these error bounds of FFT algorithms, a properly implemented FFT will rarely be a significant contributor to the numerical error, as explained in [17]. That is, an inaccurate result of FFT algorithm is rare and yet possible; such an error can be related to the data type we use (as using floats instead of doubles) and the data itself. We explain in Section 5.3.1 the cascade summation method and what kind of errors we can have. The FFT algorithms are used in many fields of science, as medical imaging (CT, MRI, and ultrasonic imaging) [55, 56], data compression [2, 58, 67], solving partial dif-
ferential equations [4], and more.

Our Approach

The type of error described in this chapter 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 Chapter 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 pro-
gram 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 [25] 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 2.2 (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 ×) 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 [20]. In this thesis, 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 2.2 (b)), would hinder effective guidance. A precise-enough abstraction (schematically shown in Figure 2.2 (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 [20], 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 success-
fully 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
[25] can serve as natural performance functions for the numerical stability algo-
rithms, 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.
Chapter 3

Preliminaries

In this chapter we present the basic concepts that we use in our approach. First we describe a program as transition system and explain how a transition system can be viewed as a graph. Then, we introduce the cross-entropy method—a generic approach to rare event simulation and combinatorial optimization. We describe the variant of the method for optimization problems and show how to apply this variant on graphs.

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 functions), to avoid unwinding of loops and inlining of all function calls (see also [20] 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 $\sigma$ and $\sigma'$, if there exists an execution of $P$ that visits $\sigma'$ in the next step after visiting $\sigma$. In particular, this means that one thread changes its location from $\sigma$ to $\sigma'$ 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 $\sigma$ and $\sigma'$ are identical except for the thread $i$);
2. let $st_i$ be the statement in the current location of the thread $i$ in $\sigma$; 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 $\sigma$;
3. the only changes of the values of data between $\sigma$ and $\sigma'$ 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 $\langle \Sigma, T, \Sigma_0 \rangle$, where $\Sigma$ 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 $P$. 

13
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 \( PF : T \rightarrow \mathbb{R} \) 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 \( \sum_{(\sigma,\sigma') \in T} PF((\sigma,\sigma')) = 1 \). 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 \( \Sigma \) 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 Chapter 4 we adapt this framework to finding rare numerical stability errors in concurrent computations.

### 3.2 Cross-Entropy Method

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 \( F \) from this space to \( \mathbb{R}^+ \), and we say that \( e \) occurs on an input \( X \) (from the probability space) if \( F(X) > \gamma \) for some pre-defined 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:

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

(3.1)
Note that this is not a distance in the formal sense, since in general it is not symmetric. Since \( g \) 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 \( g \) 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[62].

**Formal definition of the Cross-entropy Method.** Let \( \mathcal{X} \) be a set of random vectors \( \vec{X} = (X_1, \ldots, X_n) \) (in the general case, \( \mathcal{X} \) is a space of vectors). Let \( \{f(\cdot; v)\} \) be a family of probability density functions (pdfs), where \( v \) is a parameter vector. Let \( F : \mathcal{X} \to \mathbb{R}^+ \) be a function that gives each vector in \( \mathcal{X} \) a non-negative value. In the general version of the CE method, we estimate the probability

\[
l = P_u(F(X) \geq \gamma) = E_u I_{\{F(X) \geq \gamma\}} \tag{3.2}
\]

for some \( \gamma \in \mathbb{R} \) under pdf \( f(\cdot; u) \). The rare event definition is the same as in the overview of the cross-entropy method, at the beginning of this section, that is, if this probability \( l \) is very small, say, smaller than \( 10^{-5} \), we say that \( \{F(X) \geq \gamma\} \) is a rare event.

A straightforward way to estimate \( l \) by using use Crude Monte Carlo (CMC) simulation (drawing a random sample \( X_1, \cdots, X_n \) from \( \mathcal{X} \) according to \( f(\cdot; u) \) and then estimate \( l \) by examining the sample) is infeasible as explained before. A better way, which is based on importance sampling, is to draw the sample according to some other probability density function \( g \) that raises the probability of \( F(X) \geq \gamma \). Using the density \( g \) we can rewrite Equation 3.2 as

\[
l = \int I_{\{F(X) \geq \gamma\}} \frac{f(x; u)}{g(x)} g(x) dx = E_g I_{\{F(X) \geq \gamma\}} \frac{f(x; u)}{g(x)}, \tag{3.3}
\]

where \( g \) is called the importance sampling density. An unbiased estimator of \( l \) is

\[
\hat{l} = \frac{1}{N} \sum_{i=1}^{n} I_{\{F(X_i) \geq \gamma\}} \frac{f(X_i; u)}{g(X_i)} \tag{3.4}
\]

where \( \hat{l} \) called importance sampling (IS) estimator.
The ideal importance sampling density to estimate $l$ is
\[ g^*(x) = \frac{I\{F(x) \geq \gamma\} f(x; u)}{l} \]  
(3.5)
by using this change of measure in Equation 3.4 we get, for all $i$, the following equation
\[ l = I\{F(x) \geq \gamma\} \frac{f(X_i; u)}{g^*(X_i)}. \]  
(3.6)
However, $g^*$ function depends on $l$ which is unknown, and so, the CE method approximate the ideal $g^*$ by choosing tilting parameter $v$ such that distance between $g^*$ and $f(\cdot; v)$ is minimal. It is convenient to choose $g^*$ from the \{f(\cdot; v)\} family. The Kullback-Leibler “distance” in Equation 3.1 measures the distance between two densities $g^*$ and $f$, when the distance is 0 when the two densities are the same. Minimizing the Kullback-Leibler “distance” between $g^*$ in 3.5 and $f(\cdot; v)$ is equivalent to solving the maximization problem
\[ \max_v \int g^*(x) \ln f(x; v) dx, \]  
(3.7)
and by substituting $g^*$ from 3.5 we obtain equivalent maximization problem
\[ \max_v \int \frac{I\{F(x) \geq \gamma\} f(x; u)}{l} \ln f(x; v) dx \]  
(3.8)
which is equivalent to
\[ \max_v D(v) = \max_v E_u I\{F(x) \geq \gamma\} ln f(x; v) \]  
(3.9)
and using importance sampling again with a change measure $f(\cdot; w)$ we can write
\[ \max_v D(v) = \max_v E_w I\{F(x) \geq \gamma\} \frac{f(x; u)}{f(x; w)} ln f(x; v) \]  
(3.10)
for any reference parameter $w$. The optimal solution is
\[ v^* = \arg \max_v E_w I\{F(x) \geq \gamma\} \frac{f(x; u)}{f(x; w)} ln f(x; v) \]  
(3.11)
and can be estimated by following stochastic program

$$\max_v \bar{D}(v) = \max_v \frac{1}{N} \sum_{i=1}^{N} I_{\{F(X_i) \geq \gamma\}} \frac{f(X_i; u)}{f(X_i; w)} \ln f(X_i; v), \quad (3.12)$$

where $X_1, \ldots, X_N$ is a random sample from $f(\cdot; w)$.

The CE method draws the sample according to $f$ and iteratively changes $f$ to minimize the Kullback-Leibler distance (also called the cross-entropy) between $f$ and $g_\gamma$ until the minimal cross-entropy between $f$ and $g_\gamma$ is achieved. The Kullback-Leibler distance $\mathcal{D}(g, h)$ between discrete probability densities $g = \{g_1, \ldots, g_n\}$ and $h = \{h_1, \ldots, h_n\}$ is defined as

$$\mathcal{D}(g, h) = \sum_i g_i \cdot \log_2 \frac{g_i}{h_i}. \quad (3.13)$$

For continuous probability densities, the sum is replaced by an integral.

### 3.3 Cross-Entropy Method in Optimization Problems

In this section, we briefly describe a variant of the cross-entropy method adapted for optimization problems (see [12] for a full description of the method). In Section 3.4, we present a version of this method for optimization problems on graphs.

In optimization problems we define a *performance function* $S : \mathcal{X} \to \mathbb{R}^+$ ($S$ gives each vector in $\mathcal{X}$ a non-negative value). We want to find the maximum of $S$ over $\mathcal{X}$ and the corresponding vector(s) of $\mathcal{X}$ in which this maximum is reached. In this setting $\gamma$ is unknown in advance. We denote the maximal $\gamma$ as $\gamma^*$. Thus we search for all vectors such that

$$S(x^*) = \gamma^* = \max_{x \in \mathcal{X}} (S(x)), \quad (3.14)$$

and we still assume that $\{S(x) \geq \gamma^* = S(x^*)\}$ is a rare event (as in the original setting).

The CE method for optimization produces a sequence of levels $\{\gamma_t\}$ and reference parameters $\{v_t\}$. Through the iterations $\gamma_t$ and $v_t$ tend to their optimal $\gamma^*$ and $v^*$ accordingly. In each iteration we generate samples according to the current probability distribution and compute the score values for these samples (via our performance function). Then, at the end of the iteration, we update the probability
distribution so that method assigns higher probability to some (small) percentage of the sample with the highest values of $S(x)$. We continue until a stopping condition is reached. In this variant, since $\gamma^*$ is not known in advance, the stopping condition is reached when the relative standard deviation is below some threshold parameter. The algorithm of CE method for optimization problems is shown in Figure 3.1, given a density $u$ (commonly sets to be uniform), a quantile size $q$ of the top best samples ($qN$ out of $N$). The updated density $v_t$ is computed by solving Equation 3.12, with $W = f(X_i;u) / f(X_i;w) = 1$. We set $W = 1$ since the choice of the initial parameter $u = v_0$ is quite arbitrary, so using $W$ would be meaningless, while in rare event simulation it is an essential part of the estimation problem.

Define $v_0 = u$
Set $t = 1$ (level counter)
In each iteration $t$ do:
Generate a sample $\{X_1, \ldots, X_N\}$ of size $N$ from the density $f(;v_{t-1})$
Sort the sample $\{X_1, \ldots, X_N\}$ in ascending order according to the performance function $S$ for all $X_i$'s
Set the best part of the sample to be $Q(X_i) = \{X_{\lfloor (1-q)N \rfloor}, X_{\lfloor (1-q)N+1 \rfloor}, \ldots, X_N\}$
Update $v_t$ to $v_{t+1}$ by solving 3.12 with $W = 1$ and the sample $\{X_1, \ldots, X_N\}$

Figure 3.1: The pseudo-code of the CE-Method for optimization problems

### 3.4 Cross-Entropy Method Over Graphs

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 travel-
eling 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 3.2, 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, 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 in $i$-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

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

(3.15)

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 $e$. 

Figure 3.2: The pseudo-code of the CE-Method for graph optimization problems
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%).

As explained in Section 3.2, each iteration the CE method solves Equation 3.12 based on the solution of the previous iteration. In CE method for optimization problems we solve this equation with $W = 1$ as shown in Figure 3.1 in Section 3.3. Our goal is to find the optimal edges probability vector $v^*$

$$v^* = \arg\max_v E_0 I_{\{S(\pi) \geq \gamma\}} \ln f(\pi; v),$$  
(3.16)

which is the probability of going from the current node to the next node, out of all enabled nodes (defined for all edges of $G$). Given the current set of the best sampled paths $Q_i$, we set $v$ thus the probability of traversing edges in $Q_i$ is maximized in $f_{i+1}(\pi; v)$.

We demonstrate the solution by the example below. In our setting of the problem $\vec{X}$ is a path on the graph, and $v$ is a parameter that indicates for each edge the probability of taking it.

Let $\vec{X}$ be a random vector $(X_1, \ldots, X_N) \sim Ber(p)$, and parameter $v = p$. Suppose that we do not know which components of $\vec{X}$ are 0 and which are 1. There is an “oracle” that computes for us the performance function for each binary vector; where the performance function represents the number of “matches” between the unknown vector we are looking for $\vec{X}^*$ and the vector we got $\vec{X}$. For this problem (Bernoulli distribution where $v = p$), the probability density function is

$$f(\vec{X}; p) = \prod_{i=1}^{N} p_i^{X_i} (1 - p_i)^{1 - X_i},$$  
(3.17)

and since each $X_j$ can only be 0 or 1, we have

$$\frac{\partial}{\partial p_j} \ln f(X; p) = \frac{X_j}{p_j} - \frac{1 - X_j}{1 - p_j} = \frac{X_j - p_j}{(1 - p_j)p_j}. $$  
(3.18)

We can compute the maximum in Equation 3.12 by setting the first derivatives with
respect to $p_j$ equal to zero, for $j = 1, \ldots, N$:

$$
\frac{\partial}{\partial p_j} \sum_{i=N}^N I_{\{S(X_i) \geq \gamma\}} \ln f(X_i; p) = \frac{1}{(1 - p_j)p_j} \sum_{i=N}^N I_{\{S(X_i) \geq \gamma\}} (X_{ij} - p_j) = 0,
$$

(3.19)

where $X_{ij}$ is $X_j$ in sample $i$. Thus, we have

$$
p_j = \frac{\sum_{i=N}^N I_{\{S(X_i) \geq \gamma\}} X_{ij}}{\sum_{i=N}^N I_{\{S(X_i) \geq \gamma\}}}.
$$

(3.20)

That is, the probability $p_j$ is the ratio between all the samples with event $\{S(X_i) \geq \gamma\}$ that the $j$-th location is “1” and all the samples with event $\{S(X_i) \geq \gamma\}$. Our $f_{i+1}$ indicates the ratio between all samples that the edge $e$ was taken and the samples that can take edge $e$, when we set $v$ to be an edges probability vector.

**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 1:

$$
f_{i+1}^e = \alpha f_i(e) + (1 - \alpha) f_{i+1}(e),
$$

(3.21)

where $0 < \alpha \leq 1$ is the smoothing parameter. Clearly, for $\alpha = 0$ we have the original updating equation. Usually, a value of $\alpha$ 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).
Chapter 4

Cross-Entropy Based Testing for Numerical Stability Problems

In this chapter, 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 probability 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 “condense” 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. It is the best to note that the definition of the correct result can also be the common result, a known result, or an output of other algorithms that calculate the same function. It is our choice how to define the correct result. Here we choose to refer the correct result as the sequential version of the algorithm. 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 [34].

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 [25]. 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 \( \pi_i \) of a path \( \pi \) on the graph \( G \) corresponding to the
program \( P \), the value of the performance function for addition \( S^+ \) on \( \pi_i \) is

\[
S^+(\pi_i) = \sum_{i=0}^{y} \left( |\exp(r_i) - \exp(e_i)| + \text{digits}(r_i) + \right. \\
\left. \text{digits}(e_i) + \text{digits}\left(\exp\left(\max(|r_i|,|e_i|)\right)\right) \right), \tag{4.1}
\]

where

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

\[
S^-(\pi_i) = \sum_{i=0}^{y} \left( |\exp(|r_i - e_i|) - \exp(\max(|r_i|,|e_i|))| + \right. \\
\left. \text{digits}(|r_i - e_i|) + \text{digits}\left(\exp\left(\max(|r_i|,|e_i|)\right)\right) \right), \tag{4.2}
\]

where \( e_i, r_i, \) and \( \text{digits} \) 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, \( S^{+-} = (S^+, S^-) \). Then, for paths \( \pi_1 \) and \( \pi_2 \), 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
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 \( \log(\text{data}) \). 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 concrete 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.). Let $PTYPE = \{1, \ldots, m\}$ be a set of thread-sets identifiers. The abstract thread identifiers function, $\text{Abstract-PID} : \text{PID} \rightarrow PTYPE$, matches thread identifiers to their thread-sets identifiers according to chosen common characteristics. An example of an $\text{Abstract-PID}$ function with a specific $\text{PID}$ and $PTYPE$ is given in the implementation part (Section 5.1).

**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. We denote the abstract domain as $D$, and the data abstraction function as $\text{absData}_D : R \rightarrow D$. The data abstraction function translates values from $R$ to values in $D$, e.g.,

- “full abstraction” function is $\text{absData}^{1}_{\{\emptyset\}} : R \rightarrow \{\emptyset\}$, $\text{absData}^{1}_{\{\emptyset\}}(\emptyset) = \text{ConstantFunction}$; $\text{ConstantFunction}$ is a constant function that maps all values to $\emptyset$.

- “log abstraction” function is $\text{absData}^{2}_{N} : R \rightarrow N$, $\text{absData}^{2}_{N} = \text{round} \circ \text{log}$; the function $\text{round} : R \rightarrow N$ converts numbers from $R$ to $N$, thus for any $x \in R$, $\text{round}(x) \in N$ is the closest number to $x$ in $N$, and the function $\text{log} : R \rightarrow R$ is the logarithm function to base 2 of numbers in $R$. 

26
• “no data abstraction” function is \( \text{absData}_3 : \mathbb{R} \rightarrow \mathbb{R} \), \( \text{absData}_3(x) = \text{IdentityFunction} \); \( \text{IdentityFunction} \) is the identity function.

**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. We denote the thread counter abstraction function as \( \text{absCounter}_{D_k} : \mathbb{N} \rightarrow D_k = \{0, 1, \ldots, k\} \) for a given bound \( k > 0, k \in \mathbb{N} \), thus \( D_k \subseteq \mathbb{N} \) and \( \text{absCounter}_{D_k}(i) = \min(i, k) \). We set the counter abstraction function to be the identity function in case there is no thread counter abstraction (denote as \( \text{absCounter}_{\mathbb{N}} \)).

**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 thesis, 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 [20]) and by giving equivalent program locations the same value of the counter. We denote the thread location abstraction function as \( \text{absLabel} : Label \rightarrow Label_{\text{abs}} \).

**Transition System with Abstractions.** These abstractions can be used in conjunction with each other—for example, we can abstract the thread counters and data at the same time. We present the definitions of an abstract transition system in general, thus any conjunction of these abstractions is possible.

An abstract state records the current location of threads and their data; it is a tuple of abstract program counters, the state abstract location, and shared data under some data abstraction. An abstract program counter of a thread consists of its abstract thread identifier, its abstract program location, and an abstraction of thread-local data. The state abstract location is all threads’ abstract program
counters under thread counter abstraction.

**Definition 5 (Abstract Program State)** An abstract program state $\tilde{\sigma} \in \Sigma_{abs}$ is a tuple $\langle pc_{abs}, state_{pc_{abs,\tilde{\sigma}}}, Sdata_{abs,\tilde{\sigma}} \rangle$, thus $pc_{abs}$ is the set of abstract program counters ($pc_{abs} = (Abstract-PID \times Label_{abs} \times (Lvar \to D))$), $state_{pc_{abs,\tilde{\sigma}}}$ defines the state abstract location under thread counter abstraction ($state_{pc_{abs,\tilde{\sigma}}}: pc_{abs} \to \mathbb{D}_{\text{counters}}$), and $Sdata_{abs,\tilde{\sigma}}$ defines the shared variables’ current value under some data abstraction ($Sdata_{abs,\tilde{\sigma}}: Svar \to \mathbb{D}$).

Given, an abstract domain $\mathbb{D}$, an abstract thread identifiers function $Abstract-PID$, a number $k > 0$ ($k \in \mathbb{N}$), and a counter abstraction domain $\mathbb{D}_{\text{counters}}$, which is $\mathbb{D}_{k}$ for finite $k$ or else is $\mathbb{N}$.

We define $abs$ to be an abstraction function from a concrete state to an abstract state, thus any concrete state $\sigma \in \Sigma$ has an abstract state $abs(\sigma) \in \Sigma_{abs}$.

**Definition 6 (Abstraction Function of a Program State)** An abstraction function for a program state (abstraction function) is a function $abs: \Sigma \to \Sigma_{abs}$, $abs(\sigma) = \tilde{\sigma}$ for any concrete state $\sigma = \langle PID, pc_{\sigma}, Ldata_{\sigma}, Sdata_{\sigma} \rangle \in \Sigma$, and an abstract state $\tilde{\sigma} = \langle pc_{abs,\tilde{\sigma}}, state_{pc_{abs,\tilde{\sigma}}}, Sdata_{abs,\tilde{\sigma}} \rangle \in \Sigma_{abs}$, that maintain the location and the data constraints:

- **Location Constraints:** for any abstract program counter $(m, l, \text{ldata}) \in pc_{abs}$ : $state_{pc_{abs,\tilde{\sigma}}}(m, l, \text{ldata}) = absCounter_{\mathbb{D}_{\text{counters}}}(n)$ where $n \in \mathbb{N}$ and $n$ is the number of concrete threads in $(m, l, \text{ldata})$ (i.e., $n = |\{\text{pid}|(\text{pid} \in PID) \land (Abstract-PID(\text{pid}) = m) \land (\text{absLabel} \circ pc_{\sigma}(\text{pid}) = l) \land (\text{ldata} = absData_{(\mathbb{D})} \circ Ldata_{\sigma}(\text{pid}))\}|$).

- **Shared Data Constraint:** $Sdata_{abs,\tilde{\sigma}} = absData_{(\mathbb{D})} \circ Sdata_{\sigma}$ is shared data of the state $\sigma$ under data abstraction.

Given an abstract domain $\mathbb{D}$ and a domain $\mathbb{D}_{\text{counters}}$, an abstract thread identifiers function $Abstract-PID$, a counter abstraction function $absCounter_{\mathbb{D}_{\text{counters}}}$, a data abstraction function $absData_{(\mathbb{D})}$, and a thread location abstraction function $absLabel$.

Abstract transitions describe concrete execution steps between abstract states. In existential abstraction, for every concrete transition $\langle \sigma, \sigma' \rangle$ there is an abstract transition $\langle abs(\sigma), abs(\sigma') \rangle$ ($\sigma, \sigma' \in \Sigma$ and $abs(\sigma), abs(\sigma') \in \Sigma_{abs}$).

**Definition 7 (Abstract Program Transition)** An abstract program transition $t \in \Sigma_{abs} \times \Sigma_{abs}$ is a pair $\langle abs(\sigma), abs(\sigma') \rangle$ thus $\sigma, \sigma' \in \Sigma$ and $\langle \sigma, \sigma' \rangle \in T$, given an
abstraction function \( \text{abs}(\cdot) \), and a transition system \( TS = \langle \Sigma, T, \Sigma_0 \rangle \) of program \( P \).

We denote the set of all abstract transitions of program \( P \) as \( T_{\text{abs}} \) where \( T_{\text{abs}} \subseteq \Sigma_{\text{abs}} \times \Sigma_{\text{abs}} \).

An abstraction of a program \( P \) defines a transition system \( TS_{\text{abs}} \) in a natural way and an abstract graph \( G_{\text{abs}} = \langle V_{\text{abs}}, E_{\text{abs}} \rangle \) similarly to the definitions of the concrete transition system and the corresponding graph in Section 3.1. We represent an execution of \( P \) as a sequence of abstract program states (instead of a sequence of program states), where we bias the search for a erroneous concrete execution through an abstract transition system. We choose a next abstract transition out of all possible abstract transitions from the current abstract state, and only then translate it to a concrete transition to traverse. Clearly, abstract transitions do not correspond to threads’ interleavings directly.

**Definition 8 (Abstract Transition System)** For a transition system \( TS = \langle \Sigma, T, \Sigma_0 \rangle \) of program \( P \), and given an abstraction function \( \text{abs}(\cdot) \), we define an Abstract Transition System of \( P \) as a tuple \( TS_{\text{abs}} = \langle \Sigma_{\text{abs}}, T_{\text{abs}}, \Sigma_{\text{abs},0} \rangle \), thus \( \Sigma_{\text{abs}} \) is the set of abstract states (\( \sigma \in \Sigma \iff \text{abs}(\sigma) \in \Sigma_{\text{abs}} \)), \( T_{\text{abs}} \) is the set of abstract transitions (\( \forall \sigma, \sigma' \in \Sigma. \langle \sigma, \sigma' \rangle \in T \iff \langle \text{abs}(\sigma), \text{abs}(\sigma') \rangle \in T_{\text{abs}} \)), and \( \Sigma_{\text{abs},0} \) is the non-empty set of initial abstract states of \( P \) (\( \Sigma_{\text{abs},0} \subseteq \Sigma_{\text{abs}} \land \Sigma_0 \neq \emptyset \) and \( \forall \sigma_0 \in \Sigma_0. \text{abs}(\sigma_0) \in \Sigma_{\text{abs},0} \)).

The values of the probability distribution of each abstract transition are computed using the same formula as in concrete transition systems, replacing the transition system \( TS \) in the transition system \( TS_{\text{abs}} \). 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. 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 “condenses” 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
Chapter 5, sometimes a finer abstraction ends up to be more efficient than a coarser one.

**Over Approximation of the (Concrete) Transition System.** In order to guarantee that any error that can be found in the concrete transition system can be found in the abstract transition system, we have to show that any possible concrete execution of $P$ in the concrete transition system has an abstract finite path in the abstract transition system. That is, for any (concrete) program transition system $T S = \langle \Sigma, T, \Sigma_0 \rangle$ and an abstraction function $abs(\cdot)$, there is an abstract transition system $T S_{ab} = \langle \Sigma_{ab}, T_{ab}, \Sigma_{ab,0} \rangle$, thus for every concrete finite path $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_j$, there is an abstract finite path $\tilde{\sigma}_0, \tilde{\sigma}_1, \tilde{\sigma}_2, \ldots, \tilde{\sigma}_j$, where for all $i \ (0 \leq i \leq j)$, $\sigma_i \in \Sigma$, $\tilde{\sigma}_i \in \Sigma_{ab}$ and $abs(\sigma_i) = \tilde{\sigma}_i$.

We show that any finite execution in the concrete transition system has an abstract representation in the abstract transition system, simply by showing an abstract transition system that maintains this. Let $T S_{ab} = \langle \Sigma_{ab}, T_{ab}, \Sigma_{ab,0} \rangle$ be an abstract transition system for a given (concrete) transition system $T S = \langle \Sigma, T, \Sigma_0 \rangle$ and an abstraction function $abs(\cdot)$. We define $\Sigma_{ab}$, $\Sigma_{ab,0}$, and $\Sigma_{ab,0}$ as $\Sigma_{ab} = \{ abs(\sigma) | \sigma \in \Sigma \}$, and $T_{ab} = \{ (\langle\sigma, \sigma'\rangle) | (\sigma, \sigma') \in T \}$, and $\Sigma_{ab,0} = \{ abs(\sigma_0) | \sigma_0 \in \Sigma_0 \}$. Any concrete finite path $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_j$ has an abstract finite path $abs(\sigma_0), abs(\sigma_1), abs(\sigma_2), \ldots, abs(\sigma_j)$, where $\sigma_i \in \Sigma$ and $abs(\sigma_i) \in \Sigma_{ab}$ for all $i \ (0 \leq i \leq j)$, thus $\sigma_0 \in \Sigma_0$ and $abs(\sigma_0) \in \Sigma_{ab,0}$ and all the abstract transitions $\langle \sigma_0, abs(\sigma_i) \rangle$ $(0 \leq i < j)$ are in $T_{ab}$, as required.
Chapter 5

Evaluation

In this chapter, 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 [20, 21].

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.

**Probability Table.** 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 concrete 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 \( \sigma \) (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 \( \sigma \) and is 0 otherwise. In an abstract transition system of a program with at most \( n \) concrete threads, the probability vector of an abstract state \( \sigma_{\text{abs}} \in \Sigma_{\text{abs}} \) is an \( n' \)-dimensional vector of outgoing abstract transitions of \( \sigma_{\text{abs}} \), where \( n' \leq n \) and each outgoing transition represent one thread or more at an abstract location.

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 and thus the probability to have a collision drops).

### 5.2 Benchmarks

Higham [34] 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 compensated summation method.

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

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

Figure 5.1: Recursive summation method, sequential version

(1 \leq i \leq n) in a given order (see Figure 5.3.1). The concurrent version of this algorithm is presented in Figure 2.1 (our toy example in Chapter 2).

We consider two variants of recursive summation: \textit{ArraySum} over input arrays of non-negative numbers, and \textit{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 \textit{ArraySum} and \textit{ArrayDiff} by using the performance functions for floating points numbers described in Section 4.1, where in \textit{ArrayDiff}, the performance function is a tuple \((S^+, S^-)\). There are no subtractions operation \textit{Array Sum}, as it sums only non-negative numbers, hence its performance function is \(S^+\).

The \textbf{pairwise summation method}, also known as the cascade summation method, sums the set of given numbers in pairs (cf. [34, 8, 45]); 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 \lfloor \frac{n}{2} \rfloor) recursively, as shown in Figure 5.2. The concurrent version of this algorithm has \(\lfloor \log_2 n \rfloor\) stages [36], calculating the items of \(\{y_i\}\) in parallel during each stage. It is based on the basic algorithm of Figure 5.2, when each call to \textit{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: \textit{Tree Schema ArraySum}, which is the concurrent version of the basic algorithm, and \textit{PrefixSum} that returns all the partial sums along the way in addition to the final result. We explain these benchmarks in more details below.
Input:
decimal numbers sequence \( \{array_i\}, i \in [1, n] \)

Algorithm:
\[ int n = \text{size of array}; \]
\[ \text{finalSum} = \text{call PairwiseSum}(array, n); \]

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

Figure 5.2: Pairwise summation method, recursive version

Tree Schema Array Sum sums a sequence of \( n \) numbers. It creates \( n \) threads, where each thread executes the code shown in Figure 5.3. Each thread atomically updates 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 5.4. The value of each element of the sequence \( \{y_i\} \) is affected by the order of summation of the elements constructing it. The algorithm is unstable if the final result (stored in \( y_1 \)) is inaccurate. We search for a fault via \( S^+ \).

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 [43] can be informally described as follows:

- Calculate \( \log(n) \) temporary sequences, \( \{z^l_i\} \ (0 \leq l \leq \log(n)) \), where \( z^l_i = z^l_{i-1} + z^l_{i-2^l-1} \) if \( i + 1 \) divided in \( 2^l \), otherwise the value stays the same (i.e.,
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 \( i \in \{2 \ldots n\} \) performs:
\{
    \text{int} \ l = \max\{j \in [1, \log_2(n)] \wedge ((i-1) \mod (2^j) == 0)\}(j); \\
    \text{while} (z_i < l); \\
    \text{atomic} \\
    \{ \\
    y_{i-2^l} += y_i; \\
    + z_{i-2^l}; \\
    \}
\}

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

Figure 5.3: Pairwise summation method, concurrent version

\( z_i^l = z_{i-1}^{l-1} \). The first sequence (where \( l = 0 \)) is the input sequence.

- Calculate another temporary set of sequences, \( \{w_i^l\} \), where \( w_{i+2^l-1}^l = w_{i+2^l-1}^l + w_i^l \) 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 \( \{w_i^0\} \) (we set \( \{y_i\} \) to be \( \{w_i^0\} \)).

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
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.

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 $w_i = \sum_{j=0}^{\lfloor \log(\sqrt{i+1}) \rfloor} a_{i,j}$, and $a_{i,j} = z_{i-(2^j-1)}^j$ if $((i+1) \land (2^j)) = (2^j)$ 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 errors, 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 [44] 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 [39] for floating point numbers and by Møller [50] 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 5.5). In the concurrent version of the algorithm in Figure 5.5, we set

```plaintext
finalSum = 0;
error = 0;
for all $i \in \{1 \ldots n\}$ do:
{
    item_with_error = input_decimal_array[i] + error;
    error = item_with_error - ((finalsum +
        item_with_error) - finalsum);
    finalSum += item_with_error;
}
```

Figure 5.5: Møller’s serial step-wise version [49]

$finalSum$, $error$ 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, 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 Generating Benchmarks Inputs

In this section we describe the inputs for our benchmarks, how we generate an input, and why an input is rare. We need to generate an input or to assume it is given to us, since our tool ACE is based on a dynamic approach for fault detection.

The input is a set of decimal numbers drawn from a given normal or uniform distribution, with several additional numbers, which are rarely sampled; not all inputs will lead to a numerical instability in the code, even with the additional numbers. In order to find an input, which can lead to a numerical instability, we compare the results of summation at different orders (descending, ascending, descending of the absolute values, ascending of the absolute values, etc.). If any of the results was different (even if still stable), we try to find a bug using ACE and random testing. The results of these experiments are shown in Table 5.1 and Table 5.2. Random testing results refer to 15,000 or 30,000 executions per benchmark, and uniform over edges testing results refer to 1,000 or 3,000 executions per benchmark.

Normal and uniform distributions are very common for numerical applications; for example, when the inputs are readings of some physical measuring device, like a thermostat, and therefore we choose to sample from these distributions. We used their discrete form, a binomial distribution and a discrete uniform distribution, since practically the input is a set of decimal numbers in a finite precision, which is the case when using common data types (e.g., float, double, etc.).

In order to generate an input array, we do not sample from a given discrete distribution over the input domain, but rather mix elements from the middle of the domain with elements from the edges, to raise the probability of numerical instability for concurrent summation algorithms. When all elements are from the middle of the domain, any order of summation will cause almost the same relatively minor chopping/rounding of the intermediate result. A single number from the interval edges probably will cause the same chopping/rounding error of the output, with little dependency on the order of addition of the other elements, and unnecessarily affect the final result differently. Few or several elements, which are extremely smaller or larger than the common elements, can lead to a different behavior, and uncovers numerical instability in the code.

The input array created through this process is rare, since sampling from the
interval edges at least twice is a rare event (see Appendix 8.1 for a proof), and is an array in which the error is rare under the assumption that the algorithms are numerically stable (otherwise, there is no point searching a bug).

5.3.3 Numerical Stability of Benchmarks

Given a pair of a program under test and an input, we use \texttt{ACE} to search a numerical instability in the code while assuming the code is numerically stable. We explain now why these benchmarks are assumed to be numerically stable.

We use random testing in order to find whether a code of a benchmark is numerically stable or not. We perform the algorithm shown in Figure 5.6 in order to find numerical errors. We consider a program to be numerically stable for a given distribution $f$, if many unbiased executions of the code is never erroneous for all sampled arrays from this distribution. In our experimental results we set

Input: program $P$, discrete distribution $f$, and $n, m, k \in \mathbb{N}$

Output: “true” if no numerical stability error found in $P$, else “false”

Algorithm:

\begin{verbatim}
set array $x$ of size $k$;
for $i \in \{1 \ldots n\}$ do:
\{ 
  $x = \text{sample } k \{ \text{numbers from } f \text{ into an array};$
  for $j \in \{1 \ldots m\}$ do:
  \{ 
    run $P$ with input array $x$;
    if the result is abnormal return false;
  \} 
\} 
return true;
\end{verbatim}

Figure 5.6: Random testing of a benchmark $P$ with generation of many inputs from a given distribution $f$

$n = 100$ and $m = 5000$, and use two distributions: binomial distribution-$(n,p)$ and uniform distribution-$(n,a,b)$. We set $f$ to be either $(n, p) = (11099, 0.5)$ when using the binomial distribution or $(n, a, b) = (11099, 111.0, 0.01)$ when using the
uniform distribution. As the results show, even after 850 hours no numerical error was detected, which indicates that there are numerically stable settings for our benchmarks. The rest of the experimental results of the random testing with many generated arrays described in Section 5.4.

The benchmarks in Tables 5.1 and 5.2, and in Table 5.3 were tested on different domains. We use smaller domains for the benchmarks in Table 5.3 because the complexity of performing the experiment with a larger \( n \) requires much more space and time. For instance, we perform \( 2^d \) Bernoulli experiments for each sampled element, given a binomial distribution \( (n, p) \) and \( d = \log(n) \) is the number of digits. Even for relatively small \( d \), the number of operations required per input array is not trivial (e.g., prefixSum 128 with an input array of 20-digit elements, requires over \( 128 \cdot (10^{20}) \) Bernoulli experiments).

5.3.4 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 – 2640M CPU @ 2.80GHz, 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@2.83GHz, 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.
Table 5.1: *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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Random</th>
<th>Uniform over edges</th>
<th>Concrete</th>
<th>Abs1 Log-3Val</th>
<th>Abs2 Log-Binary</th>
<th>Abs3 Full-3Val</th>
<th>Abs4 Full-Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArraySum 10</td>
<td>31.67%</td>
<td>9.3%</td>
<td>10/3</td>
<td>10/2</td>
<td>10/1</td>
<td>5/1</td>
<td>5/2</td>
</tr>
<tr>
<td>ArraySum 100</td>
<td>0.01%</td>
<td>T-O</td>
<td>T-O</td>
<td>240/1</td>
<td>300/1</td>
<td>300/1</td>
<td>T-O</td>
</tr>
<tr>
<td>ArraySum 150</td>
<td>0.05%</td>
<td>0.133%</td>
<td>320/1</td>
<td>260/1</td>
<td>390/1</td>
<td>140/1</td>
<td>130/1</td>
</tr>
<tr>
<td>ArraySum 250</td>
<td>0.11%</td>
<td>T-O</td>
<td>OOM</td>
<td>180/1</td>
<td>120/1</td>
<td>T-O</td>
<td>300/1</td>
</tr>
<tr>
<td>ArraySum 500</td>
<td>0.08%</td>
<td>0.0497%</td>
<td>OOM</td>
<td>T-O</td>
<td>195/1</td>
<td>T-O</td>
<td>180/1</td>
</tr>
<tr>
<td>ArraySum 1000</td>
<td>0.04%</td>
<td>0.106%</td>
<td>OOM</td>
<td>T-O</td>
<td>150/1</td>
<td>T-O</td>
<td>350/1</td>
</tr>
<tr>
<td>ArrayDiff 100</td>
<td>0.03%</td>
<td>0.1%</td>
<td>T-O</td>
<td>240/1</td>
<td>300/1</td>
<td>360/1</td>
<td>T-O</td>
</tr>
<tr>
<td>ArrayDiff 150</td>
<td>0.08%</td>
<td>0.233%</td>
<td>T-O</td>
<td>300/1</td>
<td>400/1</td>
<td>240/1</td>
<td>200/1</td>
</tr>
<tr>
<td>TreeArraySum 128</td>
<td>2.71%</td>
<td>0.1%</td>
<td>140/2</td>
<td>120/1</td>
<td>200/1</td>
<td>180/1</td>
<td>150/2</td>
</tr>
<tr>
<td>TreeArraySum 128</td>
<td>0.9%</td>
<td>1.7%</td>
<td>150/3</td>
<td>80/1</td>
<td>90/1</td>
<td>80/1</td>
<td>150/1</td>
</tr>
<tr>
<td>PrefixSum 128</td>
<td>0.0067%</td>
<td>2.0013%</td>
<td>T-O</td>
<td>120/4</td>
<td>150/4</td>
<td>120/6</td>
<td>150/5</td>
</tr>
<tr>
<td>PrefixSum 128</td>
<td>0.4%</td>
<td>3.776%</td>
<td>90/6</td>
<td>80/7</td>
<td>80/6</td>
<td>80/4</td>
<td>120/10</td>
</tr>
<tr>
<td>S + /S – 100</td>
<td>0.04%</td>
<td>0.1%</td>
<td>T-O</td>
<td>T-O</td>
<td>100/1</td>
<td>150/1</td>
<td>300/1</td>
</tr>
<tr>
<td>S + /S – 150</td>
<td>0.05%</td>
<td>T-O</td>
<td>300/1</td>
<td>T-O</td>
<td>T-O</td>
<td>300/1</td>
<td></td>
</tr>
<tr>
<td>Kahan 150</td>
<td>0.33%</td>
<td>0.0637%</td>
<td>480/1</td>
<td>120/1</td>
<td>120/1</td>
<td>120/1</td>
<td>120/1</td>
</tr>
<tr>
<td>Kahan 500</td>
<td>0.5133%</td>
<td>0.1%</td>
<td>OOM</td>
<td>210/2</td>
<td>195/1</td>
<td>325/2</td>
<td>240/1</td>
</tr>
</tbody>
</table>
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 5.1 and Table 5.2 show the results of running $ACE$ on our benchmarks. Table 5.1 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 Table 5.2 lists 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).

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. $TreeArraySum$ was executed with 128 threads and $PrefixSum$ was executed with 256 threads; both benchmarks were executed on two different input arrays with 128 elements. $S+$
Table 5.2: Effectiveness of bug detection: A comparison of the memory consumption for our benchmarks. Compared on: (1) Concrete CrossEntropy method and (2) CrossEntropy with different abstractions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Concrete</th>
<th>Abs1 Log-3Val</th>
<th>Abs2 Log-Binary</th>
<th>Abs3 Full-3Val</th>
<th>Abs4 Full-Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArraySum 10</td>
<td>89(10)</td>
<td>49(3)</td>
<td>54(3)</td>
<td>54(3)</td>
<td>42(2)</td>
</tr>
<tr>
<td>ArraySum 100</td>
<td>44089(67)</td>
<td>5044(4)</td>
<td>4529(4)</td>
<td>5891(4)</td>
<td>5023(5)</td>
</tr>
<tr>
<td>ArraySum 150</td>
<td>33389(100)</td>
<td>10485(5)</td>
<td>8646(5)</td>
<td>6544(4)</td>
<td>4623(4)</td>
</tr>
<tr>
<td>ArraySum 250</td>
<td>81365(165)</td>
<td>11886(4)</td>
<td>6971(4)</td>
<td>23494(5)</td>
<td>13363(5)</td>
</tr>
<tr>
<td>ArraySum 1000</td>
<td>-</td>
<td>54968(5)</td>
<td>18499(4)</td>
<td>51959(5)</td>
<td>18198(4)</td>
</tr>
<tr>
<td>ArrayDiff 100</td>
<td>32183(67)</td>
<td>4400(4)</td>
<td>4594(4)</td>
<td>6488(4)</td>
<td>5940(5)</td>
</tr>
<tr>
<td>ArrayDiff 150</td>
<td>60940(100)</td>
<td>8588(5)</td>
<td>8324(5)</td>
<td>11652(5)</td>
<td>8024(5)</td>
</tr>
<tr>
<td>TreeArraySum 128</td>
<td>12688(128)</td>
<td>12694(26)</td>
<td>21286(25)</td>
<td>16603(24)</td>
<td>17964(27)</td>
</tr>
<tr>
<td>TreeArraySum 128</td>
<td>14767(113)</td>
<td>9663(23)</td>
<td>12405(24)</td>
<td>9593(24)</td>
<td>16266(24)</td>
</tr>
<tr>
<td>PrefixSum 128</td>
<td>17510(74)</td>
<td>26120(79)</td>
<td>36526(80)</td>
<td>25776(81)</td>
<td>24729(78)</td>
</tr>
<tr>
<td>PrefixSum 128</td>
<td>29479(221)</td>
<td>26026(77)</td>
<td>12596(76)</td>
<td>25916(81)</td>
<td>36521(80)</td>
</tr>
<tr>
<td>S + S − /S − 100</td>
<td>39929(67)</td>
<td>13969(5)</td>
<td>4493(5)</td>
<td>5002(5)</td>
<td>9530(5)</td>
</tr>
<tr>
<td>S + S − /S − 150</td>
<td>67850(96)</td>
<td>16966(5)</td>
<td>23119(5)</td>
<td>28871(5)</td>
<td>20603(5)</td>
</tr>
<tr>
<td>Kahan 150</td>
<td>61006(112)</td>
<td>4618(4)</td>
<td>3448(4)</td>
<td>4773(4)</td>
<td>3651(4)</td>
</tr>
<tr>
<td>Kahan 500</td>
<td>56644(370)</td>
<td>20101(4)</td>
<td>15096(4)</td>
<td>24870(4)</td>
<td>17973(4)</td>
</tr>
</tbody>
</table>

/S− algorithm was executed with 100 and with 150 threads, and Kahan algorithm was executed with 150 and with 500 threads.

**Random testing.** Table 5.3 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 5000 times on each input array (the time in the table refers to the first 100 sampled arrays).

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 $[0.01, 111.0]$; 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 *ArraySum 100* 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 hypothesis that the rounding/truncation errors are too rare to be found by randomized testing, even if the inputs are chosen at random.
Table 5.3: Results of random testing. Random testing failed to find even a single bug despite often running for more than 24 hours.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input size</th>
<th>Output Precision</th>
<th>Uniform Distr.</th>
<th>Binomial Distr.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Time</td>
<td>Number Sampled</td>
</tr>
<tr>
<td>Array Sum</td>
<td>100</td>
<td>5</td>
<td>13.25h</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>12.75h</td>
<td>100</td>
<td>13h</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>5</td>
<td>15.5h</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>15.75h</td>
<td>100</td>
<td>16h</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>5</td>
<td>18.5h</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>18.25h</td>
<td>100</td>
<td>18.5h</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>5</td>
<td>22h</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>22.5h</td>
<td>100</td>
<td>23.5h</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>5</td>
<td>29.75h</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>29.75h</td>
<td>100</td>
<td>30.5h</td>
</tr>
<tr>
<td>Prefix Sum</td>
<td>128</td>
<td>5</td>
<td>33.5h</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>33.5h</td>
<td>100</td>
<td>34h</td>
</tr>
<tr>
<td>Tree Schema Sum</td>
<td>128</td>
<td>5</td>
<td>34h</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>34.5h</td>
<td>100</td>
<td>32h</td>
</tr>
</tbody>
</table>

For each benchmark, the table lists:

1. input size;
2. the required precision of the output, in terms of the number of digits;
3. time and number of sampled arrays for discrete uniform distribution;
4. time and number of sampled arrays for binomial distribution.
Chapter 6

Related Work

In this chapter we survey the related work in the fields of numerical stability and testing. First we introduce the notation of numerical stability, explain how it is related to concurrent computations, show the causes of errors in numerical computations and demonstrate known approaches for preventing these errors. Then, we survey known testing approaches for faults detection and explain the problem to apply these approaches for numerical instabilities problems.

6.1 Numerical Stability

The notion of numerical stability is an important criterion for numerical algorithms. Regarding concurrent computations, 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) [35]. The algorithm’s result is inaccurate once the error exceeded the predefined bound and we refer an algorithm as stable if it is always accurate.

A finite precision computation can cause an accurate algorithm or method to be inaccurate. Once the input, the output or the intermediate results are given in a finite precision in a floating point format, the algorithm’s output can be inaccurate due to round-off errors or cancellation errors and can lead to disastrous results (see Introduction, Chapter 1). Hence, the stability of a numerical algorithm is an important factor in its analysis.

A round-off error is a cause of rounding or truncation of a value during the computation. In practice, we need to reduce the number of digits of the interme-
diate result to fit the unit roundoff of the machine (or the machine epsilon), which depends on the number of digits we can have. The implementation of the unit roundoff is done by rounding or truncation of the intermediate result. The unit roundoff is the upper bound of the relative error of an arithmetic operation. The relative error is: \((a' - a)/a\) (if \(a \neq 0\)), given that \(a\) is the accurate value and \(a'\) is the value in finite precision presentation. The relative error bounds are: (1) in rounding is \(0.5 \cdot 10^{1-t}\) and we say we have \(t\) correct decimal, and (2) in chopping is \(10^{1-t}\) and we say we have \(t\) significant digits [25]. A cancellation error occurs when somewhere in the computation, the relative error of the intermediate result is considerably larger than the absolute error; this error causes a loss of information. In summation methods, subtraction of two nearly equal numbers can cause a cancellation error [25].

For a given accurate value \(a\) and its value in finite precision presentation \(a'\), one can obtain the relative error between them. However, for the error analysis of arithmetic operations, one can only estimate the error bound. Accordingly, the accuracy comparison between different numerical methods is done by analyzing the error bounds [8, 9, 34, 59]. The smaller the error bound is, the more likely the algorithm is stable and accurate. Improved numerical algorithms are constructed based on this assumption, so that they minimize the error bound of the currently known best algorithm. For example, Kahan summation method attempts to reduce the error through adaptive correction of the result whereas Fast Fourier Transform (FFT) by reordering of its arithmetic operations by using pairwise summation, both algorithms achieve a small error bound and are considered to be better methods (See Section 5.3.1 and Chapter 1 respectively for discussion about these algorithms). However, in his discussion on summation methods, Higham [34] concluded that no method is uniformly better than the others and suggested guidelines to help choosing the best method for the algorithm needs. There is no one “winning” method because the actual error for some cases can be much smaller than the known error bounds of the method and moreover, can be even smaller than the actual error of other methods with tighter error bounds.

With regards to parallel computations, QuickStep [48] tries to construct stable algorithms by parallelizing a stable sequential version and adding correction to the algorithm in order to assure that the error bounds do not exceed the known error bounds of the sequential algorithm.
6.2 Existing Approaches for Faults Detection

In general, concurrent programs present a challenge for all testing approaches due 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. Other testing techniques, which combine systematic exploration, program analysis techniques and/or heuristics for guiding a dynamic exploration, can fail to find rare numerical instabilities in the code due to lack of space or time. 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. In this section, we discuss the different techniques and explain the reason they cannot fit for testing of numerical programs.

Random testing in general tends to miss rare bugs, focusing instead of uniform coverage of the state space of the program [27, 68]. As our experiments show (see Chapter 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.

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 [16, 38, 37, 54], or introduce a random interference (noise) in the scheduler with the goal of creating uncommon behaviors (see [68] or ConTest [27]). 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. Active testing (Calfuzzer) is a random testing approach that controls the scheduler explicitly [37, 38, 54, 64]. This method uses both program analysis techniques and random testing technique for respectively predicting and finding concurrency bugs. Through program analysis techniques, the tool obtains a set of statements in the program that can be involved in the creation of a concurrency bug. By using this set, the tool explicitly controls the scheduler order to uncover a real bug. A rare numerical instability in the code is a result of many round-off errors in the computation. Accordingly, the random testing can miss the bug since the set of operations is still very large. Noise makers as ConTest aim to create uncommon behaviors by introducing noise to the scheduler [27, 63, 68].
For instance, ConTest [27] is a testing tool in which different and unusual threads-interleavings are created by seeding instructions to the scheduler, such as sleep(), yield() and priority() functions, randomly or with a coverage based-decision. This method successfully finds numerical errors for computations with several threads. However, for computations with hundreds of threads, the error becomes very rare and as there is no correlation between uncommon behavior of the code and numerical instabilities in the code, this approach tends to miss bugs.

The systematic approach for generating test cases for all feasible paths of a program using concolic techniques (see, for example, [30, 65]) would not scale to numerical algorithms in our examples. Concolic testing is a systematic and automatic approach for generating test cases for all feasible paths of a program by combining a symbolic and concrete execution in order to find a bug. CUTE, DART (C programs) and jCUTE (Java programs) [30, 65, 66] are implementations of this approach. The searching strategy is based on a depth first search (DFS) of test cases with different schedules and data. After executing a test case, the algorithm backtracks and generates a new test case by solving symbolic constraints. The symbolic constraints are a conjunction of all constraints over the symbolic values on each branch point. Yet rare bugs can remain in the code undetected (when avoiding full coverage of the code) as mentioned in [65]. This can happen when the solver of the constructed constraints is not powerful enough, which is most likely the case for constraints on many floating points numbers. With no such solver, some bugs will not be detected, and redundant execution will be extracted since the solver is being used for avoiding this redundancy.

Methods in which heuristics or program analysis techniques are used to guide testing [16, 20, 37, 53, 65], 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 [33](i.e., stress testing). In this case, not finding errors cannot guarantee the correctness of all 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 [51]). CHESS is a systematic and deterministic testing tool, based on a stateless model checker [51, 52, 53] (A stateless model checker explores the state space of a program without storing states of it during the search; for model-checkers, it was first
proposed in VeriSoft [29]). CHESS avoids redundant executions and searches the state space using iterative context bounding (i.e., first exploring paths with fewer context switches). This searching strategy reduces the required time for detecting a bug once the number of iterations required for achieving full coverage of a code is known or small. 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 thesis is the recent research on using cross-entropy for testing and replay of concurrent programs [20, 21]. The implementation described in [20] is the basis for our implementation of the cross-entropy based testing of numerical instability. The algorithm in [20] 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 pre-defined 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 [21], is complementary to our algorithm and can be used to verify that the bug was fixed (by trying to replay a buggy execution). Other alternative methods for optimization problems can be used instead of the cross-entropy method as simulated annealing [1, 23, 28, 60] and genetic algorithms [31]. Simulated annealing (SA) is a generic probabilistic method to a global optimization, on which a good solution is a good approximation to the global optimum of a given function $f$ in a large search space. Due to time limitations, one can prefer using simulated annealing over exhaustive search, for searching the global optimum of $f$, given it is enough to find a good acceptable solution instead of the best possible solution. SA is an iterative approach bounded in time, on which an initial state of the system is given and in each step it searches for neighbors of the current state, saves the currently best known neighbor and chooses randomly whether to move to the current state’s neighbor or not. In the end, it returns the currently best known neighbor. Genetic algorithm (GA) is a search heuristic to optimization problems. It is inspired by natural evolution and belongs to a larger class of evolutionary algorithms (EA). GA approach starts by generating randomly an initial population, which typically contains several hundreds or thousands of possible individual solutions, and then iteratively select the best individuals to breed a new (better) generation and use these to create a new generation of individuals by crossover and/or mutation (called ge-
netic operators); the algorithm terminates once one or some of the conditions hold: the best solution (or a good enough solution) was found, there is no more improvement through the generations (i.e., the algorithm reached a plateau in the search), already created a fixed number of generation, reached the time or space limitation or by user decision.
Chapter 7

Conclusions and Future Work

In this thesis we present a new approach for detection of rare numerical instabilities in concurrent computations, implemented our approach in a tool ACE (Abstract Cross-Entropy) and showed its effectiveness in finding rare numerical stability errors (in Chapter 5). Our approach successfully finds rare numerical errors in numerical programs even when the error is a result of many instructions and involves many context switches between the program’s threads. This result is important because it shows the possibility to test numerical algorithms by examining executions of their implementation without the need to verify the correctness of each instruction separately (for all values in the domain) and without the need to perform error analysis.

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 [34]. For example, pairwise summation and compensated summation methods tend to work better for large arrays, and recursive summation method with decreasing order of summation is recommended for sums with heavy cancellation. Not surprisingly, we found numerical instabilities in all summation methods that we checked. At the same time, as we demonstrated in this thesis, existing testing methods are ill-suited for finding these types of errors, and hence they remain undetected even in mission-critical applications.

Most of the errors we found resulted from few specific scenarios (it is also the reason for the difficulty to detect them), which can be avoided by rewriting the code or by introducing several synchronization blocks, effectively blocking this particular scenario. In mission-critical applications, the addition of synchroniza-
tion resulting from our error analysis might be a small price to pay for stability.

Currently, ACE successfully detects numerical errors resulted from two arithmetic operations: “+” and “-”. This result encourages us to expand the ability of ACE to cope with larger and complex algorithms, and thus we shall think and examine how to add all arithmetic operations; adding all arithmetic operations requires a construction of a set of performance functions, which we yet have. Our goal in the future is to achieve an improved framework for numerical instabilities detection of all arithmetic operations. In addition, the future functionality of ACE shall have ability to suggest corrections to these errors and to refine the system once the coarsest abstraction fails to converge. Such a framework shall be able to detect and correct rare numerical instabilities in short time, which can save life, money and time.

In the following sections, we describe the future work in more details.

7.1 Refine Abstraction on the Fly

ACE contains an implementation of four different sets of abstractions. Once the cross-entropy method based testing with the coarsest set of abstractions fails to converge, one can find the bug by using a different set of abstractions, some of which are more refined than before. Although this technique successfully finds rare numerical instabilities in a very short time, it ignores the information that already been gathered by ACE. For larger programs, using this information can significantly reduce the total time that required for finding rare numerical instabilities. Even for our benchmarks with 500 and 1000 threads the additional time is significant (over 48 hours for all the fruitless settings).

We suggest the following schematic stages for the future implementation of on-the-fly refinement method.

Detection Stage. First, we need to identify which entries in the table are blocking us from converging to the global maximum of the performance function and find a bug. Roughly speaking, ACE will fail to converge once executions that are close to an erroneous execution, and executions that are far from an erroneous execution, are mapped to the same abstract path. Here two things can happen: either the probability of choosing this abstract path remains low, or the probability of choosing this abstract path increases but most of the concrete executions of this abstract path are without any numerical error (as it is the case for the second transition system illustrated in Figure 2.2 in Chapter 2). Of course not all of these entries should be
refined. In our future work, we will think which entries we shall refine by analyzing which entries cause ACE (with the current set of abstractions) to fail finding the bug.

**Refinement Stage.** After detecting these entries in the table, the refinement is done by splitting these entries. Using the “best” executions of previous iterations of ACE, for each entry we create two or more entries that use one or more refined abstractions. For example, if an entry represents a certain state with counter abstraction of 0-1 and under full data abstraction, we can create several refined states by using counter abstraction of 0-1-2 and log data abstraction. The probabilities on outgoing edges of these states are updated in the table by using the “best” samples of previous iterations. For this stage, we need to think which set of abstractions to take each time we refine our abstraction, as the set of refined abstractions is not fixed nor known in advance.

### 7.2 Performance Functions of Elementary Arithmetic

The ability of ACE to cope with numerical code of complex algorithms will be expanded. Commonly, the detection of a numerical error is done by examine a single arithmetic operation. The success of CE-method to find numerical instability that results from a long sequence of arithmetic operations encourage us to add in the future a wider set of performance functions to search for numerical instabilities in mission-critical applications, testing any sequence of arithmetic operations.

In order to be able to detect numerical instabilities for all types of numerical problems, we need to define a set of performance functions, which contains all arithmetic operations. The initial work of the CE-method based testing [20] listed several common patterns in programs, often associated with bugs, for which there exist natural metrics. In this thesis, we added a basic set of performance functions for common numerical problems that was not exist or described before for detecting instabilities in a program (instead of in a single instruction). We will add to the basic set also the functions $S^*$, $S^/$, $S^{sqrt}$, etc., and thus be able to apply ACE to larger set of numerical problems. Adding these functions shall be easy since the error is well described in the literature (see for example [25]). For instance, the error of division and multiplication operations is an estimation of the relative error; we can formalize performance functions that rank higher steps with larger relative error (instead of the absolute error as done in $S^+$ and $S^-$).

From this set of functions, one shall be able to construct a performance function
for finding rare numerical instabilities in a given numerical code. In this thesis, we show how to construct the performance function $S^{+-}$ from the set of performance functions we have already defined $\{S^+, S^-\}$.

### 7.3 Removing Detected Numerical Instabilities

The basic purpose of our tool ACE is the ability to produce the bug quickly and easily. Once ACE is able to produce an instance of a bug, we use the latest probability table to reproduce many instances of the bug without changing the probability table anymore. Hence, the stage of learning the bug pattern and reproducing faulty executions is over. The next stages are related to fault localization and correction.

We describe in highlights the main stages in fault detection and correction methods.

In order to fix the bug, we first produce two sets of executions. The first set contains many instances with a numerical error, and the second set contains many instances without any numerical error. Yet, the instances of the second set are created in such a way that they are close to have a numerical error. We produce these instances via the latest probability table we created through the process described in this thesis. We use these sets to localized numerical instabilities in the code. After localized numerical instabilities, ACE shall be able to suggest a correction. We can fix the code by adding several corrections as avoiding certain context switches, or using different data types (as changing floats into doubles). After adding a correction, we use the latest probability table in order to check if we are able to reproduce the bug. Here we describe in more details the future work in this direction, adding three stages to ACE: (1) fault localization, (2) inferring a fix, and (3) evaluating a solution. A schematic flow is given in Figure 7.1.

**Searching the execution space using cross-entropy with abstraction (ACE).**

ACE searches numerical instabilities using a performance function built from a set of performance functions, which are given as part of the input and are smooth heuristic functions. The performance function indicates how close an execution is to the pattern we are looking for. Currently, we are using $S^{+-}$ to indicate this, where $S^{+-}$ includes ranking a steps via the set $\{S^+, S^-\}$ (see our description in Section 4.1). For more examples of smooth heuristic functions see [20, 21].

**Producing a large number of faulty executions via ACE.** The cross-entropy (CE) method is an iterative approach based on minimizing distance between samples. Here, we sample through dynamic testing and bias the search using a predefined
• Input:
  – A concurrent Java program that solves a numerical problem, possibly containing rare numerical instabilities in the code
  – A set of performance functions for all arithmetic operations

• Output:
  – A probability table, a localized explanation of numerical instabilities in the code, and sometimes a suggested fix

• Schematic Algorithm:
  – Execute the program under a random scheduler
  – Produce a large number of faulty executions and close-to-faulty executions
  – Localize the faults
  – Infer a fix
  – Attempt to reproduce a faulty execution after removing all numerical instabilities in the code, using ACE to try and reproduce it

Figure 7.1: A high-level description of our future additional functionality to ACE

performance function (in this thesis, we use $S^+$). ACE extracts many samples using the latest probability table. Some of the instances are unstable executions of the code. Thus, we are able to create a set of unstable executions and a set of stable executions that are close to being unstable. Note that the relation between the probability table and the performance function is the reason why we are able to create both sets. The latest probability table shall create both sets, and we can classify them by using our performance function.

This functionality is based on the ability of ACE to produce many instances in short time; this requires the probability table to have some characteristics that we need to study.

Fault Localization. Using a metric distance function (ACE uses a performance function) for ranking each execution, yields two sets of executions: (i) faulty executions; (ii) non-faulty executions that are close to buggy ones (according to the metric distance). Using both sets, we perform causal analysis (e.g., [10, 22, 32]) with the following schematic idea. All executions (of both sets) are close under the
metric distance and thus share common sub-paths. Some sub-paths are common to both sets, while other paths exist only in one of them. We localize the fault by using sub-paths exist only in the faulty executions, while comparing them to close non-faulty executions. This shall give us the fault explanation of the current bug. For instance, the executions of both sets contain a sequence of operations with data of the same scale (as adding 10 times numbers with 7 digits after the decimal point). This sub-path probably contributes significantly to the numerical error of the final result.

**Inferring a fix.** Based on the results of the causal analysis, our method can suggest one of the following: (i) No correction, (ii) correction by reducing the number of threads, (iii) correction by adding a conditional context switch, (iv) correction by adding a probabilistic context switch, and (v) correction by adding more digits to the presentation of some of the intermediate results. On this stage the most important thing for us is to understand in our future work which corrections we shall suggest and when.

**Evaluating a solution.** After adding the suggested fix, one can check if the correction of the code indeed eliminates the numerical instability in the code, by using the bug pattern we obtained before and trying to reproduce the bug again via ACE.
Bibliography


[40] J.M. Keith and D.P. Kroese. Rare event simulation and combinatorial optimization using cross entropy: sequence alignment by rare event simulation.


Chapter 8

Appendix

8.1 Upper Bound Probability of Sampling (at least) Two Numbers From Interval Edges

In this section, we set the bounds on the probability of sampling arrays with certain characteristics. The arrays we are sampling, are arrays of size \(n\) from a discrete distribution on the discrete interval \([a, b]\) with at least two values from the interval edges; \(a, b \in \mathbb{D}\) and \(\mathbb{D}\) is a domain of decimal numbers with a bounded number of digits. We define the interval edges to be any value below \(\alpha\) or above \(\beta\), where \(\alpha < \beta\) and \(\alpha, \beta \in [a, b]\). Since \(\alpha\) and \(\beta\) are in the interval edges, we can assume that \(\alpha < E[x]\) and \(\beta > E[x]\).

**Upper Bound Probability to Sample a Number from Interval Edges.** Using Chebychev inequality [19, 47],

\[
Pr[|x - E[x]| \geq C] \leq (V[x] \cdot C^{-2}),
\]

where \(x\) is a random variable and \(C\) is a positive real number \((0 < C \in \mathbb{R})\), we get bounds of the probabilities to sample a number from the interval edges. The probability to sample a number equal or below \(\alpha\), is given by using Chebychev inequality with \(C = c_1 = |E[x] - \alpha|\), and \(\alpha < E[x]\) such that:

\[
Pr[x \leq \alpha] \leq Pr[|x - E[x]| \geq c_1] \leq (V[x] \cdot c_1^{-2}).
\]

**Proof 1** We show now, how to find the probability on which a random variable \(x\)
is less or equal to \( \alpha \); i.e., finding \( \Pr[x \leq \alpha] \). From changing the sign of \( x \) and \( \alpha \), we get that:

\[
\Pr[x \leq \alpha] = \Pr[-x \geq -\alpha],
\]

(8.3)

then we add to both sides \( E[x] \), thus

\[
\Pr[-x \geq -\alpha] = \Pr[(E[x] - x) \geq (E[x] - \alpha)],
\]

(8.4)

since \( \alpha < E[x] \), then \((E[x] - \alpha) > 0\), and thus it holds that \((E[x] - \alpha) = |E[x] - \alpha|\), and we get,

\[
\Pr[(E[x] - x) \geq (E[x] - \alpha)] = \Pr[(E[x] - x) \geq |E[x] - \alpha|].
\]

(8.5)

We define that \( c_1 = (|E[x] - \alpha|) \), thus

\[
\Pr[(E[x] - x) \geq |E[x] - \alpha|] = \Pr[(E[x] - x) \geq c_1].
\]

(8.6)

The inequality \(|E[x] - x| \geq c_1\) holds, if and only if \((E[x] - x) \geq c_1\) or \((x - E[x]) \geq c_1\), that is

\[
\Pr[(E[x] - x) \geq c_1] + \Pr[(x - E[x]) \geq c_1] = \Pr[|E[x] - x| \geq c_1],
\]

(8.7)

and therefore

\[
\Pr[(E[x] - x) \geq c_1] \leq \Pr[|E[x] - x| \geq c_1] = \Pr[|x - E[x]| \geq c_1].
\]

(8.8)

By using Chebychev with \( C = c_1 \), we get that

\[
\Pr[|x - E[x]| \geq c_1] \leq (V[x] \cdot c_1^{-2}),
\]

(8.9)

and thus the probability that a random variable \( x \) is at most \( \alpha \) is bounded by \((V[x] \cdot c_1^{-2})\), that is:

\[
\Pr[x \leq \alpha] \leq (V[x] \cdot c_1^{-2}).
\]

(8.10)

\( \Box \)

The probability to sample a number equal or above \( \beta \), is given by using Chebychev inequality with \( C = c_2 = (|\beta - E[x]|) \), and \( \beta > E[x] \) such that: \( \Pr[x \leq \beta] \leq \Pr[|x - E[x]| \geq c_2] \leq (V[x] \cdot c_2^{-2}) \).
Proof 2 We show now, how to find the probability on which a random variable \( x \) is great or equal to \( \beta \); i.e., finding \( \Pr[x \geq \beta] \).

We add to both sides \(-E[x]\), thus

\[
\Pr[x \geq \beta] = \Pr[(x - E[x]) \geq (\beta - E[x])],
\]

since \( \beta > E[x] \), then \((\beta - E[x]) > 0\), and thus it holds that \((\beta - E[x]) = |\beta - E[x]|\), and we get,

\[
\Pr[(x - E[x]) \geq (\beta - E[x])] = \Pr[(x - E[x]) \geq |\beta - E[x]|]. \tag{8.11}
\]

We define that \(c_2 = (|\beta - E[x]|)\), thus

\[
\Pr[(x - E[x]) \geq |\beta - E[x]|] = \Pr[(x - E[x]) \geq c_2]. \tag{8.12}
\]

The inequality \(|x - E[x]| \geq c_2\) holds, if and only if \((E[x] - x) \geq c_2\) or \((x - E[x]) \geq c_2\) holds, that is

\[
\Pr[(E[x] - x) \geq c_2] + \Pr[(x - E[x]) \geq c_2] = \Pr[|x - E[x]| \geq c_2], \tag{8.13}
\]

and therefore

\[
\Pr[(x - E[x]) \geq c_2] \leq \Pr[|x - E[x]| \geq c_2]. \tag{8.14}
\]

By using Chebychev with \(C = c_2\), we get that

\[
\Pr[|x - E[x]| \geq c_2] \leq (V[x] \cdot c_2^{-2}), \tag{8.15}
\]

and thus the probability that a random variable \( x \) is at least \( \beta \) is bounded by \((V[x] \cdot c_2^{-2})\), that is:

\[
\Pr[x \geq \beta] \leq (V[x] \cdot c_2^{-2}). \tag{8.16}
\]

\(\square\)

For a random variable \( x \), and constants \(c_1 = |E[x] - \alpha|\) and \(c_2 = (|\beta - E[x]|)\), where \(\alpha\) and \(\beta\) are defined as before, the unify upper bound of sampling a number from the interval edges is at most \((V[x] \cdot (c_1^{-2} + c_2^{-2}))\); since \(\Pr[x \leq \alpha] + \Pr[x \geq \beta] \leq (V[x] \cdot c_1^{-2}) + (V[x] \cdot c_2^{-2}) = (V[x] \cdot (c_1^{-2} + c_2^{-2}))\). We use the variance and the expected value of the distribution we are sampling from. We represent finite size decimal numbers by creating unique mapping from the domain we are sampling.
from to the neutral numbers, and thus able to use discrete distributions to sample from discrete domains of decimal numbers with bounded size of digits.

**Upper Bound Probability to Sample at least Two Numbers from Interval Edges.** We define a Bernoulli experiment (trial) with two possible outcomes of success and failure as the events of sampling from the distribution edges and its complementary event. A success event probability $p$ is at most $(V[x] \cdot (c_1^{-2} + c_2^{-2}))$ and a failure event probability $q$ is at least $1 - (V[x] \cdot (c_1^{-2} + c_2^{-2}))$. The probability of having at least $k$ successes out of $n$ Bernoulli experiments (where $0 \leq k \leq n$), is given by the following equation,

$$Pr[y \geq k] \leq \binom{n}{k} \cdot p^k,$$

for a given sequence of $n$ Bernoulli experiments, with a success probability $p$, when $y$ is a random variable which indicates the total number of successes [24]. Given that $p \leq (V[x] \cdot (c_1^{-2} + c_2^{-2}))$, we get that

$$Pr[y \geq k] \leq \binom{n}{k} \cdot (V[x] \cdot (c_1^{-2} + c_2^{-2}))^k. \quad (8.19)$$

By taking $k = 2$, for a given discrete probability distribution of a bounded domain $[a, b]$ with $\alpha, \beta, \mathbb{D}, c_1$ and $c_2$ as defined before, we get that the probability of sampling at least two items (out of $n = 150$) from the interval edges is

$$Pr[y \geq 2] \leq \binom{n}{2} \cdot (V[x] \cdot (c_1^{-2} + c_2^{-2}))^2, \quad (8.20)$$

that is,

$$Pr[y \geq 2] \leq \frac{n \cdot (n - 1)}{2} \cdot (V[x] \cdot (c_1^{-2} + c_2^{-2}))^2. \quad (8.21)$$

**Upper Bound Probability to Sample at least Two Numbers from Interval Edges of a given Binomial Distribution.** We give an example of the usage of this bound for ArraySum 150 with a binomial distribution-(11099,0.5) of the domain $[1, 11100]$; each item $x \in [1, 11100]$ is considered as $\frac{x}{100}$ in the input array of ArraySum 150. For a given $\alpha, \beta \in [1, 11100]$, thus $\alpha$ and $\beta$ are far from $E[x]$, and $\alpha < E[x]$ and $\beta > E[x]$, the bound of sampling at least two items (out of $n = 150$) from the
edges of the interval $[1, 11100]$ is,

$$Pr[y \geq 2] \leq \frac{150.149}{2} \cdot (2774.75 \cdot (c_1^{-2} + c_2^{-2}))^2 = 11175 \cdot (2774.75)^2 \cdot ((5549.5 - \alpha)^{-2} + (\beta - 5549.5)^{-2})^2,$$

where $p_{BD} = 0.5$ and $1 - p_{BD} = 0.5$, thus $E[x] = 11099 \cdot 0.5 = 5549.5$ and $V[x] = 11099 \cdot 0.5 \cdot 0.5 = 2774.75$. For instance we take $\alpha = 1001$ and $\beta = 10100$, thus the interval edges is 18% of the total number of trails of this binomial distribution, we get a very low probability to sample an array with two or more numbers from the interval edges, which is 0.0804%. This only promises us to sample candidates of arrays that can have an erroneous result due to different order of operations.
Abstract Cross Entropy (ACE) is a method for evaluating the consistency of binary classifiers. It is used to determine the consistency of a classifier with a certain threshold. This method can be used to detect errors in the implementation of algorithms, which is important for the construction of accurate and reliable systems.

The core of the method is the calculation of cross-entropy between two sets of data. The cross-entropy is calculated as the difference between the true and predicted values. The method is based on the idea that the cross-entropy should be minimum for a consistent classifier.

The method is applied to various systems and has shown good results in detecting errors and improving the consistency of the classifiers.

The main advantage of the ACE method is its ability to detect errors in the implementation of algorithms, which is crucial for the construction of accurate and reliable systems. It is especially useful in the field of machine learning, where the consistency of the classifiers is crucial for the accuracy of the results.
The accuracy of floating point summation

...
The main contribution of this thesis is the development of a parallel algorithm that can efficiently handle large datasets in various domains such as statistics, biology, chemistry, etc.

The algorithm was implemented in a variety of threads, including those written in the languages of choice for each domain. The results showed that the parallel algorithm was significantly faster than the sequential versions.

In addition to the algorithm itself, a testing framework was developed to evaluate the performance of the algorithm. This framework allowed for the testing of the algorithm in various settings, including those where the data was not uniformly distributed.

The framework was used to test the algorithm in a variety of environments, including those with large numbers of threads. The results showed that the algorithm was able to handle these environments with ease.

Overall, the thesis demonstrates the potential of parallel algorithms in handling large datasets efficiently and effectively.
המחקר שעשה בהנחיית פרופ' ירח ברקוביץ' למדעי המחשב ولم זכה חנה ערן במדליה.

לאימוט חומרים ותוכנה, מעבדות המחבר א. י. א. מ. ו. ר. קרמל, חיפה.

אני מודד לテכניט על התמיכתו הכסייתנו והנדנה בהלולעה.

מזהות שגיאות ייעובות מספרית
נדירות בשישות מקבילים

תיבורי על מחקרה

لسם מיליני חלקי של הדרישות לקבלת תואר
מגיסטר למידע במדעי המחשב

קרין אבנ

חון אלנט חכמוני – מרכז טכנולוגי לישראל
אביר היענים
当他
2013
משייאת שגיאות ייצוב מספרית
נדירות בחישובים מקבילים

קריני אבו