Expanding the Horizons of Finite-Precision Analysis

A dissertation submitted towards the degree

Doctor of Engineering (Dr.-Ing.)

of the Faculty of Mathematics and Computer Science
of Saarland University

by

Debasmita Lohar

Saarbrücken, 2023

Date of Colloquium:	27.03.2024
Dean of the Faculty:	Univ.-Prof. Dr. Jürgen Steimle
Chair of the Committee:	Prof. Dr. Joël Ouaknine
Reviewers:	Prof. Dr. Eva Darulova
	Prof. Dr. Anastasia Volkova
	Prof. Dr. Bernd Finkbeiner
	Prof. Dr. Santosh Nagarakatte
Academic Assistant:	Dr. Poulami Das

Abstract

Numerical programs are ubiquitous across many domains, including embedded systems, scientific computing, and machine learning. These programs utilize finite precision computations that inevitably introduce numerical uncertainties. These uncertainties are usually a combination of input uncertainties arising from, e.g., the noisy sensors of the underlying hardware and finite-precision (roundoff) errors that can occur at every arithmetic operation. While these errors are individually small, they can propagate through an application unintuitively and can make the final results meaningless. Thus, it is essential to verify that numerical errors in a program remain acceptably small.

In addition to ensuring accuracy, implementing finite-precision programs on real hardware also necessitates efficiency in terms of resource usage. While precise data types can yield accurate results, they often come at the cost of increased resources, such as execution time and chip area. Thus, optimizing the implementations to strike a balance between accuracy and efficiency for specific hardware applications is crucial.

Several static and dynamic analyses exist to verify and estimate roundoff errors, to test, and to optimize the precision of finite-precision programs. Static analyses typically perform sound worst-case analysis but are primarily designed for straight-line special syntax code, with limited support for conditionals and loops. These limitations also extend to sound optimization techniques. Similarly, the current dynamic analyses are also limited by their inherent inability to provide soundness guarantees and to detect errors in large-scale programs within a reasonable time, primarily due to the complexities associated with finite precision.

In this thesis, we address three limitations of state-of-the-art analysis and optimization techniques, thus expanding the individual capabilities of these analyses for finite-precision programs.

First, we introduce probabilistic analysis for finite-precision programs that goes beyond traditional worst-case analyses by capturing the effects of probabilistic inputs. We explore two important issues in this context. Our first contribution is a probabilistic analysis that takes into account the effects of numerical uncertainties on discrete decisions for both floating-point and fixed-point programs. Secondly, we present the first sound probabilistic error analysis for floating-point programs. The error analysis also handles probabilistic error specifications prevalent in today's approximate hardware. Our evaluation shows that these two analyses can analyze small but exciting embedded and neural network programs from the literature with arithmetic and elementary operations.

Next, we propose a two-phase framework that combines static and dynamic analyses to address their scalability issues for numerical programs. By integrating these approaches, we are able to (conditionally) verify the absence of special floating-point values (NaN and Infinities) and

large roundoff errors in more extensive programs with more than 2K lines of C and C++ code. This framework handles complex programming structures and intricate data representations, thus broadening the scope of numerical program verification.

Finally, we present the first sound mixed-precision fixed-point quantization technique tailored specifically for neural networks. This technique efficiently optimizes the number of bits needed to implement a network while guaranteeing a provided error bound. An extensive evaluation on existing embedded neural controller benchmarks shows that our technique outperforms generic static fixed-point precision tuners. Our technique enables scalable and efficient optimization in terms of machine cycles when compiled to an FPGA, particularly for large-scale neural networks with thousands of parameters.

Zusammenfassung

Numerische Programme sind in vielen Bereichen allgegenwärtig, darunter in eingebetteten Systemen, wissenschaftlichem Rechnen und maschinellem Lernen. Diese Programme verwenden Berechnungen mit endlicher Präzision, die zwangsläufig numerische Unsicherheiten mit sich bringen. Diese Unsicherheiten sind in der Regel eine Kombination aus Eingabeunsicherheiten, die beispielsweise durch verrauschte Sensoren der zugrunde liegenden Hardware entstehen, und Fehlern durch endliche Genauigkeit (Rundungsfehler), die bei jeder Rechenoperation auftreten können. Auch wenn diese Fehler individuell klein sind, können sie sich unintuitiv durch eine Anwendung ausbreiten und dazu führen, dass die Endergebnisse bedeutungslos werden. Daher ist es wichtig zu überprüfen, ob numerische Fehler in einem Programm akzeptabel klein bleiben.

Neben der Sicherstellung der Genauigkeit erfordert die Implementierung von Programmen mit endlicher Präzision auf realer Hardware auch eine effiziente Ressourcennutzung. Während präzise Datentypen genaue Ergebnisse liefern können, gehen sie oft mit höheren Ressourcen wie Ausführungszeit und Chipfläche einher. Daher ist es von entscheidender Bedeutung, die Implementierungen zu optimieren, um ein Gleichgewicht zwischen Genauigkeit und Effizienz für bestimmte Hardwareanwendungen zu finden.

Es gibt mehrere statische und dynamische Analysen, um Rundungsfehler zu überprüfen und abzuschätzen, um die Präzision von Programmen mit endlicher Genauigkeit zu testen und zu optimieren. Statische Analysen führen in der Regel eine Worst-Case-Analyse durch, sind jedoch in erster Linie für geradlinigen Code mit spezieller Syntax konzipiert und bieten nur begrenzte Unterstützung für Bedingungen und Schleifen. Diese Einschränkungen erstrecken sich auch auf korrekte Optimierungstechniken. Ebenso sind die aktuellen dynamischen Analysen auch dadurch eingeschränkt, dass sie nicht in der Lage sind, Garantien zu geben und Fehler in umfangreichen Programmen innerhalb einer angemessenen Zeit zu erkennen, was vor allem auf die mit endlicher Präzision verbundenen Komplexitäten zurückzuführen ist.

In dieser Arbeit befassen wir uns mit drei Einschränkungen moderner Analyse- und Optimierungstechniken und erweitern die individuellen Möglichkeiten dieser Analysen für Programme mit endlicher Präzision.

Zunächst führen wir eine probabilistische Analyse für Programme mit endlicher Genauigkeit ein, die über herkömmliche Worst-Case-Analysen hinausgeht, indem sie die Auswirkungen probabilistischer Eingaben erfasst. Wir untersuchen in diesem Zusammenhang zwei wichtige Themen. Unser erster Beitrag ist eine probabilistische Analyse, die die Auswirkungen numerischer Unsicherheiten auf diskrete Entscheidungen sowohl für Gleitkomma- als auch für Festkommaprogramme berücksichtigt. Zweitens präsentieren wir die erste fundierte probabilistische Fehleranalyse für Gleitkommaprogramme. Die Fehleranalyse berücksichtigt auch probabilistische Fehlerspezifikationen, die in heutiger Approximate Hardware vorherrschen.

Unsere Auswertung zeigt, dass diese beiden Analysen kleine, aber interessante eingebettete und neuronale Netzwerkprogramme aus der Literatur mit arithmetischen und elementaren Operationen analysieren können.

Als nächstes schlagen wir ein zweiphasiges Framework vor, das statische und dynamische Analysen kombiniert, um deren Skalierbarkeitsprobleme für numerische Programme anzugehen. Durch die Integration dieser Ansätze können wir (bedingt) das Fehlen spezieller Gleitkommawerte (NaN und Unendlichkeiten) und großer Rundungsfehler in umfangreicheren Programmen mit mehr als 2K Zeilen C- und C++-Code überprüfen. Dieses Framework verarbeitet komplexe Programmierstrukturen und komplizierte Datendarstellungen und erweitert so den Umfang der numerischen Programmverifizierung.

Abschließend stellen wir die erste korrekte Festkomma-Quantisierungstechnik mit gemischter Präzision vor, die speziell auf neuronale Netze zugeschnitten ist. Diese Technik optimiert effizient die Anzahl der zur Implementierung eines Netzwerks erforderlichen Bits und garantiert gleichzeitig eine gegebene Fehlergrenze. Eine umfassende Auswertung bestehender Benchmarks für eingebettete neuronale Controller zeigt, dass unsere Technik generische statische Festkomma-Präzisionstuner übertrifft. Unsere Technik ermöglicht eine skalierbare und effiziente Optimierung in Bezug auf Maschinenzyklen, wenn sie in einem FPGA kompiliert wird, insbesondere für große neuronale Netze mit Tausenden von Parametern.

Acknowledgements

At the heart of my Ph.D. journey stands my advisor, Eva Darulova. Her insights into research topics, belief in my abilities, continuous encouragement, and guidance on various academic as well as non-academic aspects, from maintaining a work-life balance to managing time, have profoundly shaped my academic and personal growth. She is more than an advisor and mentor; she is a role model whose influence will continue to resonate throughout my career.

I also had the privilege of collaborating with some truly wonderful individuals and exceptional researchers throughout my Ph.D. Among them, I want to extend special thanks to Maria Christakis and Anastasia Volkova. Their support, encouragement, and warmth, especially during the most challenging moments, were invaluable.

The AVA research group, including Heiko, Anastasia, and Rosa, has been an instrumental part of my journey. Heiko's help, from translating German letters to assisting with my move to new apartments, made the process of transitioning to a new country and culture feel seamless. Our lunchtime conversations were a cherished part of my pre-corona days. Anastasia's companionship and shared Ph.D. experiences provided comfort in challenging times. Additionally, she took the time to read through some of my work, offering helpful feedback, especially when I was stuck in my perspective and could not see beyond it.

I extend my gratitude to MPI-SWS for providing an exceptional research environment and support. Claudia, Annika, and Gretchen deserve special acknowledgment for their assistance in both official and unofficial matters during my time at the institute. Claudia, particularly, went beyond being a colleague, offering a welcoming presence at the MPI office where help was always within reach.

My friends in Saarbrücken — Soumya, Ahana, Sreyasi, Srestha — added a layer of warmth to my life in Germany. Their presence not only made my stay enjoyable but also brought a touch of Bengali culture to a foreign land. Over time, Saarbrücken started feeling like home! A special mention goes to one of my dearest friends, Pallavi, who shared a similar journey from IIT Kharagpur to Berlin. Our daily conversations, her support, and the multitude of shared experiences played an irreplaceable role in my Ph.D. journey.

Back home, I owe a debt of gratitude to my friends — Abik, Atreyee, Rohit, and Sumana. They stood with me through many twists and turns, some from our school days and others from university, offering encouragement and genuine friendship.

My family, especially my parents, have been my continuous support system throughout this incredible journey. I am certain it was just as hard for them as it was for me to live so far away for these years. Our visits became even more infrequent during the difficult times of COVID-19. However, they always sought to understand, helping me in every possible way and shielding me from all concerns back home so I could focus on my work. Knowing they were always just a

phone call away was a tremendous source of strength during this time.

Finally, I want to thank Ayan for his constant presence, love, encouragement, and support, which have not only made this journey possible but truly remarkable. I feel extremely fortunate to have him in my life, cherish every step we have taken together so far, and look forward to all the adventures that lie ahead.

Contents

Introduction 1

Numerical programs solve intricate mathematical problems across various domains, including image and signal processing, graphics, simulations, decision-making, and controlling the behavior of systems that interact with the physical world. For instance, a variety of science and engineering problems require simulating hydrodynamics equations, which are approximated by numerical programs [114]. Furthermore, in the aviation industry, automated collision avoidance systems rely on numerical programs [117], which may incorporate machine learning models to prevent collisions. Clearly, in these systems, the consequences of unacceptable errors can be severe. Thus, ensuring their accuracy and performance is crucial to mitigate risks and guarantee safety.

In an ideal scenario, these numerical programs would utilize infinite real-valued arithmetic for computation. However, due to limited resources and the high cost of running programs with infinite precision, implementations of these programs utilize finite-precision arithmetic, such as floating-point [85] or fixed-point [176] arithmetic.

Consequently, these numerical programs introduce errors into the computation due to the difference between each finite precision value and real value potentially after each operation. While these errors are individually small, they can propagate through the computation in an unpredictable way and cause significant discrepancies from the ideal real-valued result. In some cases, these errors can even cause the program execution to take incorrect paths, resulting in completely meaningless outcomes.

In addition, floating-point arithmetic features special values such as not-a-number (NaN) and Infinity [113]. NaN represents an undefined or unrepresentable value that can arise from operations like dividing zero by zero. Infinity, on the other hand, occurs when a computation results in a value that is not representable with the maximum number of finite bits. These special values further contribute to the complexity of reasoning about and (manually) debugging computations in finite-precision arithmetic, posing challenges for developers.

Therefore, it is essential to detect, analyze, and quantify these errors automatically and to ensure that numerical programs provide accurate results within the limitations of finite precision arithmetic and sensor uncertainties.

1.1 Finite-Precision Error Analysis

There has been much work on verifying numerical errors for floating-point and fixed-point arithmetic. There exist several static analysis tools and techniques that (automatically) analyze numerical programs and obtain sound bounds for numerical errors. These techniques can

additionally verify the absence of special float values (NaN and infinity) in the computation.

These techniques can be broadly classified into two approaches. The first approach involves forward data-flow analyses [88, 89, 55, 53], that automatically compute sound error bounds on floating-point (and some also on fixed-point) roundoff errors. The second approach is based on global optimization [57, 162, 140] that formulates the error-bound computation as an optimization problem, where the objective is to maximize the absolute error bound subject to interval bounds on the input parameters of the program.

Both data-flow and optimization-based roundoff-error analyses have successfully been applied to compute sound bounds on the roundoff errors for many interesting benchmarks from various domains like scientific computing, physics simulation, and embedded systems. However, they have two significant limitations.

Limitation 1: Only Worst-Case Error Analysis The state-of-the-art tools mentioned above compute *worst-case* roundoff errors, i.e., at each step of the computation, they compute the largest possible absolute error that can appear for the given ranges of program inputs. This analysis can be overly pessimistic in the following scenarios.

First, consider the case where a program relies on the result '`res`' of a numerical computation to make a decision, as in `if(res){...} else{...}`. It may be the case that due to roundoff errors, the finite-precision computation makes a different decision than the real-valued one. Using a worst-case analysis in this situation would conclude that the finite-precision program *always* makes the different (wrong) decision. This result completely disregards the fact that not all inputs within the range of program inputs would necessarily lead to the program taking the wrong path.

Secondly, many numerical applications are often designed to be robust to some noise and tolerate errors—as long as they remain within acceptable bounds (e.g., in control systems, where one feedback iteration may compensate for a larger error of a previous one). For these programs, employing a worst-case analysis would result in overly pessimistic results as it would assume that the most significant error *always* occurs. This approach fails to effectively utilize the inherent error tolerance of the system.

Finally, the challenge of computing sound but tight bounds on roundoff errors is further complicated by the presence of probabilistic error behaviors introduced by approximate hardware [137, 152], which has gained significant attention due to its potential to enable energy-efficient computing in various applications. To efficiently utilize resources (i.e., minimizing power consumption and area), these hardware circuits may intentionally introduce large but infrequent errors. For such cases, the worst-case analysis is even more pessimistic as it *always* considers the largest (worst-case) errors, even if they appear rarely.

Limitation 2: Scalability Static worst-case roundoff error analyzers mostly focus on multivariate straight-line code written in a special syntax [162, 57, 88], with limited support for conditionals and loops [88, 89, 55]. As a result, their applicability is mostly limited to small numerical programs that consist of a single function. However, in practice, realistic numerical programs are longer and often contain complex data and program structures.

There do exist scalable static analysis tools – some of these tools [135, 88, 26] rely on abstraction-based static analysis techniques while others [119, 48] are based on bounded model checking and SMT decision procedures that perform exact bit-precise reasoning. Unfortunately, none of these techniques can effectively analyze large *numerical* programs – they suffer from significant over-approximations, require manual parameterization, or do not scale particularly for floating-point arithmetic due to their complexity.

In contrast to static analysis, dynamic analysis executes the program and observes its behavior at runtime [141, 178], usually offering better scalability to find errors in the programs. They can analyze large general programs with various program constructs, including loops and conditionals, without overapproximations. However, dynamic analyses are limited to the test inputs executed within specified analysis times. Unfortunately, unexplored inputs may potentially contain bugs. Consequently, these analyses cannot guarantee the absence of errors that is essential for safety-critical numerical systems.

Dynamic analyses that specifically focus on proving the presence of infinities [77], NaNs, or cancellation errors [22, 38, 182] exist [1]. Unfortunately, none of these techniques can handle large numerical programs automatically. For example, the tool S3FP [38] requires the user to generate a high precision ('real') version of the program and ensure that the original program and the generated code produce outputs with the same bit length. The scalability of back-end SMT solvers limits more guided techniques like symbolic execution [129] for floating-point theories.

In summary, the analysis of realistic numerical programs remains challenging due to the lack of scalability of sound worst-case roundoff error analyzers and the current limitations of static and dynamic analysis techniques for numerical programs.

1.2 Finite-Precision Optimization

Implementing numerical programs on actual hardware faces a tradeoff between accuracy and efficiency. While these programs are designed to account for errors that may arise from noisy environments or imprecise implementations, deploying them on hardware requires efficient resource utilization while staying inside these predefined error bounds. This optimization can be challenging, as higher precision reduces roundoff errors, but may be computationally expensive or even impractical for hardware with limited resources. On the other hand, reducing precision can conserve resources like chip area and computation time but may result in more significant roundoff errors. Therefore, achieving an optimal tradeoff between accuracy and efficiency is crucial in implementing numerical programs on resource-constrained hardware systems.

In this thesis, our primary focus is on specializing the tradeoff between accuracy and efficiency for neural networks (NN). These networks are becoming prevalent components in safety-critical systems, with applications ranging from closed-loop NN controllers for adaptive cruise control

[1] A very recent tool, EFTSanitizer [41], has been introduced since our work, demonstrating its ability to detect NaN, infinity, and cancellation errors in large numerical programs. While we have not included a comparison with this tool, it would be interesting to assess its capabilities under similar resource and time constraints.

for cars to aircraft collision avoidance systems for airplanes [111].

There have been efforts to automatically verify the safety of limited-size NN systems that bound errors from noises or inaccuracies [69, 68, 67]. However, implementing these high-precision floating-point NNs on resource-constrained embedded systems is often impractical due to high computation costs or the unavailability of dedicated floating-point co-processors or software-based emulation capabilities. As a result, trained NNs are typically quantized using low-precision fixed-point arithmetic [92, 87] to achieve efficiency in terms of area or latency.

Numerous approaches have been proposed for the quantization of neural networks. Some of these methods use a uniform precision for all layers of a model [92, 87, 120] which might be suboptimal; if one precision is barely not enough, we need to upgrade all operations to a higher precision. Also, all layers may not have similar effects on the overall accuracy of the model. To alleviate this problem, some work [146, 163, 159] focuses on mixed-precision quantization where different operations or layers are performed in different precision, thus saving more resources.

Despite the promising results for NN classification benchmarks, these techniques face the following challenge when it comes to optimizing NN controllers in safety-critical systems.

Limitation 3: Usability and Performance The quantization techniques mentioned above are specifically designed for NN *classifiers*, focusing on dynamically comparing classification accuracy on specific test datasets. As a result, they cannot handle NNs that do not perform classification tasks and rather implement regression tasks, which are common for NN controllers computing (continuous) control values. In addition, these techniques are unsound, i.e., they cannot guarantee (classification) accuracy, which is essential for safety-critical systems.

Several *sound* techniques for mixed precision tuning have been developed primarily for general-purpose numerical programs. There exist tools such as FPTuner [39] and Daisy [54, 109] that have successfully been applied to numerical programs from various domains, including scientific computing, simulations, and embedded controllers.

Unfortunately, applying these tools for NN mixed-precision fixed-point quantization is limited. FPTuner is specifically designed for floating-point programs and cannot handle fixed-point programs. While Daisy supports both floating-point and fixed-point arithmetic, it relies on a heuristic search approach that becomes intractable as program sizes grow. Additionally, Daisy works only on generic straight-line code without loops and does not handle programs with data structures like matrices and vectors.

In summary, the quantization of NNs performing regression tasks remains challenging due to the unavailability of dynamic analysis tools and the limited scalability of sound mixed-precision tuners for large NN fixed-point implementations involving matrices, vectors, and loops.

1.3 Thesis Contributions

This thesis aims to expand the horizons of state-of-the-art finite precision error analysis and optimization techniques in three directions corresponding to the identified limitations. First,

```
/* valid input range: [-15.0, 15.0] */
def controller(x:Float, y:Float, z:Float): Float = {
   val res = -x*y - 2*y*z - x - z
   if (res > 400.0) raise_alarm()
   else do_nothing()
   return res
}
```

Figure 1.1: A non-linear controller encoded in Scala

input distribution	# if	# else
$\mathcal{U}$ [-15, 15]	1563	98437
$\mathcal{N}$ (0, 7.5)	497	99503

Figure 1.2: Scenario 1: sampled frequency of taking branches with uniform and gaussian inputs

we propose a probabilistic solution to the overly pessimistic state-of-the-art worst-case error analyses of programs in the presence of numerical uncertainties. Secondly, we propose a scalable (conditionally) sound analysis that increases the reach of state-of-the-art numerical error analysis tools beyond what they can do individually for realistic numerical programs. Finally, we introduce a mixed-precision quantization for neural networks for resource-constrained embedded hardware. In the following section, we provide a detailed explanation of each of these contributions.

Solution 1: Probabilistic Analysis of Numerical Uncertainties We present novel probabilistic analyses that consider the probability distributions of program inputs, propagate them through the computations, and capture their effects on discrete decisions, where the finite-precision program may take a different path than the real-valued one, and on round-off errors.

To illustrate the motivation behind the proposed solutions, let us consider a 32-bit floating-point Scala implementation of a non-linear controller (shown in Figure 1.1) taken from [13]. It accepts three inputs `x`, `y`, and `z`, with valid ranges $[-15.0, 15.0]$ and computes a result '`res`' in the same precision.

We explain two scenarios with this example. First, let us assume that based on the result `res`, the program makes a discrete decision of whether an alarm should be triggered. For the sake of presentation, we show in Figure 1.1 a decision where the real-valued execution always takes the '`else`' path. However, due to unavoidable roundoff errors, there are certain inputs for which the finite-precision execution might deviate from the expected path and take the '`if`' branch instead.

Secondly, we assume the case where, like many real-world applications (e.g., control systems where one feedback iteration compensates for a larger error of a previous iteration), the program tolerates large roundoff errors when they occur with low probability.

To understand the behavior of the finite-precision execution for the decision, we sampled 10^5 inputs from the provided input ranges using both uniform and gaussian (with a mean and variance of 0.0 and 7.5, clipped within the given range) distributions in 32-bit floating-point precision. We executed the program with the sampled inputs and counted the frequencies of the `if` and `else` branches as shown in Figure 1.2.

The results in Figure 1.2 confirm that only a small subset of executions deviates and takes the wrong (`if` path for our example) branch due to roundoff errors. Therefore, in this scenario, a

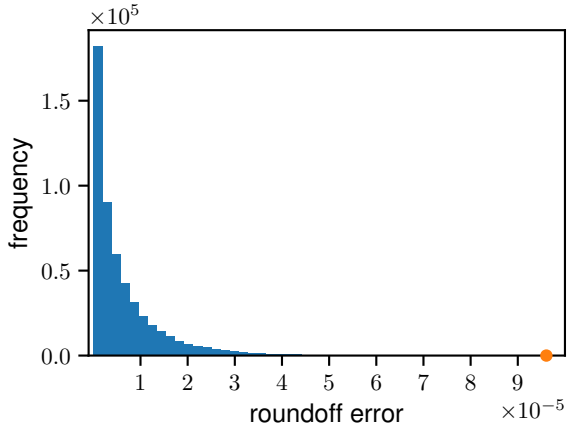

Figure 1.3: Scenario 2: sampled errors in 32-bit floating-point precision with uniform inputs

more meaningful metric would be to determine *how often* the program takes the wrong branch, which we refer to as the '*wrong path probability*'. Furthermore, these results also show that the varying probability distributions of inputs significantly affect the frequency of program deviations from its correct, real-valued path.

To get insights into the second scenario where the program tolerates large roundoff errors, we sampled the frequency of large roundoff errors. We started with uniformly randomly sampling 10^5 inputs and compared the absolute errors of the 32-bit floating-point implementation with an arbitrary-precision 300-bit implementation using the MPFR library [75]. The errors and their corresponding frequencies are shown in Figure 1.3.

From the plot in Figure 1.3 we observe that for most inputs, the errors are small, and the largest error marked with the orange-colored dot rarely occurs. Additionally, as the magnitudes of roundoff errors depend on inputs, the frequency of the largest (worst-case) error occurring would also depend on the probability distributions of inputs.

All these observations emphasize the necessity of considering the probability distribution of inputs and propagating them through the computation. By employing a probabilistic analysis of programs, we can achieve a precise estimate of the wrong path probability. Furthermore, through probabilistic analysis of errors, we can compute the probability of certain error bounds, thus capturing differentiated error behavior of programs.

Such a probabilistic error analysis is even more relevant in hardware with approximate architectures and storage. These hardware often introduce large errors in arithmetic operations with a certain probability in favor of increased efficiency. To precisely capture these probabilistic specifications and compute a tight error bound, a probabilistic analysis of errors is necessary.

We present two novel data-flow analyses that take into account the probability distribution of inputs and propagate them through arithmetic computations by leveraging and extending probabilistic affine arithmetic [11] [2].

Our first technique aims to compute precise wrong path probabilities for programs with

[2]Probabilistic affine arithmetic is thoroughly discussed in Section 2.3.

discrete decisions. However, computing precise estimates using probabilistic affine arithmetic alone is not enough as it tends to overapproximate the probabilities too much to be useful in practice. To mitigate this, we propose to subdivide the whole input domain into smaller subdomains, compute wrong path probabilities individually for each subdivision, and then (normalized) sum these to determine the overall wrong path probability.

Unfortunately, the high computational cost of probabilistic affine arithmetic limits its application across all subintervals. To address this, we further extend our technique with reachability checks to soundly determine subdomains that are actually relevant for computing the probability of wrong results. The probabilistic analysis is then only run where needed, thus making it precise and scalable enough for small yet realistic numerical applications.

We have implemented the analysis in a prototype tool on top of an existing worst-case roundoff error analyzer Daisy [55]. Our tool works on numerical programs with arithmetic and elementary operations. It considers different probability distributions on the inputs, including uncertain ones, and fully automatically computes wrong path probabilities considering roundoff errors in floating-point as well as fixed-point arithmetic. We have evaluated the tool on several embedded controllers as well as machine learning classifiers and obtained sound yet precise wrong path probabilities.

Our second technique focuses on computing probabilistic roundoff errors of numerical programs. We extend probabilistic affine arithmetic for errors that soundly overapproximate all possible distributions of errors. To reduce over-approximation in the analysis, we further combine it with a novel probabilistic version of interval subdivision and obtain an efficient analysis and a tight distribution of errors.

To facilitate the interpretation of error distributions inspired by probabilistic error specifications in approximate hardware, we also introduce a new error metric that computes a smaller error than the worst-case one that occurs with a predefined high probability. Our analysis can also consider errors with probabilistic specifications for approximate hardware systems.

We have developed a prototype tool called '*PrAn*' and evaluated it on several benchmarks from the literature. Our results show that PrAn fully automatically computes significantly smaller error bounds, i.e., it can determine that with probability at least 0.85 (for our experiments), the errors are, on average, 17% and 16.2% and up to 49.8% and 45.1% smaller than worst-case errors for gaussian and uniform input distributions, respectively.

Solution 2: Conditional Verification of Realistic Programs We propose a two-phase analysis to (conditionally) prove the absence of infinity and special floating-point values such as not-a-number (NaN) as well as cancellation errors in large programs with complicated programming and data structures.

To present the primary motivation behind this solution, let us consider the '`N-body`' program shown in Figure 1.4, taken from Rosetta Code [4]. Our goal is to verify the absence of special floating values and cancellation errors or to test for their presence in this program. This program takes as input the masses, positions, and velocities of three bodies, whose ranges are provided by the user, and simulates their interaction under gravity over several time steps. The program

```
int main(int argc, char* argv[]) {... // Reads masses, positions and velocities
  for(int i=0; i<timeSteps; i++) { simulate(mass, pos, v); ...}
}

void simulate() { compute_accelerations(mass, pos); ...}

void compute_accelerations(double mass[], vector pos[]){
  for(int i=0;i<bodies;i++){ ...
    for(int j=0;j<bodies;j++) {if(i!=j) {
      acc[i] = numerical_kernel(mass[j], pos[i], pos[j], acc[i]);}}}}

vector numerical_kernel(double mass, vector pos_i, vector pos_j, vector acc) {
  return add_vectors(acc, scale_vector(g*mass/pow(mod(subtract_vectors(pos_i,pos_j)),3),
    subtract_vectors(pos_j,pos_i)));} // compute acceleration
```

Figure 1.4: N-body problem encoded in C

is single-threaded and moderately sized, comprising 108 lines of code with control flows and function calls.

Unfortunately, none of the existing static and dynamic analysis tools could prove the absence or the presence of special floating-point values and cancellation errors in the 'N-body' program. We exhaustively employed all tools capable of directly handling the program without requiring extensive manual parameterization or annotation, such as Astreé [135], CBMC [119], AFLGo [27]. There is, therefore, a clear need for scalable, automated verification or debugging techniques for these programs.

Upon examining the N-body program, our main insight is that only a relatively small part of a program often performs complex numerical computations. For example, the function 'numerical_kernel' in Figure 1.4 is a small but numerically intensive part of the program. These functions can effectively be analyzed using state-of-the-art floating-point analyzers when provided with preconditions bounding the input ranges of the kernel. However, obtaining such preconditions manually is challenging as the kernels are nested behind several loops and function calls, as in the N-body program. To compute these ranges automatically, a scalable program analysis technique without sophisticated floating-point reasoning could be used.

Based on this observation, we propose a two-phase analysis framework that combines different program analyses to (conditionally) verify the absence of special values and cancellation errors in numerical kernels 'concealed' in large programs. First, we utilize a scalable program analysis to infer the ranges of inputs of a kernel in the context of the containing application. In the second phase, a sound floating-point analyzer assumes these ranges to verify the kernels. The verification may be conditional as it is performed under the assumption that the computed ranges precisely describe possible values of the kernel inputs.

We have developed a tool called '*Blossom*' that automatically infers the ranges of numerical kernels. Our evaluation shows that our framework can handle C and C++ programs with

```
def controller_tora(in: Vector): Vector = {
  weights1 = Matrix(List(List(-0.102887,...,0.015634),...,List(-0.152597,...,-0.072199)))
  bias1 = Vector(List(0.157552,...,0.165616))
  layer1 = relu(weights1 * in + bias1)
  weights2 = Matrix(List(List(0.14632,..., -0.032045),...,List(-0.089955,...,0.021892)))
  bias2 = Vector(List(0.205404,...,0.184322))
  layer2 = relu(weights2 * layer1 + bias2)
  weights3 = Matrix(List(List(-0.119052,..., -0.01671),...,List(0.116748,...,0.217796)))
  bias3 = Vector(List(0.179621,...,0.265042))
  layer3 = relu(weights3 * layer2 + bias3)
  weights4 = Matrix(List(List(-0.036561,...,0.264104)))
  bias4 = Vector(List(10.197819))
  out = linear(weights4 * layer3 + bias4)
  return out
} /* ensuring error <= 1e-3 */
```

Figure 1.5: Network Architecture of Tora Controller

several numerical kernels, complex programming structures, and over 2K lines of code. We used the computed kernel ranges generated by Blossom and could conditionally prove the absence of special floating-point values and cancellation errors for several kernels. Additionally, we reported overflow and cancellation errors in some cases, subsequently confirmed through counter-example generation or manual inspection. These results are beyond the capabilities of state-of-the-art static and dynamic analysis techniques.

Solution 3: Mixed Precision Quantization of Neural Networks In the final part of the thesis, we present the first sound and fully automated mixed-precision fixed-point quantization technique that specifically targets feed-forward deep neural networks.

We explain the motivation of our approach using a NN shown in Figure 1.5 written in a small domain-specific language. This NN, taken from [67], is a fully connected feed-forward neural network consisting of 4 inputs, 3 hidden layers, 20,800 parameters (including both weights and biases), and 1 output. The objective of this NN is to control translational oscillations by a rotational actuator. Assuming an error bound of $1e-3$, our goal is to soundly quantize the network and generate a fixed-point implementation that can be directly synthesized into an FPGA.

Among the existing techniques, only Daisy [54] can generate a sound mixed-precision fixed-point version of a program that guarantees a predefined error bound. In order to use Daisy, we converted the matrix and vectors to simple variables and unrolled the corresponding loops in Figure 1.5. However, after 5 hours, Daisy timed out and could not generate an optimized implementation for the network.

The key insight of our solution lies in recognizing that quantizing NN variables and constants while ensuring a roundoff error bound is fundamentally an optimization problem. This problem involves integer decision variables that represent the discrete choices of the number of bits to

use for NN operations. Additionally, we need real-valued constraints to ensure that the total error of the neural network remains within the specified bound and no overflow occurs.

Based on the above idea, we formulate the sound mixed-precision fixed-point quantization of NNs as a mixed integer linear programming (MILP) problem. A naive formulation, however, results in non-linear constraints that are computationally intractable. We show how to use over-approximations to relax the problem to linear constraints that are efficiently solvable.

We have developed a fully automatic prototype tool called '*Aster*', which generates a mixed fixed-point precision implementation that satisfies the error bound (with respect to real-valued input) and minimizes a customizable cost function. These implementations can be directly synthesized on FPGAs. Though Aster is primarily designed for continuous feed-forward regression neural networks, our proposed approach can be extended to other types of neural networks that have sparse matrices and activation functions that can be piece-wise linearized [175].

We have evaluated Aster extensively on NN embedded controller benchmarks from the literature [131, 111, 90]. Our results show that Aster is substantially faster and more scalable than Daisy, especially for larger networks with thousands of parameters, with improvements in optimization time on average of $\sim 67\%$.

1.4 Outline

We explore three avenues to expand the horizons of current finite-precision analysis and optimization: probabilistic analysis, scalable static and dynamic analyses, and optimization. Below, we provide an overview of the structure and chapters included in the thesis.

Chapter 2 presents the essential background on finite-precision formats, various types of errors that arise in computations, existing techniques for finite-precision error analysis and optimization, and probability propagation. We consider these concepts and techniques as fundamental baselines for the later chapters.

Chapter 3 presents a novel probabilistic analysis that captures the effects of probabilistic inputs and the numerical errors on discrete decisions. Our analysis targets both floating-point and fixed-point programs.

This chapter corresponds to the paper that appeared as part of the ESWEEK-TCAD special issue and was presented at EMSOFT'18 1. The prototype tool is publicly available on Zenodo: `https://doi.org/10.5281/zenodo.8042198`.

Chapter 4 introduces the first sound probabilistic error analysis for floating-point programs that considers probability distributions of inputs and analyzes roundoff errors probabilistically.

The material presented in this chapter is based on the publication at iFM'19 2. Our prototype tool PrAn implemented as an extension of the previous work, is also available on Zenodo: `https://doi.org/10.5281/zenodo.8042198`.

Chapter 5 proposes a two-phase analysis for conditional verification of special floating-point values and cancellation errors in complex numerical functions embedded within larger programs.

The content of this chapter corresponds to the publication at TACAS'21 3. Our prototype tool '*Blossom*' is publicly available on Zenodo: `https://doi.org/10.5281/zenodo.8043359`.

Chapter 6 presents the first sound and fully automated mixed-precision quantization technique that specifically targets deep feed-forward neural networks.

The work presented in this chapter is based on the publication that appeared as part of the ESWEEK-TECS special issue and was presented at EMSOFT'23 4. The tool '*Aster*' is publicly available on Zenodo: `https://doi.org/10.5281/zenodo.8123416`.

Chapter 7 reviews the extensive body of related work in automated numerical error analysis and optimization.

Chapter 8 concludes the thesis and explores possible future directions.

1.5 Publications

The content of the thesis is based on the following publications in peer-reviewed conferences and journals. Although I am the primary author of these papers, I will use the academic 'we' throughout the thesis to acknowledge the support and contributions of my collaborators, without whom this work would not have been possible.

1. **Debasmita Lohar**, Eva Darulova, Sylvie Putot, and Eric Goubault, "***Discrete Choice in the Presence of Numerical Uncertainties***", in International Conference on *Embedded Software (EMSOFT)* published in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2381-2392, Nov. 2018, doi:10.1109/TCAD.2018.2857320

2. **Debasmita Lohar**, Milos Prokop, and Eva Darulova, "***Sound Probabilistic Numerical Error Analysis***", in International Conference on *Integrated Formal Methods (IFM)*, vol 11918, pp. 322-340, 2019, doi:10.1007/978-3-030-34968-4_18

3. **Debasmita Lohar**, Clothilde Jeangoudoux, Joshua Sobel, Eva Darulova, and Maria Christakis, "***A Two-Phase Approach for Conditional Floating-Point Verification***", in International Conference on *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 12652, pp. 43–63, 2021, doi:10.1007/978-3-030-72013-1_3

4. **Debasmita Lohar**, Clothilde Jeangoudoux, Anastasia Volkova, and Eva Darulova, "***Sound Mixed Fixed-Point Quantization of Neural Networks***", in International Conference on *Embedded Software (EMSOFT)* published in *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5s, pp. 1–26, Sept. 2023, doi:10.1145/TECS.2023.3609118

Background 2

In this section, we lay the essential groundwork that serves as a fundamental baseline for this thesis. We begin by introducing finite-precision formats, arithmetic, and the errors that arise during computation. Then, we present the state-of-the-art techniques that estimate finite-precision ranges for programs and compute bounds on finite-precision errors.

Furthermore, we present an existing technique for probability propagation [1], which plays an essential role in the first part of the thesis. Our proposed techniques presented in chapters 3 and 4 are built on these analyses to propagate input distributions through numerical computations.

Finally, we present the important background of finite-precision optimization, which seeks to minimize resources while remaining within specified error bounds.

2.1 Finite-Precision Formats

Implementing real-valued arithmetic on hardware introduces the need for using finite-precision arithmetic due to the inherent limitations in representing real numbers with a finite number of precision bits. This section will delve into the two most frequently used finite-precision formats in numerical programs: floating-point and fixed-point representation.

2.1.1 Floating-Point Representation

Floating-point representation [85] is the most commonly used method for expressing real numbers in finite precision. A floating-point number, denoted by x, can be represented as follows:

$$x = (-1)^s \times m \times \beta^e \tag{2.1}$$

Here, $s \in \{0, 1\}$ determines the sign of the number. The significand, denoted as m, is given by $\mid M \mid \times \beta^{1-p}$, where β stands for the radix or base, and p represents the precision. The significand consists of one digit before the radix point and up to $p-1$ digits after the point. The exponent e is an integer, such that $e_{min} \leq e \leq e_{max}$, that determines the scaling of the number.

To ensure a more expansive range of representable values while maintaining precision, the normalized representation of floating-point numbers is commonly employed. In this representation, the most significant bit of the significand is assumed to be 1. Consequently, the exponent e determines the position of the radix point, allowing both large and small numbers to be represented. This representation also ensures uniform spacing between consecutive

[1] Other probabilistic analyses, similar or complementary to our techniques, are described in chapter 7

precision	word size	sign	exponent	significand	exponent bias
single	32	1	8	23	127
double	64	1	11	52	1023
quad	128	1	15	112	16383

Table 2.1: Floating-point formats specified by IEEE 754 standard and values for exponent bias

representable numbers across the entire range.

IEEE Standard for Floating-Point The IEEE Standard for Floating-Point Arithmetic (IEEE 754-2019) [100] governs the computation and standardized representation of floating-point numbers, their operations, and rounding behavior with radix 2 and 10. For the scope of this thesis, we center our focus on the extensively employed binary floating-point numbers utilizing a base of 2.

Within this standard, a normalized representation for floating-point numbers assumes that the most significant bit of the significand is perpetually set to 1. Consequently, in formats adhering to the IEEE specification, there is no requirement to explicitly store the first bit of the significand. Instead, only the trailing significand (p-1 bits) is stored, which is often referred to as the fraction.

Given that the exponent's range spans positive and negative values, a biased representation is employed in this standard. In single precision, where the exponent is allotted 8 bits, the bias is set at 127 (for double precision, it is 1023). This implies that if the exponent bits are interpreted as an unsigned integer, the effective exponent of the floating-point number is derived by subtracting 127 (or 1023) from this integer value. This unbiased exponent, derived through subtraction, serves to differentiate it from the biased exponent.

The IEEE standard defines three distinct binary precision levels: single (32-bit), double (64-bit), and quad precision (128-bit). The key attributes for these precision levels, including word size, the sign bit, exponent field size, significand size, and the respective exponent bias, are summarized in Table 2.1. This standardized categorization brings consistency and comparability across various computing platforms, facilitating uniform representation and computation of real numbers in finite precision.

Rounding Modes The IEEE standard defines the following four rounding modes that determine how results of arithmetic operations are rounded when they cannot be represented exactly in the chosen precision.

1. Round to nearest (ties to even): This mode rounds to the nearest representable value, and in the event of a tie, it selects the value with the even least significant digit. In the case of a tie and both having an odd least significant digit (this can happen only at precision 1, possibly when converting a number like 2.5 into a decimal string), the value with a larger magnitude is selected.

2. Round up (toward $+\infty$): This mode rounds up to the next representable value larger than the exact result.

3. Round down (toward $-\infty$): This mode rounds down to the next representable value smaller than the exact result.

4. Round toward zero: This mode simply truncates the extra bits beyond the desired precision.

The IEEE standard requires that the result of addition, subtraction, multiplication, division, and square root be exactly rounded. That is, the result must be computed exactly and then rounded to the nearest floating-point number (using ties to even).

Special Values The IEEE floating-point format also defines special values when the exponent field is composed entirely of 0s or 1s.

When the exponent has only 0s, the corresponding number is referred to as a *denormalized* or *subnormal* number. In this scenario, a 0 is implicitly assumed before the binary point. For instance, in both single and double precision, denormalized numbers follow the pattern $(-1)^s \times 0.f \times 2^{-126}$ and $(-1)^s \times 0.f \times 2^{-1022}$, where f represents the fraction and s denotes the sign bit.

Furthermore, the value *zero* is also considered a denormalized number with all fractional bits set to 0. It may be noted that despite their equivalence in floating-point comparison, -0 and $+0$ are different.

Infinity values ($+\infty$ and $-\infty$) are indicated by an exponent consisting entirely of 1s and a fraction entirely of 0s. The sign bit distinguishes between positive infinity and negative infinity. Operations involving infinities adhere to well-defined rules in IEEE floating-point format and hence are propagated through the computations.

The concept of *NaN* (*Not a Number*) is employed to signify a value that does not represent a real number. NaN values are represented by a specific bit pattern featuring an exponent of all 1s and a non-zero fraction. There are two types of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN). A QNaN typically is a result of an undefined mathematical operation and propagates freely through most arithmetic operations without signaling exceptions. QNaN is characterized by the highest fraction bit being set. An SNaN, on the other hand, has the highest fraction bit cleared and is used to signal an exception in the operations.

Floating-point stands as an efficient approach for finite precision, and operations in this format are supported by dedicated hardware units known as Floating-Point Units (FPUs). These FPUs streamline the execution of floating-point operations, offering exceptional computational speed while adhering to the IEEE 754 standards. FPUs are well-integrated into modern processors.

Nevertheless, certain scenarios, particularly when involving high-precision computations, may encounter limitations in hardware support. In such cases, performing high-precision floating-point computations requires resorting to software simulation, incurring substantial computational overhead. Consequently, implementing floating-point operations on resource-constrained applications may be too expensive due to the unavailability of dedicated hardware support or the high computational cost associated with software simulation-based approaches.

2.1.2 Fixed-Point Representation

Fixed-point representation emerges as a compelling alternative to the floating-point format [176]. Unlike floating-point, fixed-point maintains a constant number of decimal places that cannot be altered dynamically. Fixed-point representation is particularly well-suited for scenarios with only integer arithmetic hardware support, making it a common choice for embedded devices that often lack a dedicated floating-point unit due to resource constraints.

In a fixed-point implementation, all program variables and constants are represented using integers and have an (implicit) representation as a triple $<s, Q, \pi>$, where:

- $s \in \{1, 0\}$ indicates the sign bit (1 for signed, 0 for unsigned),
- $Q \in \mathbb{N}$ denotes the total word length (overall number of bits), and
- $\pi \in \mathbb{N}$ represents the position of the binary point, counted from the least significant bit.

The representation effectively divides the overall number of bits into an integer part, I, and a fractional part, π. (Since the position of the binary point determines the number of bits allocated to the fractional part, we use the same notation for the fractional part.) The number of integer bits is computed as $I = Q - \pi - 1$. The range of representable numbers in this integer range is defined as $[-2^I, 2^I]$. The fractional part controls the *precision* of a variable or an operation—the larger the number of fractional bits, the more precisely the value can be represented.

Arithmetic operations on fixed-point variables can be implemented efficiently using only integer arithmetic and bit-shifting [14] or can use equivalent efficient hardware implementations, e.g., on FPGAs.

Fixed-point representation also offers the advantage of efficient bit utilization. Unlike floating-point numbers, which come in only three standard precisions and have a fixed number of bits allocated for the significand, fixed-point representation allows for assigning any number of bits as a valid precision. This inherent flexibility provides manual control over precision and accuracy, enabling fine-tuning each operation as required.

However, employing fixed-point representation does present certain challenges. The integer bits must be decided before compilation for all variables and constants of a program. These integer bits should be sufficient to prevent overflow during computations. Additionally, within the same program, variables may possess differing formats (varied numbers of integer and fractional bits). Hence, if the formats of two operands of an arithmetic operation do not match, the operands must be aligned with respect to the binary point prior to computation. Thus, writing such fixed-point programs manually is difficult and prone to errors.

2.2 Finite-Precision Analysis

Finite precision representation and its computations introduce roundoff errors, potentially at every operation. These errors are generally small, but they can unexpectedly propagate through

numerical computations and can make the results absolutely meaningless [97]. Hence, it is important to compute a bound on these errors.

Two different approaches exist to soundly bound these errors. One approach employs global optimization techniques [132, 140, 162, 56], while the other relies on data-flow analysis techniques [60, 64, 53, 52, 55]. Both these techniques compute the errors in the worst case, i.e., assume that the largest error always happens.

The optimization-based techniques reduce the computation of roundoff errors as an optimization problem, aiming to maximize the errors. Tools like real2Float [132], FPTaylor [162], and Satire [56] leverage symbolic Taylor expansions to formulate the maximization objective, whereas Precisa [140] employs a parametric abstract analysis for roundoff errors.

In contrast, data-flow analysis-based techniques use the domains of interval arithmetic (IA) [139] and affine arithmetic (AA) [62] to track ranges, compute worst-case errors at each operation, and propagate both the ranges and errors throughout the computation. In this thesis, we adopt data-flow analysis-based approaches to compute roundoff errors of numerical programs. Hence, the following sections center on this particular approach, where we delve into the details of the abstract domains of IA and AA for ranges and roundoff errors.

2.2.1 Interval Arithmetic

In numerical computations, the most natural way of presenting the ranges is using intervals. An interval X is defined with a lower and an upper bound $[l_x, u_x]$, where a variable $x \in X$ can take any value between these bounds, i.e., $l_x \leq x \leq u_x$. When the lower and upper bounds are equal, the interval becomes a *point interval*. Basic arithmetic operations on intervals are defined as:

$$[l_x, u_x] \circ [l_x, u_x] = [\min_{l \in R} l,\ \max_{u \in R}\ u],\ \ R \in \{l_x \circ l_y, l_x \circ u_y, l_y \circ l_x, l_y \circ u_x\} \tag{2.2}$$

where, $\circ$ represents the operations, i.e., $\circ \in \{+, -, *, /\}$. The square root is computed analogously. For unary, operations are applied to the lower and upper bounds of the variable.

Despite its efficiency, IA has limitations. It cannot track correlations between variables (textitdependency problem). In IA, $X - X$ does not produce $[0, 0]$; instead, it yields $[l_x - u_x, l_x + u_x]$ whose diameter is twice the width of X. This limitation leads to high over-approximations of the true ranges, especially in complicated expressions or long iterative computations.

2.2.2 Affine Arithmetic

To address the dependency problem and track linear correlations, affine arithmetic (AA) was introduced. An affine variable $\hat{x}$ is represented as an affine expression on the *noise symbols* ε_i:

$$\hat{x} := x_0 + \sum_{i=1}^{n} x_i \varepsilon_i,\ \ \varepsilon_i \in [-1, 1] \tag{2.3}$$

where x_0 is the *central value* of the affine form, and the co-efficients $x_1, \cdots x_n$, which are called *partial deviations*, determine the deviation of noise symbols $\varepsilon_1, \cdots \varepsilon_n$ around x_0. The number of noise symbols depends on the specific affine form, and different affine forms can use a varying number of noise symbols, some of which may be shared with other affine forms, thereby capturing correlations between variables.

Let us return to the example of subtracting a variable from itself ($x - x$). In AA, the difference between an affine form and itself is identically zero because both operands of the subtraction share the same noise symbols with the same coefficients.

An affine form $\hat{x}$ computes the ranges of the corresponding variable x as follows:

$$x \in \left[x_0 - \sum_{i=1}^{n} \mid x_i \mid,\ x_0 + \sum_{i=1}^{n} \mid x_i \mid\right]$$

where $\sum_{i=1}^{n} |x_i|$ defines the *total deviation* from x_0.

AA can compute precise results for linear operations that are computed term-wise:

$$\alpha\hat{x} + \beta\hat{y} + \zeta = (\alpha x_0 + \beta y_0 + \zeta) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\varepsilon_i$$

However, non-linear operations are more involved as they need to be approximated by linearization. For example, a multiplication operation of two affine variable $\hat{x}$ and $\hat{y}$ is defined as follows:

$$\hat{x} * \hat{y} = x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i) + T\varepsilon_{n+1} \tag{2.4}$$

where $T\varepsilon_{n+1}$ is the upper bound for the approximation error for the linearization. Other non-linear functions are computed similarly. Due to this approximation, the results of non-linear operations lose correlation with the operands and are less precise. For more details on AA, we refer the readers to [62].

2.2.3 Computing Roundoff Errors

Roundoff errors can be expressed in terms of either absolute errors or relative errors. For a real-valued function $f(x)$ with a real-valued input x in a range $[a, b]$, the absolute error is defined as follows:

$$\epsilon_{abs} = \max_{x \in [a,b]} \mid f(x) - \tilde{f}(\tilde{x}) \mid \tag{2.5}$$

In this equation, $\tilde{f}(\tilde{x})$ represents the finite-precision counterpart using a finite-precision input $\tilde{x}$. The relative error is applicable when the evaluation of $f(x)$ does not involve zero, and it is defined below.

$$\epsilon_{rel} = \max_{x \in [a,b]} \mid \frac{f(x) - \tilde{f}(\tilde{x})}{f(x)} \mid \tag{2.6}$$

However, relative errors can only be defined for programs where $f(x)$ does not include zero. Though there exist some methods to bound relative errors [108], most state-of-the-art tools focus on absolute error analysis [60, 64, 53, 52, 55]. Our work also focuses solely on this type of analysis. Hence, this section will delve into techniques for computing absolute errors.

Computing absolute errors in an automated way using Equation 2.5 is not feasible for several reasons. First, the floating-point function $\tilde{f}(\tilde{x})$ with floating-point input $\tilde{x}$ is highly irregular and discontinuous, making it unsuitable for automated analysis. Secondly, the real-valued evaluation of the function is not available for most inputs and functions. Hence, the computation of the finite precision function $\tilde{f}(\tilde{x})$ is performed by replacing all operations, variables, and constants with the following abstraction:

$$x\tilde{\circ}y = (x \circ y)(1+e) + d, \ \mid e \mid \leq \epsilon_M, \ \mid d \mid \leq \delta_M, \quad \text{where } \circ \in \{+, -, \times, \div\} \tag{2.7}$$

The computation of the square root follows a similar approach, and the unary minus operation introduces no roundoff errors. It is important to note that this abstraction assumes a rounding-to-nearest mode, the standard default, and no overflow scenarios.

In Equation 2.7, $\tilde{\circ}$ denotes the finite-precision counterparts of operations present in $\{\circ\}$ set. The symbol ϵ_M signifies the machine epsilon, relevant to normal values, and establishes the upper limit on the relative error for a specific finite precision. The symbol δ_M dictates roundoff errors for subnormal values. For floating-point precision, the values of ϵ_M and δ_M are 2^{-24} and 2^{-150} respectively for single precision, and 2^{-53} and 2^{-1075} for double precision. In the fixed-point format, ϵ_M is defined by the number of bits dedicated to the fractional part ($2^{-\pi}$, following our notation of fixed precision highlighted in Section 2.1.2), while δ_M equates to 0.

Thus, the finite-precision implementation of a function $\tilde{f}(\tilde{x})$ can be abstracted with e and d as $f(x, e, d)$ considering that there is no special value (NaN, infinity). This abstraction is used by state-of-the-art roundoff error analysis tools that compute the absolute roundoff errors in the worst-case and use the following equation:

$$\epsilon \leq max \mid f(x) - f(x, e, d) \mid \tag{2.8}$$

where x is defined within the range $[a, b]$ while e and d retain their previous definitions, and the max function defines the error in the worst case.

Data-Flow based Error Analysis State-of-the-art data-flow analysis tools [60, 64, 53, 52, 55] for roundoff errors systematically traverse the abstract syntax tree (AST) of an arithmetic program and compute worst-case roundoff errors at the output. The process involves two steps:

1. *range analysis* that computes real-valued ranges
2. *error analysis* that propagates errors from subexpressions and computes the new worst-case roundoff errors using the ranges computed in the first step.

Thus, the ranges and errors are propagated through the computation to produce a sound range of the output and a sound bound on the roundoff errors in the final result. Both these steps

can be performed in a single traversal of the AST or as separate passes. Using two separate passes enhances modularity, enabling the utilization of diverse methods, such as the domains of interval arithmetic (IA) or affine arithmetic (AA), for precise range and error computation. This adaptability provides a better accuracy efficiency tradeoff, making it feasible to customize strategies based on the program's specifics and the objective of the analysis.

A fundamental challenge in data-flow analysis is minimizing over-approximations in the computed errors due to non-linear operations, as both IA and AA methods tend to overestimate the ranges. Existing tools utilize different strategies to reduce over-aproximation.

For instance, tools like Fluctuat [64], Rosa [53], and Daisy [55] combine *interval subdivision* with interval arithmetic (IA) and affine arithmetic (AA). In this approach, the input domain is subdivided into subintervals of typically uniform width, and then separate error analyses are performed on each subinterval. Finally, the overall error is computed as the maximum error over all subintervals. This method computes tighter error bounds, as smaller input domains tend to generate reduced over-approximations. However, this approach does come at the cost of increased analysis time, particularly for functions involving multiple variables. Another strategy to reduce over-approximation is to use Satisfiability Modulo Theories (SMT) solvers. Both Rosa and Daisy incorporate interval arithmetic and a non-linear SMT solver. This combination iteratively refines error bounds over the course of the analysis, thus providing tighter error bounds. However, the runtime of this method depends heavily on the SMT solver.

All these roundoff error analysis tools compute errors in the worst-case scenario. However, such bounds are often overly pessimistic. In practice, these worst-case bounds may not be realized, and certain applications might be tolerant of slightly larger but infrequent errors. The current roundoff error analysis techniques are unable to account for such nuanced (probabilistic) behavior. Additionally, these techniques do not offer insights into the distribution of variables within the provided input ranges.

2.3 Probabilistic Static Analysis

While the state-of-the-art analysis does not handle probabilistic errors, there are approaches that can compute probabilistic ranges by considering input distributions and propagating them through numerical computations. These methods extend IA and AA further with Interval Dempster Shafer (DSI) structures and probabilistic affine arithmetic. In the following sections, we review the essential background of these probabilistic domains and introduce the terminologies that we will use in chapters 3 and 4.

2.3.1 Dempster Shafer Interval Analysis

Interval Dempster-Shafer (DSI) structures [158] associate probabilities (or weights) with a set of intervals. These intervals, together with the probabilities, are called focal elements. On these

focal elements, the arithmetic operations are defined that allow the probabilities to propagate through the numerical computation. Here, we provide a short overview of the DSI structures and the arithmetic operations. For more details, we refer the interested readers to [158, 71].

DSI Structure Formally, a DSI structure consists of a finite set of focal elements represented by closed intervals, as follows.

$$d = \{\langle \mathbf{x_1}, w_1 \rangle, \langle \mathbf{x_2}, w_2 \rangle, \cdots, \langle \mathbf{x_n}, w_n \rangle\} \tag{2.9}$$

Here, $\mathbf{x_i} \in I$ is a closed non-empty interval, $w_i \in]0, 1]$ is the associated probability, and the sum of all the weights is 1: $\sum_{i=1}^{n} w_i = 1$. The focal elements express that the value of a variable represented by d is within $\mathbf{x_i}$ with probability w_i, but the variable can (non-deterministically) pick any value within the interval $\mathbf{x_i}$. We show an example of a DSI structure below.

$$d = \langle [-1, 0.25], 0.1 \rangle, \langle [-0.5, 0.5], 0.2 \rangle, \langle [0.25, 1], 0.3 \rangle, \langle [0.5, 1], 0.1 \rangle, \langle [0.5, 2], 0.1 \rangle, \langle [1, 2], 0.2 \rangle$$

The DSI d represents the set of probability distributions with support $[-1, 2]$, where the probability of picking a value between -1 and 0.25 is 0.1, the probability of picking a value between -0.5 and 0.5 is 0.2, and so on.

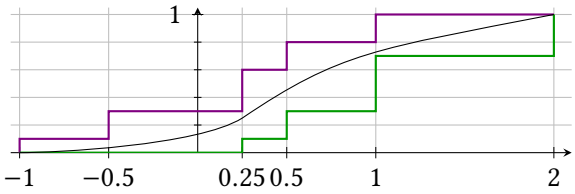

Figure 2.1: $\underline{P}$ (in green) and $\overline{P}$ (in violet) soundly bounds the distribution of x(in black)

An alternative way to represent these probabilities is to use a pair of discrete cumulative distribution functions (CDFs) $\underline{P}$ and $\overline{P}$ such that $\underline{P}$ is left-continuous, $\overline{P}$ is right-continuous step functions for all values of the variable x. Such a pair $[\underline{P}, \overline{P}]$ is known as discrete probability box or P-box [71], that encloses all probability distributions whose CDF P satisfy $\forall x, \underline{P}(x) \leq P(x) \leq \overline{P}(x)$. Figure 2.1 shows a P-box graphically that bounds the actual distribution.

Arithmetic Operations Let $d_X = \{\langle \mathbf{x_i}, w_i \rangle, i \in [1, n]\}$ and $d_Y = \{\langle \mathbf{y_j}, v_j \rangle, i \in [1, m]\}$ be two DSI structures. We want to compute $d_Z = d_X \odot d_Y$ where $\odot \in \{+, -, \times, \div\}$. These arithmetic operations distinguish the following two cases.

1. **Independent:** In case two DSIs d_X and d_Y are *independent*, computing the resultant DSI structure $d_Z = d_X \odot d_Y$ is straightforward – intervals are computed with usual interval arithmetic as shown in Section 2.2.1 and the weights are simply multiplied.

$$d_Z = \{\langle \mathbf{z_{i,j}}, r_{i,j} \rangle \mid i \in [1, n], j \in [1, m]\} \text{ where, } \mathbf{z_{i,j}} = \mathbf{x_i} \odot \mathbf{y_j}, r_{i,j} = w_i \times v_j \tag{2.10}$$

It may be noted that the number of focal elements grows exponentially with each of these operations.

2. **Dependent with unknown dependency:** The *dependent* case is more involved as it needs to consider *any* dependency between d_X and d_Y to compute a sound over- and under-approximation of the probability. Thus, for arithmetic operations on dependent DSIs, the alternate representation of the P-box is more suitable. The solutions are first computed for P-boxes, and then the resultant P-boxes are transformed into DSIs. Here, we briefly explain the method proposed in [24], which we have used for our analysis.

   To compute the solution $d_Z = d_X \odot d_Y$ as a P-box $[\underline{P_{d_Z}}, \overline{P_{d_Z}}]$, we first compute the arithmetic operation on the intervals of all pairs of focal elements (as in the independent case), obtaining a matrix of intervals: $\boldsymbol{x_i} \odot \boldsymbol{y_j} = [\underline{z_{i,j}}, \overline{z_{i,j}}]$. However, computing the probabilities as before ($r_{i,j} = w_i \times v_j$) is not possible. We only know the following constraints, corresponding to the rows and columns of this matrix:

$$\forall i \in [1,n],\ \sum_{j=1}^{m} r_{i,j} = w_i \qquad \forall j \in [1,m],\ \sum_{i=1}^{n} r_{i,j} = v_j$$

   Intuitively, we want to compute an upper and a lower bound on the cumulative distribution function at every point in the domain. Since the P-boxes are step functions; we effectively only need to perform this computation at the endpoints of the intervals in the interval matrix. The maximization, resp. minimization of the cumulative distribution function can be phrased as a linear program:

$$\begin{array}{rcll} \underline{F_{d_Z}}(z) & = & \textbf{minimize} & \sum_{\overline{z_{i,j}} \le z} r_{i,j} \\ & & \textbf{such that} & \forall i \in [1,n],\ \sum_{j=1}^{m} r_{i,j} = w_i \\ & & & \forall j \in [1,m],\ \sum_{i=1}^{n} r_{i,j} = v_j \end{array}$$

   Thus, we compute the lower P-box. The constraint $\sum_{\overline{z_{i,j}} \le z} r_{i,j}$ expresses that all $r_{i,j}$'s have to be taken into account, whose corresponding intervals are definitely below z. The formula for $\overline{P_{d_Z}}$ is analogous.

   After generating the resultant P-box $[\underline{P_{d_Z}}, \overline{P_{d_Z}}]$ we need to convert it back to the DSI d_Z. The conversion is straightforwardly done by first obtaining the intervals $\boldsymbol{x_i}$ by matching the lower and upper P-Box, then weights w_i are computed from the height of the steps.

The dependent arithmetic is clearly costly and, in addition, may lose some precision. It is thus important to keep track of dependencies between variables differently in order to be able to apply independent arithmetic operations as much as possible. This can be done by utilizing affine arithmetic, which already takes into account linear dependencies and allows to use of independent operations on the noise symbols, thus improving both the efficiency and the accuracy of the method.

2.3.2 Probabilistic Affine Analysis

DSI structures effectively propagate probability distributions with intervals; however, they cannot track linear dependencies between variables. To address this limitation and track dependency information, Bouissou et al. [30] extended affine arithmetic with DSIs. This extension enables DSIs associated with noise symbols to compute probability distributions while affine arithmetic handles the (linear) dependency information between them.

Probabilistic Affine Forms Concretely, a probabilistic affine form for a variable x is computed by summing the weighted DSIs associated with each noise symbol:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i d_{\epsilon_i} + \sum_{j=1}^{m} x_j d_{\eta_j} \tag{2.11}$$

This representation explicitly distinguishes between independent noise symbol d_{ϵ_i} and dependent noise symbol d_{η_j} with an undefined dependency. Both these noises are basically DSIs with support $[-1, 1]$. Thus, noise symbols in standard affine arithmetic track nondeterministic uncertainty, while probabilistic affine forms can capture detailed probabilistic information.

Arithmetic Operations Linear and unary arithmetic operations over probabilistic affine forms work exactly the same as standard affine form operations as presented in Section 2.2.2, and the noise DSIs remain the same. However, for non-linear operations, like multiplication, bounding the non-linear part requires generating a new noise symbol with unknown dependency along with the existing noises.

Let us assume $\hat{x}$ and $\hat{y}$ to be two probabilistic affine forms with d_{ϵ_i} d_{η_j} corresponding to independent and dependent noise terms, respectively. For the resultant probabilistic affine form $\hat{z} = \hat{x} \times \hat{y}$, the affine term is computed using affine arithmetic rules, and the existing noise terms remain the same. Additionally, a new noise symbol $d_{\eta_{m+1}}$ is introduced to represent the unknown dependency, and its definition is given below. For more details, we refer readers to [30].

$$\begin{aligned} d_{\eta_{m+1}} = \frac{1}{T} \Bigg(& \sum_{i=1}^{n}\sum_{j=1}^{m}(x_i y_j + x_j y_i) d_{\epsilon_i} \times d_{\eta_j} + \sum_{i=1}^{n}\sum_{k>i}^{n}(x_i y_k + x_k y_i) d_{\epsilon_i} \times d_{\epsilon_k} \\ & + \sum_{j=1}^{m}\sum_{l>j}^{m}(x_j y_l + x_l y_j) d_{\eta_j} \times d_{\eta_l} + \frac{1}{2}\Big(\sum_{i=1}^{n} x_i y_i d_{\epsilon_i}^2 + \sum_{j=1}^{m} x_j y_j d_{\eta_j}^2\Big) \Bigg) \end{aligned}$$

where T is defined as follows:

$$\begin{aligned} T = & \sum_{i=1}^{n}\sum_{j=1}^{m} \mid x_i y_j + x_j y_i \mid + \sum_{i=1}^{n}\sum_{k>i}^{n} \mid x_i y_k + x_k y_i \mid \\ & + \sum_{j=1}^{m}\sum_{l>1}^{m} \mid x_j y_l + x_l y_j \mid + \frac{1}{2}\Big(\sum_{i=1}^{n} \mid x_i y_i \mid + \sum_{j=1}^{m} \mid x_j y_j \mid\Big) \end{aligned}$$

The arithmetic operations corresponding to independent and dependent noise terms become independent and dependent arithmetic, respectively.

The abstract domains of DSI and probabilistic affine forms have been utilized to propagate input distributions along with ranges through numerical computation. However, these methods have not been previously used in the context of finite-precision error analysis. In this thesis, we extend and adapt these methods to the context of the effects of discrete decisions and roundoff error computation.

2.4 Finite-Precision Optimization

Finite-precision roundoff error analysis centers on assessing the precision of numerical programs. Increasing precision decreases roundoff errors, but there is always a tradeoff between accuracy and the efficiency or cost of resources. The more precise the computation, the less resource-efficient it is. This tradeoff is addressed by the finite-precision optimization. Various strategies can be employed to optimize numerical programs, from enhancing performance while maintaining accuracy bounds to improving accuracy while preserving performance.

This thesis utilizes both of these optimization aspects. We employ *expression rewriting* to increase accuracy and *mixed precision tuning* to reduce the cost of resources in chapters 5 and 6, respectively. In the following section, we provide details of these techniques implemented in the state-of-the-art tool Daisy[54], which serves as our chosen baseline framework. We will discuss other relevant techniques and tools within the context of related work in chapter 7.

Expression Rewriting In finite-precision arithmetic, several properties that hold true in real arithmetic no longer apply. For instance, the simple associative property does not hold in finite precision, i.e., $(a + b) + c \neq a + (b + c)$. This discrepancy arises from the distinct roundoff errors that occur when adding a and b before adding c and when the order of addition is altered. This non-associativity presents an avenue for optimization, specifically in the form of rewriting expression evaluation orders to minimize roundoff errors.

However, determining the most optimal expression rewrite to minimize roundoff errors is a complex task, given the many ways to order an expression. Consequently, the search space for the optimal evaluation order can be expensive. Thus, exhaustive enumeration is impractical for expressions containing a large number of operations.

The state-of-the-art tool, Daisy [54], offers a rewriting optimization that finds a better ordering of the arithmetic expressions in terms of roundoff errors. This optimization strategy relies on a genetic algorithm-based search approach, utilizing real-valued properties like associativity and distributivity to rearrange the expressions.

Daisy starts this optimization with an initial population consisting of a predetermined number of copies of the original expression. In each search iteration, the population is sorted based on their respective rounding errors. The most suitable candidate's order of evaluation is then randomly mutated while adhering to the rules of real-valued identities, and the errors are

computed again. This iterative process is repeated for 30 iterations, and finally, it returns an expression with the smallest seen roundoff error.

It may be noted that this method employs a heuristic search, which is incomplete. However, the technique has demonstrated remarkable effectiveness in discovering expressions with smaller roundoff errors than the initial expression.

Mixed-Precision Tuning Another optimization method is to increase efficiency or reduce resource cost while being inside an accuracy bound. Precision assignment focuses on this particular optimization.

A uniform precision can be assigned to all variables, constants, and operations in a program. However, this process may not be optimal in terms of performance. For instance, if a single precision is insufficient at a certain point in the program, with uniform precision, we need to upgrade *all* operations to a higher precision to guarantee the error bound. However, not all operations have the same impact on the overall accuracy of the program.

Therefore, assigning different precisions to different operations can be more efficient to achieve a target error bound while reducing the cost of resources. This approach, known as *mixed precision* optimization, allows for a more flexible and efficient program implementation.

Various techniques exist for mixed-precision tuning of numerical programs [39, 54, 49, 21]. Here, our primary focus is on the mixed-precision tuning incorporated in Daisy [54]. Daisy's mixed precision tuning works for both floating-point and fixed-point programs; we use the fixed-point precision tuning as a comparative reference in chapter 6.

Daisy [54] assigns different precisions to different program inputs, variables (including intermediate ones), and constants such that a user-defined error bound is met. For fixed-point programs, the process starts by assigning one bit and gradually increasing the number of bits uniformly for all variables until the error bound is met. Alternatively, for floating-point programs, Daisy initializes all variables with the highest available precision and gradually systematically reduces their precision until it becomes impossible to lower the precision further while maintaining the specified error bound.

In the next step, Daisy employs a heuristic search based on delta-debugging to assign precision to variables. It uses a static sound error analysis and a static (but heuristic) performance cost function to guide the search. It first divides the list of all variables into two equal-sized partitions and recursively attempts to reduce the precision of these individual partitions. The recursion halts when a precision assignment is discovered that satisfies the error bound.

Given that multiple valid type assignments could be feasible, Daisy employs the cost function to identify the assignment with the lowest cost for optimal performance. It supports several different cost functions. For instance, an *area-based* cost function for fixed-point programs essentially counts the number of bits assigned to all variables. For floating-point programs, a *simple abstract* cost function is used that assigns 1, 2, and 4 costs to single, double, and quad arithmetic and cast operations, respectively. There is also a *platform-specific* cost function that assigns distinct costs to each floating-point operation based on the execution time of individual operations performed in isolation.

Daisy's mixed-precision tuning demonstrates effectiveness for small numerical programs, especially those employing floating-point precision. In floating-point precision, there are limited precisions to select from, which contributes to the success of this approach. However, for fixed-point programs, due to the large choice (of combinations) of different precisions for individual operations, the search space is huge. Consequently, the computational cost associated with this heuristic search increases exponentially with the size of the program. Furthermore, for mixed-precision tuning, Daisy lacks support for complex data structures, such as matrices and vectors.

Discrete Choice with Numerical Uncertainties

3

State-of-the-art s finite-precision analysis of numerical programs primarily focuses on bounding absolute roundoff errors in the *worst-case*. These worst-case analyses compute, at each step of the computation, the largest error for a specified range of program inputs, propagate these errors throughout the program, and thus, provide a bound on the most significant roundoff error that could arise at the output. These analyses have efficiently been applied to multivariate straight-line programs [57, 88, 162] with some limited support for conditionals and loops [88, 89, 53].

However, none of these present techniques analyze finite-precision programs with discrete decisions, i.e., a decision that is taken based on the results of numerical computations. Recall our '`controller`' example with a discrete decision from Figure 1.1:

```
res = ...; if(res < 400) raise_alarm() else do_nothing()
```

Such programs play crucial roles in various systems where decision-making is required, such as determining whether to raise the alarm as our example `controller`, selecting among multiple discrete control signals, and more. Additionally, machine learning classifiers are now being used extensively in resource-constrained critical systems that decide if signals like heartbeat readings are normal or abnormal [32].

Worst-case analysis for these programs leads to overly pessimistic results. Due to inevitable roundoff errors, finite-precision executions will take different paths than the ideal real-valued computations for at least some of the inputs. Worst-case analysis would simply conclude that the program always takes the wrong branch, which is clearly not useful in practice.

Ideally, a program should make the correct decision for most inputs. Hence, in this case, a more meaningful measure would be to guarantee that the probability of wrong execution, which we refer to as the *wrong path probability*, is sufficiently low to ensure the safety of the application. As all inputs may not be equally likely, to accurately estimate the wrong path probability, it is essential to consider the probability distributions of the inputs and propagate them through the computation.

In this chapter, we aim to compute sound bounds on wrong path probabilities for programs with discrete decisions by considering the probability distributions of inputs. We explore two state-of-the-art techniques to achieve this: exact symbolic probabilistic inference [82] and sound probabilistic affine arithmetic [11]. These techniques enable us to propagate the probability distributions in an exact or over-approximated way. We utilized these techniques to compute wrong path probabilities for numerical programs from various domains, including embedded systems, scientific computing, and machine learning.

Our results show that exact probabilistic inference does not scale for the majority of our benchmarks. Although probabilistic affine arithmetic scales well, it fails to compute tight bounds on wrong path probabilities. To address this limitation, we extended the analysis with our novel interval subdivision and reachability checks. Our technique subdivides the input domain, identifies relevant subdivisions leading to incorrect program paths using reachability checks, and performs probabilistic analysis only on those relevant subdomains. Thus, our technique can *efficiently* compute accurate wrong path probabilities.

We have implemented the proposed technique as an extension to the state-of-the-art tool Daisy [55]. Our tool can fully automatically compute sound wrong path probabilities for floating-point and fixed-point programs with discrete decisions, considering uniform and gaussian distributions of inputs.

Contributions To summarize, this work makes the following contributions:

a) we provide a precise problem definition and present, to the best of our knowledge, the first sound analysis of the effects of numerical uncertainties on discrete decisions,

b) we extend an existing probabilistic static analysis with interval subdivision and reachability checks,

c) we evaluate our analysis and the various design decisions that we make on a set of representative embedded examples

d) we also show that while a combination of state-of-the-art symbolic probabilistic reasoning provides exact results, it does not scale well, and

e) we implement our analysis in a prototype tool implemented on top of the existing worst-case analysis tool Daisy which is publicly available on Zenodo (see `https://doi.org/10.5281/zenodo.8042198`).

The material in this chapter is based on the publication titled '*Discrete Choice in the Presence of Numerical Uncertainties*' [166] co-authored with Eva Darulova, Sylvie Putot, Eric Goubault. Eva Darulova contributed to the experiments using the symbolic inference technique and provided extensive feedback on the writing of the paper, while Sylvie Putot and Eric Goubault provided valuable feedback on the technical background.

3.1 Overview

In this section, we first describe the problem we are considering more precisely, introduce the terminologies we will use throughout the following chapters, and present a high-level overview of our approach using an illustrative example.

```
res = -x1*x2 - 2*x2*x3 - x1 - x3
if (res <= 0.0) raise_alarm()
```

Figure 3.1: Running example of a non-linear controller

3.1.1 Problem Description

We focus on the finite-precision numerical programs that make discrete decisions based on the computed numerical result. We define our input program, without the decision, as a real mathematical function $f : \mathbb{R}^n \to \mathbb{B}$, where $\mathbb{R}^n$ represents the multivariate real-valued input domain and $\mathbb{B}$ represents the boolean output domain. The function f includes arithmetic operations $(+, -, *, /, \sqrt{})$ as well as transcendental functions $(\sin, \cos, \exp, \log)$.

In the hardware implementation of this function, it is represented as a finite-precision function $\tilde{f} : \mathbb{F}^n \to \mathbb{B}$, where $\mathbb{F}^n$ represents the finite-precision input domain. Due to numerical uncertainties, such as input uncertainties and round-off errors in both the input and computation, the real function f and the finite-precision function $\tilde{f}$ may yield different results for the same set of ideal inputs.

Our goal is to automatically compute a *guaranteed bound* on the probability that f and $\tilde{f}$ return different results. We will call this probability the *wrong path probability* (WPP). This probability can never be zero because of the inevitable presence of numerical uncertainties. However, ensuring that the probability is small enough for the application to be useful in practice is important.

While our focus is primarily on straight-line numerical programs without conditionals or loops, our approach can be generalized to handle nested conditionals and other discrete return types by considering each path and return value separately.

We consider an example from [14], where an alarm can be raised based on the control output calculated by a nonlinear controller in Figure 3.1. Here, `x1`, `x2`, `x3` are the inputs of the controller and may carry some uncertainties from the source. We want to compute the WPP of this program.

3.1.2 Our Proposed Technique

We present an overview of our technique in Figure 3.2. It takes as input a finite-precision program with probabilistic inputs and computes the WPP. We first compute a bound on the numerical uncertainties and define an interval, which we refer to as a '*critical interval*,' where the program may make a wrong decision if the result falls within this interval. Next, we propagate the probability distributions of the input variables to the result. Finally, by computing the intersection between the critical interval and the probability distribution of the result, we determine the probability of the program taking the wrong path. In the following sections, we provide a detailed explanation of each of these parts of our solution.

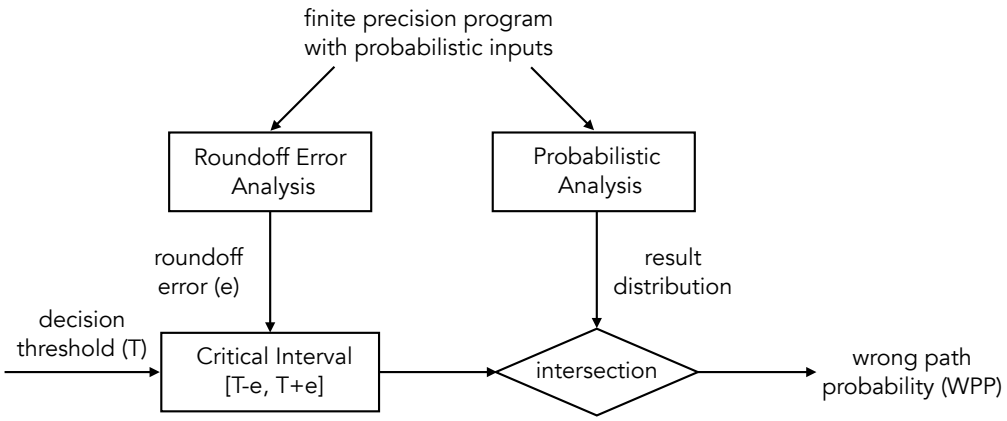

Figure 3.2: Overview of our approach

Bounding Numerical Uncertainties The numerical input program might take a wrong execution path in the presence of input uncertainties, the roundoff errors that occur in the input, and the roundoff errors that accumulate through numerical computation. Hence, we need to compute a bound e on the roundoff error that can occur in the worst case during the computation of a function f. Several techniques exist for this purpose, and in our approach, we utilize the methods established in [55] that are based on interval and affine arithmetic. The details of how we compute the roundoff error using these techniques are presented in Section 2.2.

In the context of our example program (see Figure 3.1), we consider the 16-bit fixed-point roundoff error as the numerical uncertainty. To determine a bound on this uncertainty, we utilized the Daisy and computed $e = 0.2042266$ as the worst case roundoff error.

Defining Critical Intervals The input program contains a discrete decision boundary where the ideal and uncertain computations may diverge. This boundary is determined by a single *threshold* value t. The numerical error (e) we computed in the previous step determines the maximum possible deviation from this threshold in the worst case. Using this error, we define the critical interval $[t - e, t + e]$. If the value of the computation falls into this interval, then the uncertain, finite-precision computation may compute a different discrete result. The probability that the ideal computation lies in $[t - e, t + e]$ is thus a sound bound on the wrong path probability.

In our running example, the decision threshold t is set to 0.0. The computed value of e is 0.2042266. Therefore, the critical interval defined for this program is $[-0.2042266, 0.2042266]$.

Propagating Probabilities Next, we proceed to propagate the input distributions through the numerical computation to compute a sound bound on the probability that the result of the numerical computation falls within the critical interval. We can express the wrong path probability precisely as a probabilistic program as in Figure 3.3, where we reuse a specific syntax proposed in [82]. We have added the roundoff error e with an *assert* statement in the computation. Moreover, we choose uniform distributions for inputs.

There can be two entirely different approaches for propagating probability distributions. One is using exact symbolic inference tools like PSI [82]. SI analyzes probabilistic programs using symbolic domains, yielding precise distributions of the results. We can then utilize

```
x1 := uniform(-15.0, 15.0);
x2 := uniform(-15.0, 15.0);
x3 := uniform(-15.0, 15.0);

res := -x1*x2 - 2*x2*x3 - x1 - x3;
error := 0.2042266; // computed externally
assert(0.0 - error <= res && res <= 0.0 + error);
```

Figure 3.3: Probabilistic program encoding the wrong path probability for Figure 3.1

numerical evaluators like Mathematica [102] to compute the wrong path probability (WPP). In our experiments, we employed PSI and Mathematica for our benchmarks, details of which are presented in Section 3.3.2. However, our experiments show that this method does not scale to larger programs. Even for our small running example in Figure 3.3, we did not obtain any results within 20 minutes.

The other alternative to using exact reasoning is an abstraction-based approach that uses probabilistic affine forms to propagate uncertain distributions. The high-level algorithm of probabilistic affine forms was introduced in the original work [30], which computes a sound over- and underapproximation of the probability distribution. However, neither the implementation details nor any public tool are available for this approach. We have reimplemented the analysis presented in [30]. Our experiments show that this analysis scales better than the symbolic inference approach. For example, the WPP of the running example in Figure 3.3 was computed in 0.63 seconds.

However, this method suffers from high overapproximations. As a result, the WPPs computed for benchmark programs become trivial and not practically useful. For instance, in the case of our running example, the computed WPP is 1.0, which does not provide any meaningful information in practice.

Extending Probabilistic Analysis for Computing Wrong Path Probability The reason why the above approaches do not work well is that they compute general probability distributions, but in our case, we are interested in a specific probability value – the probability of the result being inside the relatively small critical interval. However, it is not immediately obvious which inputs *do* lead the program execution to compute the result inside the critical interval, as most of the input space does not come close to this interval. Ideally, we could perform a backward reachability analysis starting from the critical interval to obtain only the relevant input domain. Unfortunately, computing such a sound input domain is nontrivial for multivariate programs.

We combine probabilistic static analysis with interval subdivision, a standard technique used in static analyses to reduce overapproximations. While it can improve the probability bounds by itself, performing probabilistic analysis for all subdomains can be too expensive to be useful in practice. To improve scalability, we further use a nonlinear decision procedure in a forward manner to check the reachability of the input subdomains and altogether remove parts of them that definitely cannot reach the critical interval. This process narrows down the input domain for

the probabilistic analysis and thus improves scalability. For our running example in Figure 3.3, our method computes a wrong path probability of 0.07060 in 155 seconds.

In summary, we obtain a technique that can compute fairly tight probability bounds (judging from the few examples where PSI computes a result), and which is nonetheless scalable enough to handle programs which appear in embedded systems in practice and for which guaranteed results are of importance. In the following section, we present our technique in detail.

3.2 Probabilistic Static Analysis

To accurately propagate the uncertainties of the inputs, we opt to use probabilistic affine arithmetic that maintains the dependencies between variables in the computation. The detailed description of probabilistic affine forms and the arithmetic is presented in the background section of this thesis (see Section 2.3.2). However, since no existing tool was available for this approach, we took the initiative to re-implement it from scratch.

Our re-implementation involved making several design decisions to ensure the analysis was useful in practice. We adapted the probabilistic affine forms for our purpose and incorporated several crucial design decisions. In this section, we first describe our adaptations and the key design decisions we made. Later, we present our extension, which involves performing interval subdivisions and reachability checks to enhance the analysis further. These adaptations and extensions are essential to achieve an efficient and scalable approach for computing the wrong path probability (WPP).

3.2.1 Our Adaption of Probabilistic Affine Forms

In the original presentation [30] of the probabilistic affine form (PAA), a variable x is equipped with noise symbols (d) that captures the probability distributions along with (linear) dependency information in terms of Dempster-Shafer Intervals (DSIs) with support $[-1, 1]$. More specifically, it tracks two types of noise terms: one representing independent ones and one with an undefined dependency. We show the original representation in Equation 2.11.

However, in our work presented in this chapter, we have chosen not to make a distinction between two noise symbols. Instead, we keep a set of indices for each noise symbol, indicating the potential dependencies on other noise terms, and use only one noise symbol (d_η) for the entire computation:

$$\hat{x} = x_0 + \sum_{i=1}^{k} x_i d_{\eta_i}$$

Here, k represents the total number of noise terms. Depending on whether a d_{η_i} has a dependency on the current running sum, the addition operation is considered either dependent or independent.

Independent arithmetic is straightforward; however, for dependent arithmetic, we need to

solve a linear programming (LP) problem similar to the original work. The details of these arithmetic operations are discussed in Section 2.3.2.

3.2.2 Design Decisions for the Implementation

In order to use probabilistic AA to propagate input distributions, the input distributions need to be first transformed into Dempster Shafer Intervals (DSIs). We provide convenient methods to discretize the distributions. Our implementation focuses explicitly on the uniform and normal distributions; however, it should be straightforward to extend it to other distributions. The overall accuracy of the computed distributions clearly depends on the amount of discretization: a finer discretization into more focal elements increases the accuracy of the computed probability distributions, but it also naturally increases the analysis time.

Furthermore, the results depend on several factors, including the algorithm used to reduce the number of focal elements of the Dempster-Shafer Intervals (DSIs), the translation of probabilistic AA back into a DSI, the choice of LP solver, and the soundness of internal computations. In this section, we will describe each of these factors in detail.

Reduction Algorithm With each arithmetic operation, the number of DSI focal elements grows exponentially and can soon become a performance bottleneck. We thus need to keep the length of the DSIs bounded. We present a reduction method that, after each arithmetic operation, checks the length of the DSI. If the length is larger than a user-specified threshold, it combines focal elements while retaining as much information as possible. The idea of this combination is to repeatedly merge two focal elements, $e_i = \langle [a_i, b_i], w_i \rangle$ and $e_j = \langle [a_j, b_j], w_j \rangle$, producing one focal element with a wider interval and larger weight:

$$e_{ij} = \langle [\min(a_i, a_j), \max(b_i, b_j)], w_i + w_j \rangle$$

The decision of whether or not two focal elements should be merged depends on their weights and interval widths. Furthermore, our algorithm only merges two focal elements when their intervals overlap in order to reduce the loss of accuracy. We show the reduction method in Algorithm 1.

The algorithm takes as input the DSI `d` and a threshold `T` that bounds the length of `d`. At first, it merges two overlapping focal elements when the weight of the second is less than a small value of `1e-5` (line 2 - 4). When this does not reduce the number of focal elements sufficiently, i.e., less than the threshold `T` (line 5), we merge all focal elements whose widths or weights are less than one standard deviation from the average (lines 6-10 and 12-15). Additionally, this merging is only applied when the standard deviation is sufficiently large (`1e-3` for the interval widths and `1e-4` for the weights). Without this check, should all focal elements have identical width (resp. weight), they would all be merged into a single focal element. It may be noted that all these numerical thresholds are determined empirically and could be updated easily based on the user's choice of the DSI length. If the number of focal elements in the DSI is still more than `T`,

Algorithm 1 Reduction algorithm

```
1:  procedure REDUCE(DSI d, threshold T)
2:      for e_i : (x_i, w_i) ← d do
3:          if w_i < 1e − 5 && overlaps(x_i, x_{i−1}) then
4:              merge(e_i, e_{i−1})                          ▷ merging low weights
5:      if length(d) > T then
6:          avg = compute_avg_width(d), sd= compute_std_dev_width(d)
7:          if sd > 1e − 3 then                              ▷ variation in widths
8:              for e_i : (x_i, w_i) ← d do
9:                  if width(x_i) < (avg − sd) && overlaps(x_i, x_{i−1}) then
10:                     merge(e_i, e_{i−1})
11:     avg = compute_avg_weight(d), sd = compute_std_dev_weight(d)
12:     if sd > 1e − 4 then                                  ▷ variation in weights
13:         for e_i : (x_i, w_i) ← d do
14:             if w_i < (avg − sd) && overlaps(x_i, x_{i−1}) then
15:                 merge(e_i, e_{i−1})
16:     if length(d) > T then
17:         reduce_alternate_overlapping(d)
18:     return d
```

we merge every two overlapping focal elements, irrespective of weight and interval width (lines 16-17). This algorithm in 1 is repeatedly applied until the size of the DSI is sufficiently reduced.

Converting AA to DSI After performing all arithmetic operations, the distribution of the result is represented as a probabilistic affine form. In order to compute the probability of the value of the result being inside the critical interval, the affine form needs to be converted to a DSI representation(a set of intervals and weights). This conversion requires the affine terms to be summed up using the dependency information collected by the affine form. For this purpose, we implement a greedy strategy to maximize independent arithmetic operations and reduce costlier dependent operations as much as possible. Our algorithm first partitions the affine terms into sets where each set contains mutually independent terms. These terms are added using independent addition. The resulting mutually dependent partial sums are finally added (in any order) using the dependent addition.

Floating-point Arithmetic To ensure the soundness of the probabilistic analysis, we implement our prototype using rationals, i.e., infinite-precision integers. Thus, the prototype does not suffer from internal roundoff errors; however, it naturally increases the running time of the analysis. For operations that cannot be computed in rationals, e.g., square root, we use the MPFR arbitrary-precision arithmetic library [74] and ensure that the result is rounded correctly to ensure overapproximated sound bounds.

To solve the maximization and minimization problems of the linear programs for the dependent arithmetic, we utilize an external off-the-shelf solver GLPK [7]. GLPK is implemented with floating-point arithmetic internally and is generally efficient and precise. However, we cannot guarantee the results – it does occasionally generate solutions with extraneous focal

Algorithm 2 Interval subdivision with reachability check

```
1:  procedure COMPUTE_WPP(expr e, env E, dist d, divLim L, discretLim X, criticalIntvl cI)
2:      numDiv = pow(L, 1/len(E))                          ▷ distributing L
3:      subDom = cartesian_subdiv(E, numDiv, d)
4:      ρ = 0                                              ▷ total WPP
5:      for (dom, ρ_dom) ← subDom do
6:          if cheack_reachable(e, dom, cI) then
7:              outDSI = prob_analysis(e, dom, X)
8:              ρ_loc = intersect(cI, outDSI)
9:          else
10:             ρ_loc = 0                                  ▷ critical interval not reachable
11:         ρ = ρ + ρ_dom × ρ_loc                          ▷ add local WPP
12:     return ρ
```

elements with weights on the order of double floating-point roundoff errors. An entirely satisfactory solution here would be to use a guaranteed LP solver, such as Lurupa [115] that uses interval arithmetic internally and guarantees the results even in the presence of roundoff errors or LPex [65], an exact solver, that certifies the result. However, the effects of such solvers on analysis performance are uncertain. Given that the probabilities we compute are many orders of magnitude larger than double-precision floating-point roundoff errors, we have kept GLPK as our solver.

3.2.3 Our Extension: Interval Subdivision + Reachability Checks

Using only probabilistic AA, we can compute a sound overapproximation of the wrong path probability (WPP). However, our experimental results show that for many of our benchmarks, the WPP computed is 1 or close to 1 (see the column marked 'A' in Table 3.3 in the 'Experimental Evaluation' section). Because of such high overapproximations, probabilistic static analysis alone is not enough for our purpose. To reduce the overapproximation, we propose to subdivide the 'outer' input interval into multiple subdomains (the user can specify the total number of subdivisions) and to perform a reachability check on each subdomain before it is passed to the probabilistic static analysis. The overall method is shown in Algorithm 2.

The algorithm takes as input an arithmetic expression e, an environment E mapping variables to input intervals, the assumed distribution of inputs d, a limit on outer subdivisions L, a limit on the discretization of input distributions X and the critical interval cI.

The algorithm first distributes the outer subdivision limit L among the input variables (line 2) and then subdivides each input variable's interval $numDiv$ times such that we obtain overall at most L subdomains (line 3).

For each subdomain dom, the algorithm then checks whether the critical interval is reachable (line 6). This check is performed by encoding reachability as an SMT query and discharged using Z3 [63]. We set a timeout of 1s; if the SMT solver times out, we assume the critical interval

is reachable, thus ensuring soundness. If the critical interval is potentially reachable, we perform probabilistic analysis, obtaining a sound overapproximation of the output distribution as a DSI (line 7). From this, we obtain the wrong path probability ρ_{loc} for the subdomain dom by summing up the probabilities of the $outDSI$'s focal elements, which intersect with the critical interval (line 8). If the SMT solver determines that the critical interval is not reachable, from dom, the wrong path probability of the particular sbdomain is $\rho_{loc} = 0$.

Finally, the algorithm returns the overall WPP as the weighted sum $\rho = \sum \rho_{dom,i} \times \rho_{loc,i}$, where the weights ρ_{dom} depend on the input distribution. Our implementation currently supports normal and uniform distributions; e.g. for the latter, all ρ_{dom}'s are the same.

It may be noted that interval subdivisions with reachability checks together without probabilistic analysis can also be used to compute the wrong path probability by adding the probabilities of the input subdomains from which the critical interval is reachable. This method can be efficient and precise in some cases; however, our experiments in Table 3.3 show that the proposed combination of probabilistic analysis and subdivisions with reachability checks overall computes the tightest wrong path probability for the benchmarks.

Running Example Recall our example from Figure 3.3 which takes three inputs $x1, x2, x3$ uniformly distributed in $[-15.0, 15.0]$. Choosing $L = 8000$, our algorithm subdivides each input range into 20 pieces, each with equal weight. For this example, the reachability checks determine that for 87% of the subdomains, the critical interval is unreachable, and thus, for these, the probability analysis is skipped. For the remaining 13%, probabilistic analysis computes the total WPP as 0.07060. The WPP using only subdivision and reachability checks and no probabilistic analysis would be 0.07992. □

3.3 Experimental Evaluation

In this section, we extensively evaluate our proposed probabilistic analysis to compute the wrong path probabilities of a set of benchmarks we present next. Here, we also show how the existing exact symbolic inference techniques can be used to compute the wrong path probabilities and the effectiveness of the approach for our purpose.

3.3.1 Benchmarks

As no standard benchmark set exists, we retrofit a number of existing benchmarks from the area of finite-precision roundoff error estimation [161, 53]. Each of these benchmarks comes with lower and upper bounds on input variables. These benchmarks cover various scenarios from diverse domains:

- **`Polynomial approximations of math functions:`** We collected 3 benchmarks that compute polynomial approximations of mathematical functions that are often costly to compute precisely. The benchmark `sine` implements seventh-order and `sineOrder3` implements

third-order approximation of Taylor's series of the sine function, and the benchmark `sqroot` computes an approximation of a variable's square root.

- **`Physics expressions:`** The benchmark `doppler` computes the change in wave frequency due to the Doppler effect, and `turbine` implements 3 non-linear equations that model the stress on a turbine rotor. We divided these 3 equations into 3 benchmarks – `turbine1`, `turbine2`, and `turbine3`.

- **`Embedded controllers:`** `traincar`, taken from [14], models a simple pulling train car with the objective of quickly reducing the possible oscillations between the locomotive and the car. We have unrolled the linear control loop up to 4 iterations and generated 4 benchmarks – `traincar1`, `traincar2`, `traincar3`, and `traincar4`. Additionally, we have the `rigidBody` benchmarks from the same source, computing the angular momentum for a rigid spacecraft with two controls.

- **`Embedded image processing applications:`** `bsplines` are used in embedded image applications which we adapted from [124]. There are 4 basic bspline functions that we considered into 4 benchmarks – `bspline0`, `bspline1`, `bspline2`, and `bspline3`.

Table 3.2 shows the number of arithmetic operations and variables of each benchmark. We consider all input variables to be either uniformly or all normally distributed within the input ranges. For variable x, which is uniformly distributed on $[a, b]$, we declare it as `x := uniform(a, b)`. For a normally distributed variable, we choose one standard deviation from the mean and declare it as the following program:

```
tmp := gauss(a, b);
observe(-1.0 <= tmp && tmp <= 1.0);
x := mid + radius * tmp;
```

with mid $= (a+b)/2$ and radius $= |(a-b)/2|$.

For each benchmark, we select two thresholds: *low*, where we expect the wrong path probability to be low, and *high*, where we expect the probability to be high. We generate these thresholds by simulation. From a plot of the recorded results, we choose one threshold from the more and one from the less likely result region.

3.3.2 Using Symbolic Inference

As mentioned in Section 3.1.2, propagating probability distributions is possible using exact symbolic inference tools. In this section, we attempt to compute the wrong path probabilities of our benchmarks that are expressed as probabilistic programs (as shown in Figure 3.3) using a state-of-the-art probabilistic programming inference tool.

Several approaches to probabilistic inference exist. As exact probabilistic inference is a hard problem [46], most algorithms compute numerical approximations using sampling [136, 35, 143].

Such methods, however, do not provide accuracy guarantees. The recently developed PSI (Probabilistic Symbolic Inference) solver [82] performs *symbolic* inference and generates a compact representation of the *exact* final distribution. This distribution is, in general, still quite complex and large, but it can be numerically evaluated using Mathematica [102], for which we can specify an output precision.

Experimental Setup We empirically evaluate to which extent exact probabilistic inference is suitable for computing wrong path probabilities. For these experiments, we consider roundoff errors as the only uncertainty, i.e., we assume that the inputs are otherwise exact. For this, we combine three off-the-shelf tools.

- **Roundoff error analysis with Daisy:** In the first step, we use the open-source tool Daisy [55] to compute a sound absolute roundoff error bound on the numerical part of each benchmark. Daisy performs forward data-flow analysis and supports both floating-point and fixed-point arithmetic and can also be used to propagate input errors.

  We used Daisy and computed roundoff errors for 16-bit fixed-point arithmetic as well as 32-bit (single precision) floating-point arithmetic and recorded the roundoff error as a constant in the probabilistic program. The computation of roundoff errors for all benchmarks for one precision took under 30 seconds.

- **Symbolic probabilistic inference with PSI:** In the second step, we run PSI on the probabilistic programs, each of the same forms as in Figure 3.3. We choose all inputs to be either uniformly or normally distributed on the same support for which we have computed roundoff errors. All inputs are independent. PSI computes the output and the probability distribution of the assertion failing; we record the latter for our experiment. We set a timeout of 10 minutes (we did not observe different results with longer timeouts).

- **Numerical evaluation with Mathematica:** PSI returns the exact simplified expression of the wrong path probability in the input format of Mathematica. This expression can then be numerically evaluated using the `N[...]` command. We set the output precision to be five decimal digits (Mathematica adjusts the internal precision automatically) and a timeout of 10 minutes for Mathematica to compute the wrong path probabilities.

We used PSI downloaded on 22 March 2018, Mathematica version 10.4.0.0 and Daisy's version from 6 December 2017. We ran all experiments on a Debian desktop with 3.3 GHz and 32 GB RAM.

Results and Discussion Table 3.1 shows the wrong path probabilities for benchmarks where both PSI and Mathematica finished within the time limit. For `doppler`, `rigidBody`, `turbine`, and `traincar`, either PSI or Mathematica timed out so that we do not show them in the table.

Table 3.2 shows the real-time (measured by the bash time command) taken by PSI to compute the exact probability distribution. We also measured the time taken by Mathematica (using the AbsoluteTiming command); however, we observed that either Mathematica reported a time

benchmarks	16-bit fixed-point				32-bit floating-point			
	normal		uniform		normal		uniform	
	high	low	high	low	high	low	high	low
`sine`	8.12e-4	4.55e-4	9.55e-4	2.98e-4	6.47e-7	3.63e-7	7.61e-7	2.38e-7
`sineOrder3`	2.14e-3	5.24e-4	2.47e-3	4.49e-4	1.64e-6	4.03e-7	1.90e-6	3.44e-7
`sqroot`	3.07e-2	9.59e-4	2.79e-2	1.10e-3	9.62e-6	3.03e-7	8.74e-6	3.49e-7
`bspline0`	1.73e-2	6.05e-4	1.82e-2	7.23e-4	1.00e-5	3.53e-7	1.05e-5	4.21e-7
`bspline1`	3.26e-3	1.48e-3	3.46e-3	1.59e-3	2.39e-6	1.08e-6	2.54e-6	1.16e-6
`bspline2`	2.78e-3	1.26e-3	2.95e-3	1.32e-3	2.09e-6	9.46e-7	2.22e-6	9.95e-7
`bspline3`	2.78e-3	3.58e-4	2.30e-3	4.27e-4	1.70e-6	2.32e-7	1.49e-6	2.77e-7

Table 3.1: Wrong path probability computed by PSI (not shown benchmarks did not complete within 20 min)

below 1 ms or it timed out after 10 minutes. For this reason, we only report PSI's running times. The benchmarks for which PSI successfully computes results (as reported in Table 3.2) but do not appear in Table 3.1, Mathematica timed out.

Where PSI and Mathematica are able to compute wrong path probabilities, they indeed compute a smaller probability for lower thresholds than for higher ones. Similarly, for smaller uncertainties, i.e., 32-bit floating-point roundoff errors, the probabilities are also significantly smaller, as expected. While the values for uniform and normal input distributions are not directly comparable, we have included them as different distributions pose different difficulties for inference. This can be seen in Table 3.2 where PSI can compute a result for `rigidBody1` for normal and for `turbine2` for uniform input distributions only.

In conclusion, our results confirm that the symbolic inference-based approach is capable of computing precise WPP, but it does not scale for the majority of benchmark programs.

3.3.3 Using Probabilistic AA + Our Extension

In this section, we extensively evaluate our proposed analysis presented in Algorithm 2. Our algorithm is conceptually simple but very effective for computing tight wrong path probabilities. Nonetheless, it has several parameters which need to be specified. In particular, it performs two types of subdivisions, one for discretizing the DSIs and another for subdividing the input intervals. To determine suitable parameters, we conduct a systematic empirical exploration, presenting a subset of the results in this section and comparing them in terms of running time.

We identify one setting that allows for a fully automated analysis and performs comparatively better among other settings, both in terms of wrong path probability computation and running time. This setting is used for our next comparison of low and high thresholds. Subsequently, we

benchmarks	#ops	#vars	normal distribution (s)	uniform distribution (s)
doppler	8	3	*	*
sine	18	1	1.301	0.688
sineOrder3	5	1	0.437	0.523
sqroot	14	1	0.565	0.767
bspline0	6	1	0.729	0.354
bspline1	8	1	0.690	0.303
bspline2	10	1	0.729	0.356
bspline3	4	1	0.716	0.994
rigidBody1	7	3	121.510	-
rigidBody2	14	3	-	-
turbine1	14	3	-	-
turbine2	10	3	-	106.348
turbine3	14	3	-	-
traincar1	6	3	15.553	15.330
traincar2	10	5	353.004	352.947
traincar3	14	7	355.616	353.52
traincar4	18	9	-	-

Table 3.2: Average analysis time of PSI (- : timeout, * : segfault)

employ the same setting for further evaluations on additional benchmarks in Section 3.3.4. All these experiments are conducted using the same Debian system as used for PSI.

3.3.3.1 *Evaluation of Parameter Settings*

We define the following different subdivision strategies that we denote with the letters A-E.

- **A, probabilistic analysis only:** In this setting, we do not subdivide the outer interval, i.e., $L = 0$. The probabilistic analysis discretizes each initial variable distribution into DSIs with 25 focal elements. We have observed empirically that discretizing the DSIs further extensively increases the running time of the analysis without quite improving the results. Hence, we used 25 as the initial discretization of the DSI; however, the number of focal elements can increase up to 100 during computation.
- **B, non-probabilistic analysis:** Here, we only subdivide the outer interval and do not apply any probabilistic analysis. If the critical interval is reachable, as determined using the SMT solver, we assign 1 to the local probability and 0 otherwise.
- **C, outer subdivision and probabilistic analysis with coarse discretization:** In this setting, we apply both discretization and outer subdivision but choose the focus on the latter, i.e.,

settings	A	B	C	D	E
# DSI subdiv	25	0	2	4	4
# outer subdiv	0	32000	16000	8000	8000 (deriv.)
doppler	1.00e+0	4.68e-2	3.06e-2	2.17e-2	**2.16e-2**
sine	3.24e-1	**3.13e-5**	5.76e-5	6.45e-5	6.45e-5
sineOrder3	3.58e-1	**6.25e-5**	1.17e-4	1.23e-4	1.23e-4
sqroot	1.00e+0	**3.13e-5**	6.25e-5	9.38e-5	9.38e-5
bspline0	1.00e+0	**3.13e-5**	5.47e-5	6.06e-5	6.06e-5
bspline1	9.60e-1	6.25e-5	3.13e-5	**1.95e-5**	**1.95e-5**
bspline2	9.16e-1	**6.25e-5**	1.23e-4	1.72e-4	1.72e-4
bspline3	1.00e+0	3.13e-5	3.13e-5	**7.81e-6**	**7.81e-6**
rigidBody1	1.00e+0	7.99e-2	7.11e-2	7.06e-2	**7.05e-2**
rigidBody2	1.00e+0	1.02e-1	1.05e-1	**9.23e-2**	1.07e-1
turbine1	1.00e+0	5.75e-2	5.77e-2	**4.82e-2**	5.25e-2
turbine2	1.00e+0	5.28e-2	5.44e-2	**4.64e-2**	4.91e-2
turbine3	1.00e+0	6.97e-2	6.39e-2	**5.39e-2**	5.73e-2
traincar1	9.83e-2	4.81e-2	2.98e-2	**1.86e-2**	2.47e-2
traincar2	1.14e-1	2.96e-1	1.74e-1	**9.17e-2**	1.23e-1
traincar3	**1.33e-1**	5.37e-1	3.66e-1	1.97e-1	**1.33e-1**
traincar4	**2.04e-1**	8.55e-1	7.13e-1	4.20e-1	2.83e-1

Table 3.3: Wrong path probabilities computed with different settings for 32-bit floating-point roundoff errors as uncertainty, uniform distributions, and a high threshold as the critical interval

we choose a large $L = 16000$ and a small number of focal elements (2 per variable DSI) for the initial discretization.

- **D, outer subdivision and probabilistic analysis with finer discretization:** As in setting C, we use both kinds of subdivisions but use a smaller $L = 8000$, and 4 initial focal elements (per variable DSI), overall maintaining 32000 divisions.

- **E, derivative guided outer subdivision:** For settings B - D, we subdivide each input variable equally (outer subdivision column in the table). However, some variables may influence the result more or less. Based on this idea, we used the magnitude of the derivative with respect to each input variable to decide which variable's interval should be subdivided more: variables with larger derivatives are subdivided more, keeping the entire subdivisions under 32000.

Table 3.3 presents the wrong path probabilities computed, and Table 3.4 compares the running time averaged over 3 runs for the different strategies. The row '# DSI subdiv' gives the number of

settings	A	B	C	D	E
#DSI subdiv	25	0	2	4	4
#outer subdiv	0	32000	16000	8000	8000 (deriv.)
`doppler`	14.350	187.080	136.750	184.340	188.300
`sine`	3.540	279.760	201.480	75.50	74.630
`sineOrder3`	0.270	195.140	90.900	49.380	49.200
`sqroot`	79.510	212.920	107.280	54.350	53.880
`bspline0`	0.760	207.970	104.77	51.088	51.086
`bspline1`	3.950	195.920	98.780	49.160	49.80
`bspline2`	3.020	196.290	98.620	49.600	49.520
`bspline3`	0.760	189.350	94.085	47.510	47.470
`rigidBody1`	0.650	189.260	116.032	154.540	147.830
`rigidBody2`	1.260	234.950	144.070	138.850	241.038
`turbine1`	65.240	250.590	323.70	375.170	424.580
`turbine2`	17.810	223.600	215.740	258.420	250.760
`turbine3`	49.150	238.160	356.710	447.970	523.220
`traincar1`	0.210	184.950	99.150	58.290	59.780
`traincar2`	1.240	112.280	97.430	301.020	384.860
`traincar3`	1.230	109.720	57.820	186.310	460.530
`traincar4`	1.320	132.270	21.320	73.900	650.280

Table 3.4: Analysis times in seconds (averaged over 3 runs) of different settings

focal elements of each input variable distribution, and '# outer subdivision' the maximum value of L in 2. To ensure a fair comparison, we choose the number of outer and DSI subdivisions such that the total number of divisions is at most 32000. We also kept a limit on the number of DSIs during computation, which is 100, to keep the analysis scalable.

The results in Table 3.3 confirm that probabilistic analysis alone (setting A) is not precise to compute wrong path probability except for the linear `traincar` controllers, where this setting does provide the best results. The reason behind this high overapproximation is that the intervals of the focal elements tend to be wide, and thus, too many intersect with the critical interval. In general, outer subdivision alone (setting B) provides much better results than setting A, i.e., tighter wrong path probabilities. We observe that this setting performs the best for univariate functions (`sine`, `sineOrder3`, `sqroot`, `bspline0`, `bspline2`) to provide reasonable wrong path probabilities. The combination of outer and coarse DSI subdivision (setting C) provides overall decent results but not the best. The combination with finer DSI subdivision (setting D) provides the best results for 8 out of 17 benchmarks. Somewhat surprisingly, derivative guided outer subdivision (setting E) provides a (small) benefit only for 3 benchmarks (`doppler`,

benchmarks	high threshold	low threshold	benchmarks	high threshold	low threshold
doppler	2.16e-2	4.93e-3	rigidBody2	9.23e-2	6.10e-2
sine	6.45e-5	6.25e-5	turbine1	4.82e-2	4.47e-3
sineOrder3	1.23e-4	6.25e-5	turbine2	4.64e-2	3.35e-3
sqroot	9.38e-5	7.62e-5	turbine3	5.39e-2	2.08e-3
bspline0	6.05e-5	6.64e-5	traincar1	1.86e-2	2.61e-3
bspline1	1.95e-5	6.05e-5	traincar2	9.17e-2	3.51e-3
bspline2	1.72e-4	5.66e-5	traincar3	1.97e-1	7.29e-4
bspline3	7.81e-6	6.64e-5	traincar4	4.20e-1	7.19e-3
rigidBody1	7.06e-2	6.28e-3			

Table 3.5: Wrong path probabilities for high and low thresholds (setting D)

rigidBody1, traincar3) compared to setting D. We observe that a combination of outer and DSI subdivisions provides the best results, with setting D being better overall.

We note that for those benchmarks, where PSI computes a probability *using exact inference*, the results with our static analysis method are quite close, confirming that the overapproximations due to static analysis are acceptable.

The results in Table 3.4 show that the imprecise probabilistic analysis (setting A) is the least expensive for all benchmarks and the non-probabilistic analysis (setting B) the most expensive for 15 out of 17 benchmarks in terms of running time. The combinations of these two (settings C, D, E) provide a tradeoff between the precision of the computed wrong path and the running time of the analysis. Setting D and E are pretty close, D being faster for benchmarks with more input variables (traincars with 3, 5, 7, and 9 inputs). Hence, we utilize setting D for our experiments in the following sections.

It may be noted that we keep the running times below 10 minutes (except for traincar4 where the analysis using setting E takes 650.28 seconds which is a bit more than 10 minutes and 50 seconds), which is comparable to the 10-minute timeout for PSI.

Low vs High Thresholds To evaluate the precision of the probabilistic analysis with setting D, we use two thresholds: one high and one low, as we did with PSI (Table 3.1) and compare the results in Table 3.5. With the high threshold, we expect to compute a larger wrong path probability than what we compute with the low threshold. This is indeed the case for 14 out of 17 benchmarks. However, we observe that our static analysis method is not able to detect the low probability threshold as consistently as PSI. This is because our method computes an overapproximation rather than computing the distributions precisely, as in PSI, to keep the analysis scalable even for complex distributions.

3.3.4 Further Evaluation

In this section, we perform additional experiments with setting D, the overall most successful setting found in terms of WPP and analysis time presented in Table 3.3 and Table 3.4. We compare the results for different-sized uncertainties for uniform and normal distributions, as well as independent and dependent inputs. Along with a subset of the previous benchmarks presented in Section 3.3.1, we choose several additional benchmarks for these evaluations where studying discrete choices is particularly relevant, which we describe next in detail.

- **`filter:`** We unroll 3 and 4 times a linear 2nd order numerical filter taken from [11], that appears frequently in embedded software to obtain benchmarks `filter3` and `filter4` with 15 and 25 arithmetic operations and 3 and 4 input variables, respectively.

- **`solveCubic:`** This benchmark, obtained from GNU Scientific Library, has been used in previous work [179] in the context of test case generation for floating-point programs with branches. The benchmark has 14 operations and 3 variables.

- **`classID:`** We trained a Linear Support Vector Classifier using the Python sklearn library on the Iris standard data set, which comes with sklearn, and extracted source code from the classifier using sklearn-porter [8]. For our benchmark, we have inlined the weights computed, and the three versions of this benchmark (`ID0`, `ID1`, `ID2`) correspond to the three decision variables, respectively classes, in the data set. They each have 15 arithmetic operations and 4 input variables.

- **`neuron:`** This is a simplified version of the DNN, which is supposed to have learned the 'AND' operator [145]. This neural network has one hidden layer of size 2 and six weights defining them. We consider here a one-input version, keeping one of the inputs constant. This benchmark uses the exponential function and has 22 arithmetic operations.

Different Sized Uncertainties In this first experiment, we compare the wrong path probabilities computed using our method for different-sized critical intervals: obtained by computing roundoff errors for 16-bit fixed-point and 32-bit floating-point arithmetic without input uncertainties, as well as assuming an input uncertainty of 0.001 (and 32-bit floating-points). All errors have been computed using the Daisy tool. The four missing entries in the fixed-point arithmetic column are due to overflows in the 16-bit implementation, and for the neuron benchmarks, an input error of 0.001 leads to a potential division by zero.

Table 3.6 shows the results. We observe that while the wrong path probabilities are all relatively small, our method is not able to distinguish in a notable way, except for the `bspline` benchmark, between different-sized critical intervals. This is, as before, with the low and high thresholds, a fundamental limitation of our abstraction-based method that computes an overapproximation of the wrong path probabilities, which we show here for completeness.

benchmarks	no uncertainty		input uncertainty.: 0.001
	16-bit fixed	32-bit float	32-bit float
bspline2	1.38e-3	5.66e-5	6.08e-3
rigidBody1	6.35e-3	6.28e-3	6.32e-3
rigidBody2	-	5.05e-2	5.22e-2
traincar1	2.72e-3	2.61e-3	2.66e-3
traincar2	3.53e-3	3.51e-3	3.53e-3
traincar3	-	7.29e-4	7.36e-4
traincar4	-	7.19e-3	7.19e-3
filter3	3.51e-3	3.23e-3	3.31e-3
filter	6.81e-4	6.72e-4	6.74e-4
solveCubic	3.32e-5	3.13e-5	3.32e-5
classID0	5.79e-2	5.70e-2	5.76e-2
classID1	5.79e-2	5.70e-2	5.74e-2
classID2	9.50e-2	9.37e-2	9.44e-2
neuron	-	1.38e-1	-

Table 3.6: Wrong path probabilities for different critical intervals

Uniform vs Gaussian Inputs In this section, we use our analysis to compare how the wrong path probability changes with the input distributions. For this, we run our analysis with all inputs uniformly and normally distributed but keeping the thresholds, i.e., critical intervals, constant.

Table 3.7 shows the results. As expected, the wrong path probabilities can differ substantially. This result confirms the importance of taking input distributions into account while doing the analysis. We also show the running times, as we have observed differences – with normal distribution, the running time is generally much higher than the uniform distribution. We suspect that these differences are due to the reduction method, which will have different effects for different input distributions.

Dependent Inputs Until now, we have always assumed that all inputs were independent and dependencies were only introduced by arithmetic operations. In this experiment, we look at the wrong path probability computed with all inputs being either independent or dependent (with unknown dependency).

Table 3.8 shows the results. We observe that, in general, the wrong path probabilities computed are greater for the dependent case. This is expected as we have to consider all possible dependencies. The difference is, however, not too large, and the results are still useful.

Curiously, we also observe that the running times of our method are *smaller* for the dependent case, even though, in this case, we are calling the LP solver for every arithmetic operation. But

benchmarks	wrong path probability		time (in seconds)	
	uniform	normal	uniform	normal
bspline2	5.66e-5	6.43e-5	53.30	59.86
rigidBody1	6.28e-3	4.39e-3	69.20	153.81
rigidBody2	5.05e-2	5.10e-2	195.46	595.88
traincar1	2.61e-3	1.69e-3	52.02	51.89
traincar2	3.51e-3	2.10e-3	68.07	74.67
traincar3	7.29e-4	4.22e-4	25.01	35.00
traincar4	7.19e-3	5.49e-3	20.70	42.13
filter3	3.23e-3	1.97e-3	54.54	54.46
filter4	6.72e-4	3.17e-4	47.14	48.94
solveCubic	3.13e-5	1.30e-5	58.00	147.38
classID0	5.70e-2	6.37e-2	331.13	609.52
classID1	5.70e-2	6.49e-2	293.04	532.12
classID2	9.37e-2	0.100	383.73	661.24
neuron	0.138	0.160	16.34	16.31

Table 3.7: Uniformly vs. normally distributed inputs

this is also the reason for the shorter running times; the GLPK solver we use is implemented in floating-point arithmetic, whose results we translate to rationals at the interface to our implementation. Calling the solver often effectively does not allow our rationals to grow too large, thus keeping the running time low.

3.3.5 Future Improvements

While our prototype already computes reasonable wrong path probabilities, the performance of our implementation could be improved in several ways. First, the probabilistic analysis of each subdomain is trivially parallelizable, but we currently compute it sequentially as the GLPK solver is not threadsafe (but has a convenient interface for a prototype implementation). Secondly, a smarter subdivision similar to branch-and-bound methods could reduce the number of subdomains checked by SMT. Next, instead of DSIs, we could employ a more sophisticated method using the concentration of measure inequalities [31]. This method does not necessarily provide tighter probability bounds but is more efficient in general (but also more complex to implement and the implementation from [31] is not available). Finally, our implementation using rationals is costly and can be improved either by using high-precision floating-point arithmetic internally—at the expense of some guarantees—or by using interval arithmetic with floating-point bounds and outwards rounding—at the expense of the complexity of implementation.

benchmarks	wrong path probability		time (in seconds)	
	independent	dependent	independent	dependent
bspline2	5.66e-5	6.25e-5	53.30	54.27
rigidBody1	6.28e-3	1.37e-2	69.20	53.53
rigidBody2	5.05e-2	7.68e-2	195.46	77.91
traincar1	2.61e-3	5.22e-3	52.02	52.32
traincar2	3.51e-3	9.48e-3	68.07	56.88
traincar3	7.29e-4	7.20e-3	25.01	19.08
traincar4	7.19e-3	8.89e-2	20.70	9.32
filter3	3.23e-3	3.78e-3	54.54	54.20
filter4	6.72e-4	5.33e-4	47.14	47.36
solveCubic	3.13e-5	2.50e-4	58.00	56.43
classID0	5.70e-2	0.119	331.13	60.38
classID1	5.70e-2	8.66e-2	293.04	57.92
classID2	9.37e-2	0.158	383.73	62.57
neuron	0.138	0.138	16.34	6.82

Table 3.8: Independent vs. dependent inputs (with uniform input distributions)

3.4 Conclusion

In this chapter, we have considered several guaranteed analyses for bounding the effects of numerical uncertainties on discrete decisions in terms of the probability of making the wrong decision. Our results show that existing exact and static analysis approaches possess limitations in terms of both scalability and accuracy.

However, we have successfully demonstrated that these limitations can be effectively overcome with our novel approach, which combines a non-probabilistic reachability analysis with probabilistic static analysis. Our method efficiently computes wrong path probabilities, which are tight enough to be useful in practice for small but interesting embedded programs.

Sound Probabilistic Roundoff Error Analysis

4

So far, roundoff error analysis for finite-precision programs performs *worst-case* analysis, i.e., considers that the most significant error gets introduced at every arithmetic operation. However, such an analysis can be overly pessimistic as the largest error may not occur in practice for every operation. Furthermore, many applications are designed to be robust to some noise and tolerate errors — as long as they remain within some acceptable bounds and appear infrequently. For instance, in control systems, feedback can compensate for larger errors in subsequent iterations. To address this differentiated behavior, it is crucial to compute the *probability* of certain error bounds.

Such an analysis of errors is even more relevant in light of the advancements in approximate computing [174], which introduces approximate architectures and storage. These hardware components are more resource-efficient but often exhibit probabilistic error behaviors, where operations have a certain probability of returning a larger error. In such cases, it is crucial to consider these probabilistic error specifications to compute precise error bounds and their probabilities, which is beyond the capabilities of state-of-the-art roundoff analyzers.

Several techniques exist [31, 156, 82, 143, 35, 136, 153], including ours presented in chapter 3, that track or compute probability distributions; however, they do not consider and support probabilistic reasoning about *errors* due to finite-precision arithmetic or approximate computing, or they only compute coarse-grained error estimates instead of error probability distributions [137].

In this chapter, we introduce the first sound probabilistic error analysis for floating-point numerical programs. This analysis computes a (discretized) probability distribution of roundoff errors at the program output that soundly over-approximates all possible error distributions.

To compute the error distribution, our analysis extends the abstract domain of probabilistic affine arithmetic discussed in Section 3.2 for roundoff errors. This probabilistic error analysis is combined with a probabilistic version of interval subdivision, enabling us to compute a precise probability distribution of errors efficiently.

Our analysis focuses on tracking two types of errors. Firstly, we consider standard floating-point arithmetic, where inputs are distributed according to a user-given distribution. In this setting, we consider the worst-case roundoff error according to IEEE754 standard [99] for each operation. However, as roundoff errors depend on the magnitude of variables, which are specified by probability distributions, we effectively obtain a probability distribution of errors.

Secondly, we extend our analysis to handle probabilistic error specifications inspired by approximate hardware. In this case, we not only consider probabilistic input ranges but also incorporate errors with probabilistic specifications, i.e., with a certain probability, the error for each operation can be larger than usual [137, 152].

We observe that the computed (discretized) distributions of errors are hard to interpret and use by a programmer. Hence, inspired by the error tolerance property of programs, we propose a new error metric. This metric defines that with a (high) probability p, the error does not exceed a refined threshold C_p, which is smaller than the overall worst-case error.

Recall our `controller` example from the introduction (Figure 1.1) where a result is computed as `res = -x*y - 2*y*z - x - z`. In 32-bit floating-point precision, the worst-case error in `res` is 1.73e-4. Our analysis computes that with probability $p = 0.85$, the error is at most $C_p = 1.54\text{e-}4$.

We implemented the probabilistic error analysis in a prototype tool called *Probabilistic Analyzer (PrAn)*, which extends the existing numerical analyzer Daisy [55]. We evaluated PrAn on various benchmarks from the literature, and the results demonstrate its effectiveness in soundly computing refined error bounds, considering a predefined probability.

In our experiments, we assumed that the programs could tolerate large errors with a low probability of 0.15. We aimed to compute bounds on the errors that occur with a higher probability p of 0.85. Taking this probability into account, along with the standard floating-point error specification, PrAn computes sound error bounds that are, on average, 17% and 16.2% and up to up to 48.9% and 45.1% smaller than worst-case errors for gaussian and uniform input distributions, respectively.

In conjunction with p=0.85, we also considered a probabilistic error specification. Under this assumption, individual operations have a probability of 0.1 for committing errors of larger magnitude (2 $\times$ standard floating-point error). With this probabilistic scenario and an occurrence probability of 0.85, PrAn computes sound error bounds that are, on average, 30.6% smaller than worst-case errors, with improvements of up to 73.1%.

Contributions In summary, in this work, we present:

a) a general static analysis that computes probabilistic distributions of finite-precision errors,

b) an instantiation of this analysis for exact and approximate error specifications,

c) a procedure to extract useful, human-readable refined error specifications, and

d) an open source tool '*PrAn*' implemented as an extension of Daisy, which is available on Zenodo (see `https://doi.org/10.5281/zenodo.8042198`).

The work presented in this chapter is based on the publication titled '*Sound Probabilistic Numerical Error Analysis*' [167] co-authored with Milos Prokop and Eva Darulova. Milos Prokop contributed to the generation of visual graphs and explored alternative error representations under my supervision, while Eva Darulova provided valuable feedback on the writing and editing of the paper.

4.1 Overview

Before presenting our proposed probabilistic error analysis in detail, we provide an overview of the problem and our proposed solution with an example.

```
// x: [-2, 2]
x := gaussian(-2.0, 2.0)
res := 0.954929658551372 * x - 0.12900613773279798 * (x * x * x)
```

Figure 4.1: Running example

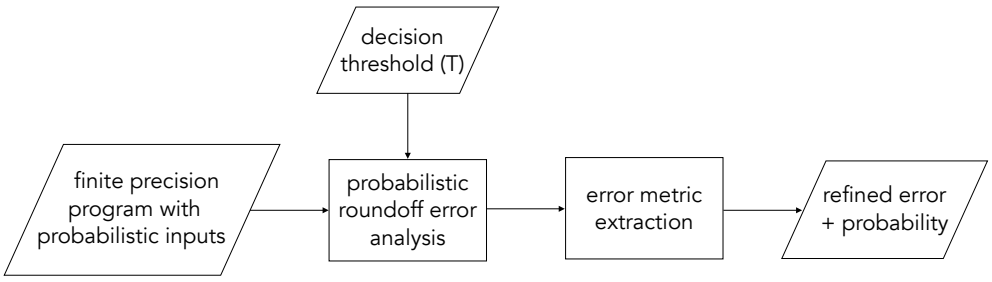

Figure 4.2: Overview of our approach

Problem Description The program presented in Figure 4.1 computes a third-order approximation of the sine function. The user specifies the distribution of input variables: here 'x' is distributed in [-2, 2] following a gaussian distribution. Suppose the user has further set the threshold probability to $p = 0.85$, which states that the program can tolerate large errors occurring with a probability of 0.15.

We can use state-of-the-art error analyzers to compute the worst-case error in `res`. For example, we used Daisy [55] and computed the worst-case error in single precision as 4.62e-7. However, this worst-case roundoff error may not always occur. Hence, we want to compute a sound refined bound C_p on the error that occurs with a probability of at least 0.85.

Our Proposed Technique Figure 4.2 shows the overview of our proposed technique. We propose to probabilistically track roundoff and other approximation errors and provide less pessimistic error bounds than a worst-case analysis. Our analysis takes into account the distributions of input variables, tracks errors probabilistically throughout the program for a user-provided uniform floating-point precision, and computes a *sound* over-approximation of the error distribution at the output. Finally, we show how to extract an *error metric* from the output distribution. The user of our approach provides a *threshold probability* p, and we can extract a tighter refined error bound C_p than worst-case error, that holds with probability p.

With our probabilistic error analysis tool PrAn, we can compute a refined error C_p of 2.67e-7 that occurs with at least the threshold probability of 0.85. This refined error is almost 43% smaller than the worst-case error of 4.67e-7 [1].

Our work focuses on straight-line arithmetic expressions as shown in Figure 4.1. However, it is possible to extend our analysis to reason about loops via loop unrolling and loop invariants as they are done in previous techniques presented in [52] and [53, 88, 140] respectively, and conditionals by a path-by-path analysis as in [140, 53]. All these techniques reduce these complex

[1]These refined bounds improve resource efficiency compared to worst-case errors for programs that can tolerate large but infrequent errors.

Algorithm 3 Tracking probabilistic roundoff errors

```
1:  procedure COMPUTE_PROB_ERROR(expr f, env E, dist d, divLimit L, precision prec, prob p)
2:      discrtDist = discretize(E, d, L)
3:      subDom = subdivide(E, d, L)
4:      errDist = φ
5:      for (dom_i, ρ_i) ← subDom do
6:          absErr_i = eval_prob_roundoff(f, dom_i, prec)        ▷ see Algorithm 2
7:          normErr_i = normalize_prob(absErr_i, ρ_i)
8:          errDist = merge(errDist, normErr_i)
9:      (err, prob) = extract_error_metric(errDist, p)
10:     return (err, prob)                  ▷ returns the refined error with probability
```

program structures into straight-line code, thus facilitating error analysis.

Moreover, our analysis primarily computes absolute roundoff errors. Some techniques compute relative errors directly [108] if the output range does not include zero, which, however, often happens in practice. Our method can also compute relative errors from the absolute ones only when the final range does not include zero.

4.2 Tracking Probabilistic Errors

Algorithm 3 shows the high-level overview of our analysis. Our algorithm takes as input the following parameters:

1. a program given as an arithmetic expression f,
2. ranges of the input variables E which define the environment of the program,
3. the probability distribution d of the input variables,
4. a limit L on the number of total subdivisions,
5. a uniform floating-point precision *prec*, and
6. a threshold probability p.

Our implementation of the analysis currently considers the same probability distribution for all input variables as well as uniform floating-point precision. However, it can be straightforwardly extended to consider different distributions and mixed precision.

Algorithm 3 first discretizes each input range distribution into subdomains (line 2). For each subdomain, it runs the probabilistic error analysis for the specified floating-point precision *prec* (line 6). The computed error distribution is then normalized by multiplying the probability of the subdomain occurring (line 7). Then, the new error is merged with the error distribution *errDist* (line 8) that accumulates error distributions for each subdomain to generate the complete error distribution of the output error. Finally, from *errDist*, we extract the error metric (line 9) and its actual probability, which may be larger than the requested p, and return it (line 9).

In Section 4.2.1, we first describe a simplified version of Algorithm 3, which performs probabilistic interval subdivision but computes errors for each subdomain with standard worst-case roundoff error analysis. While interval subdivision is standardly used to decrease overapproximations, to the best of our knowledge, it has not been used previously to compute probabilistic error bounds. Then, we extend this analysis with our novel probabilistic error analysis in Section 4.2.2, and combine it with probabilistic interval subdivision in Section 4.2.3. Finally, we show how our probabilistic error analysis can be further extended to account for approximate error specifications standard in today's approximate hardware in Section 4.2.4.

4.2.1 Probabilistic Interval Subdivision

First, we consider a relatively straightforward extension of the worst-case error analysis, which considers the distributions of input variables. For now, let us assume the function `eval_prob_roundoff` in Algorithm 3 (line 6) computes a worst-case error instead of a probability distribution, and focus on the interval subdivision where the intervals are subdivided probabilistically (line 3).

Our algorithm first subdivides each input interval equally and, for multivariate functions, takes their Cartesian product to generate subdomains. The probability of each subdomain is computed by taking the product of the probabilities of the corresponding subintervals. In this way, the algorithm generates a set of tuples (dom_i, ρ_i) where a value is in subdomain dom_i with probability ρ_i. This interval subdivision is similar to the subdivision implemented for our probabilistic analysis in Section 3.2.3.

Now, we can run the standard worst-case data-flow error analysis on each subdomain using state-of-the-art tools like Daisy to compute an error bound $absErr_i$. To make the worst-case analysis precise, we utilize affine arithmetic for the errors and interval arithmetic for the ranges, as it tends to have fewer over-approximations for non-linear expressions. Hence, the algorithm computes for each subdomain i an error tuple $(absErr_i, \rho_i)$, i.e. with probability ρ_i, the worst-case error is $absErr_i$. From these error tuples collected by $errDist$, our analysis extracts the error metric according to the use provided threshold probability p.

Extracting the Error Metric We want to extract an error C_p, smaller than the worst-case error, that holds with threshold probability p provided by the user. The function `extract_error_metric` in Algorithm 3 repeatedly removes tuples with the most significant error and subtracts their corresponding probability while the total probability of the remaining tuples remains at least p. Finally, we return the refined error and its probability, which is at least p but may be higher.

Running Example Recall our example in Figure 4.1, which has a worst-case error of 4.62e-7 in single precision. If we use $L = 100$ and $p = 0.85$, our prototype tool PrAn probabilistically subdivides the input domain into 100 subdomains and computes a reduced error bound of 2.97e-7 that holds with at least probability 0.85 (the worst-case error remains unchanged). If the program is immune to big errors with probability 0.15, the relevant error bound is thus reduced

Algorithm 4 Probabilistic range and roundoff error analysis

```
1:  procedure EVAL_PROB_ROUNDOFF(ASTnode node, subdom dom, precision prec)
2:      if node = lhs op rhs then                                  ▷ binary operation
3:          realRange = eval_range(lhs, rhs, op, dom)
4:          errLhs = eval_prob_roundoff(lhs, realRange, prec)
5:          errRhs = eval_prob_roundoff(rhs, realRange, prec)
6:          propErr = propagate(errLhs, errRhs, op)
7:          newRange = to_DSI(realRange + propErr)
8:      else                                                       ▷ node is a variable
9:          newRange = to_DSI(dom)
10:     for (x, w) ← newRange do
11:         roundoff = max_roundoff(x, prec)
12:         errDist.append(roundoff, w)
13:     return add_noise(propErr, normalize(errDist))
```

substantially by 35%. □

4.2.2 Probabilistic Roundoff Error Analysis

The error computed using only probabilistic interval subdivision is pessimistic as the actual error computation still computes worst-case errors. To compute a tighter distribution of errors at the output, we propose to track roundoff errors probabilistically throughout the program. To the best of our knowledge, this is the first sound probabilistic analysis of roundoff errors. For now, we consider an exact error specification, which means that the roundoff error introduced at each individual arithmetic operation is still the worst-case error, following the IEEE754 standard specification. However, we still obtain a distribution of errors, as the roundoff errors depend on the ranges of the values of the variables. As not all values of a variable are equally likely due to the distribution of the input variables, the errors are also not equally likely. In this section, we present the probabilistic error analysis only. In the following section, we combine this analysis with a probabilistic interval subdivision to compute tighter error distribution at the output.

The function `eval_prob_roundoff` shown in Algorithm 4 extends probabilistic AA presented in Section 2.3.2 to propagate error distributions through the program by performing a forward data-flow analysis recursively over the abstract syntax tree (AST) of an input program. For each AST node (*node*), an input subdomain (*dom*), and a precision (*prec*), the algorithm computes the distribution of accumulated roundoff errors. This error distribution is a sound over-approximation, i.e., the actual distribution lies within the computed upper and lower bounds. The soundness of our technique follows from the soundness of the underlying probabilistic AA.

For each binary arithmetic operation *op*, our analysis first computes the real-valued range (line 3) using probabilistic AA as it is done in Section 3.2. However, computing the accumulated errors is more involved and constitutes our extension over probabilistic AA and one of our main contributions of this work. To compute these accumulated errors, the algorithm first computes

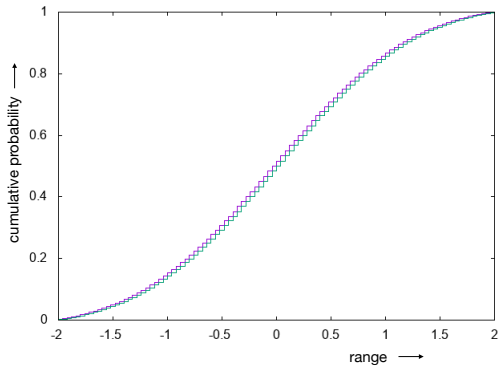

Figure 4.3: Input Distribution

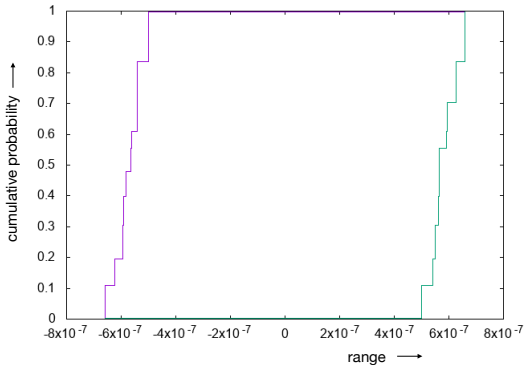

Figure 4.4: Error Distribution

the errors of its operands (lines 4-5) and, using these errors of the operands, computes the propagated error *propErr* in line 6. Our probabilistic error propagation extends rules from standard AA to probabilistic AA. However, for probabilistic AA, the operations themselves are more expensive. Depending on whether errors from one operand are correlated with the other one, the analysis solves linear programming problems. The propagated errors are then added to the real-valued range distribution to obtain the finite-precision range distribution (line 7), from which we can compute roundoff errors. Before doing so, we convert the probabilistic affine form to a DSI representation (`to_DSI`), which is more amenable for roundoff error computation.

Each focal element of the finite-precision DSI for the range assigns a probability w to an interval x. Hence, we can compute the roundoff error for each focal element following the floating-point abstraction presented in Section 2.1.1, and assign the weight w to that error (lines 10-12). That is, our procedure computes roundoff errors separately for each focal element, and appends them to generate a new error DSI (line 12). The newly committed error DSI is then normalized to [-1, 1] and added as a new noise term to the probabilistic affine form of the propagated errors (line 13). Thus, the error distribution is propagated throughout the program, and finally, we obtain a probabilistic affine form of the error at the output.

Extracting the Error Metric To extract the error metric from the probabilistic affine form of the error, we first transform it into a DSI structure, which we can then use to remove the large errors with low probabilities. We first sort the focal elements and then remove the elements with the largest error magnitude from the error DSI, similarly, to Section 4.2.1, until the total weight of the resultant DSI sums up to the threshold p. Our tool, PrAn, returns the maximum absolute value of the remaining focal elements as the refined error. This extraction effectively tries to find as small an error bound as possible, which will fall within the probability p.

Running Example We utilized probabilistic analysis on our sine example from Figure 4.1. We first discretize the input distribution into a DSI structure with 100 focal elements as shown in Figure 4.3. From this, our tool PrAn obtains the probability distribution of the error as a DSI structure shown in Figure 4.4. As the figure shows, the intervals of the DSI are extremely wide due to high over-approximation incurred and they almost fully overlap with each other.

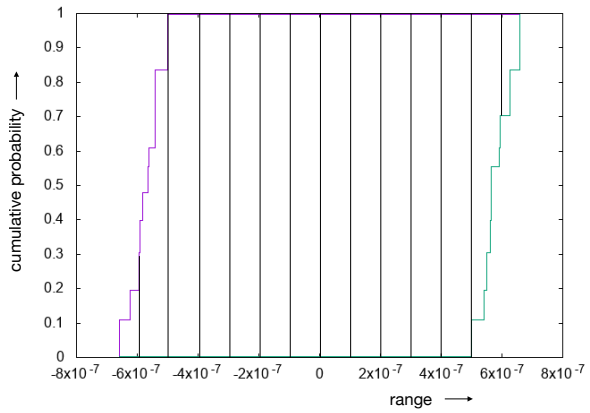

Figure 4.5: Alternative Error Metric with Vertical Subdivision

As a result, we could not refine the worst-case error using probabilistic analysis only. We also observe that the error computed using this analysis is even larger than our simple extension of the worst-case analysis with probabilistic interval subdivision. □

Alternative Error Metric Another choice of error representation could be to compute the probability of the most significant errors occurring. To extract this information, the error distribution is conceptually subdivided vertically first, as shown in Figure 4.5. In this way, we can compute the probability of the large error, which will be between the outermost extension and the following subdivision. We then need to sum all the weights intersecting with the specified interval. We observed this alternative error specification to be less useful in practice due to broad overlaps between the focal elements. We thus focus the remainder of this work on the first type of error metric.

4.2.3 Probabilistic Error Analysis with Interval Subdivision

To reduce the over-approximation incurred with probabilistic analysis only, we propose to combine the error analysis with our probabilistic interval subdivision from Section 4.2.1. We thus generate an error distribution for each subdomain and combine the distributions to obtain the final error distribution. The over-approximation is less because every subdomain is much smaller than the whole input environment. Thus, it improves the accuracy overall.

It may be noted that though we compute the error DSI for each subdomain as a step function, due to the merging *at the end*, the overall distribution cannot be a step function anymore. However, the actual error distribution is still inside the resultant distribution, thus guaranteeing soundness.

For the full analysis with probabilistic error analysis and subdivision, we choose 2 initial DSI discretizations and 50 outer interval subdivisions (to keep a fair comparison with 100 overall initial subdivisions). Note that during the computation, the number of DSI discretizations grows exponentially, i.e., does not remain at 2. However, we limit the total number of DSI discretization during the computation to 100, which we observed to be a good compromise between accuracy

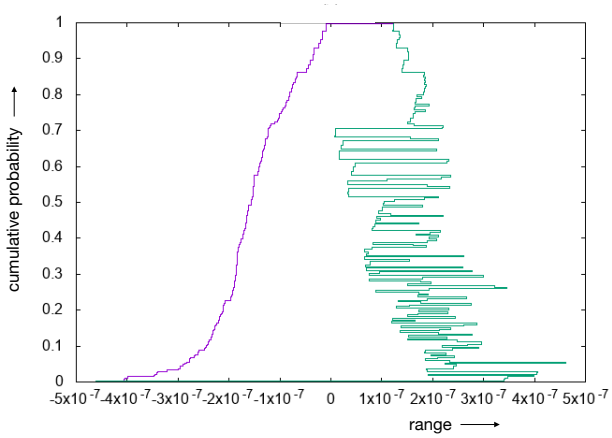

Figure 4.6: Error Distribution with full Probabilistic Analysis

and efficiency.

Running Example Figure 4.6 shows the resulting error distribution using the full probabilistic analysis. As mentioned before, this distribution is not a step function due to merging. However, the distribution is much tighter than the distribution obtained in Figure 4.4 with probabilistic error analysis only. With our full analysis, we are able to compute a refined error of 2.67e-7 (with a threshold probability of 0.85). This refined error is almost 43% smaller than the worst-case error of 4.67e-7 and 10% smaller than the error of 2.97e-7 computed using only probabilistic interval subdivision. □

4.2.4 Probabilistic Analysis with Approximate Error Specification

Until now, we have considered programs with exact error specification that considers only one error at every computation. However, some programs might exploit approximations to enable better performance in terms of energy and resource utilization. *Approximate computing* is an emerging design paradigm that does precisely that. Approximate hardware may, for instance, introduce large but infrequent errors in a computation [137, 152], where precision is not paramount in order to utilize resources more efficiently. Such approximate error specifications are themselves probabilistic: with a certain probability, the error produced by an operation is larger than the usual worst-case error bound. For such specifications, the worst-case analysis considers the most significant error for all computations, even if it occurs only infrequently, thus computing highly pessimistic error bounds.

With our proposed probabilistic error analysis, we can incorporate such an approximate error specification when the larger errors are bounded with a known upper bound. We extend the function `max_roundoff` in Algorithm 4 to compute a sound probabilistic error bound with approximate specifications. Previously, we computed one error for each focal element of the range DSI structure. Now, we compute multiple errors depending on the error specification,

resulting in appending multiple focal elements to the error distribution DSI, and propagate them through computation.

For example, for a specification where a larger error can occur with a certain probability, we compute two errors and thus obtain two focal elements at line 12 in Algorithm 4: one focal element that keeps track of the big error (with usually a small probability ρ), and one focal element which tracks the regular error (with probability $1 - \rho$).

Running Example We explain the method in detail with our running sine example from Figure 4.1. Let us consider an approximate error specification for the example where an error of size twice the usual machine epsilon ($2 \times \epsilon_m$) occurs with probability $\rho = 0.1$ and the regular roundoff error (ϵ_m) appears with probability 0.9. The state-of-the-art worst-case analysis computes 7.34e-7 as the worst-case error. As expected, in this probabilistic setting, the absolute worst-case error is much higher than the worst-case error of 4.62e-7 with exact error specification, where at each step, only an error of ϵ_m appears.

We parameterize our prototype tool PrAn with 2 DSI discretizations and 50 outer interval subdivisions for this experiment to make a fair comparison with other cases with exact error specifications. With probabilistic error specification and a threshold probability of 0.85, PrAn computes refined bounds of 5.02e-07 and 3.86e-7 using only probabilistic interval subdivisions and our complete probabilistic error analysis with probabilistic subdivisions, respectively. As these results show, only probabilistic subdivisions cannot take advantage of the approximate error specification, whereas the complete analysis does consider this property of the program and refines the error substantially.

As expected, all these bounds with approximate error specifications are higher than the cases with exact error specifications, where the refined bounds were 2.97e-7 and 2.67e-7 with only probabilistic analysis and the complete probabilistic analysis, respectively. □

4.2.5 PrAn: Probabilistic Roundoff Error Analyzer

We have implemented our proposed probabilistic error analysis technique(s) in the prototype tool PrAn, which is implemented on top of the state-of-the-art tool Daisy. Given an input distribution, PrAn fully automatically computes the error distribution, graphically plots the distribution, and finally extracts the error metric from it. The analyses in PrAn are implemented using sound intervals with arbitrary-precision outer bounds (with outwards rounding) using the GNU MPFR library [75], thus ensuring soundness and efficiency simultaneously. We use GLPK [7] as our simplex solver as it is done in our previous implementation of the probabilistic AA. The solver uses floating-point arithmetic internally and may thus introduce roundoff errors into the analysis. An entirely satisfactory solution would use a guaranteed LP-solver as Lurupa [115] or LPex [65]. However, given that the probabilities we compute are many orders of magnitude larger than double-precision floating-point roundoff errors and the performance of the guaranteed solvers is uncertain, we used GLPK and left this exploration for future work.

4.3 Experimental Evaluation

In this section, we rigorously evaluate our analysis implemented in PrAn to generate a refined bound, considering that the system can tolerate a large error with a certain probability. Before going into the experimental settings and evaluation, we first describe the set of benchmarks that we have used for this work. We have used all the benchmarks from our previous work presented in Section 3.3.1 and Section 3.3.4, which were collected from the domains of scientific computing (`sine, sineOrder3, sqroot, solveCubic`), embedded systems (`bspline, traincar, rigidBody, filter`) and machine learning (`classID` and `neuron`). We have also added a few new benchmarks, details of which are presented next.

Additional Benchmarks The benchmark `filter` is already introduced before in Section 3.3.1. In this work, in addition to `filter3` and `filter4`, we use another version (`filter2`) with 6 arithmetic operations by unrolling the control loop twice.

We have further extracted a polynomial support vector classifier trained on the Iris standard data set from the Python sklearn library using sklearn-porter [8]. For our benchmark `poly`, we have inlined the weights computed, and the three versions of this benchmark (`ID0, ID1, ID2`) correspond to the three decision variables, respectively classes, in the data set. Each of these benchmarks has 23 arithmetic operations.

4.3.1 Experimental Setup

For experiments, we have considered an error metric that returns a refined error given a probability threshold of 0.85. We use 32-bit single precision as the target precision. All experiments were performed on a Debian desktop computer with a 3.3 GHz i5 processor and 16GB RAM.

We have evaluated PrAn with two kinds of subdivisions, with two types of input distributions, and with two types of error specifications: we consider DSI discretization as well as input interval subdivision, uniform and gaussian distributions of inputs, including the DSI discretization and interval subdivision), and exact and probabilistic error specifications. We define eight different settings using the letters 'A-H' to evaluate PrAn.

- **A, B: probabilistic interval subdivision only** These settings use only probabilistic interval subdivisions from Section 4.2.1. Here, we limit the total number of outer interval subdivisions to 100. Both these settings also use an exact error distribution, i.e., only machine epsilon (ϵ_m) occurs for every arithmetic operation. The difference is that setting A considers gaussian, whereas B considers uniform distributions for program inputs.

- **C, D: entire probabilistic analysis** The complete probabilistic error analysis with probabilistic interval subdivision from Section 4.2.3 is used for both these settings. For a fair comparison, we limit the total number of input subdomains from DSI and input interval

subdivision to 100, i.e., when probabilistic AA is used, each input DSI is subdivided into 2, and the input interval subdivisions are determined such that the overall number of subdomains remains below 100. Note that *during* the computation, the number of DSI subdivisions is limited by 100 (not 2). Settings C and D consider gaussian and uniform distributions, respectively, for the input variables of the program and the exact error specification as it is in settings A and B.

- **E, F, G: entire probabilistic analysis with approximate error specifications** These three settings perform the entire probabilistic analysis with 100 total subdivisions (DSI and interval as in settings C and D) considering approximate error specifications (as in Section 4.2.4). Settings E and F assume a large error with magnitude $2 \times \epsilon_m$ occurs with probability 0.1, and the smaller error ϵ_m occurs with probability 0.9. Gaussian and uniform distributions were used in setting E and F, respectively. Setting G uses the same parameters as it is in setting E, except it uses a different approximate error specification. It considers a much larger error of magnitude $4 \times \epsilon_m$ with probability 0.1.

- **H: probabilistic interval subdivision only with approximate error specification** Setting H considers an approximate error specification as in setting G but performs only probabilistic interval subdivision from Section 4.2.1. We increase the input interval subdomains to 200, thus providing an additional edge to the probabilistic interval subdivision. With this setting, we evaluate whether probabilistic AA is helpful for approximate error specifications or simply using more interval subdivisions, which are relatively cheap, is sufficient.

It may be noted that we consider independent inputs for all our experiments. Additionally, we utilize uniform and gaussian input distributions for our experiments as PrAn automatically generates discretized DSI for these distributions. However, our approach can handle any discretized input distribution provided as a DSI, as well as inputs with (unknown) correlations. Our choice of input distribution is arbitrary, as the 'right' distribution is application-specific.

4.3.2 Results and Discussion

Table 4.1 and Table 4.2 show the absolute worst-case and refined error values with settings A, C, and E considering gaussian inputs and settings B, D, and F considering uniform inputs, respectively. The '# DSI subdiv' gives the number of focal elements of each input distribution, and the row '# outer subdiv' shows the maximum number of interval subdomains in both tables.

As the results show, the worst-case errors computed by settings A, B (probabilistic interval subdivision only) and settings C, D (entire probabilistic analysis) remain the same for many cases (7 and 11 benchmarks with gaussian and uniform distributions). For others, the worst-case error computed by settings A, B and C, D slightly differ because the subdivisions produce slightly different subdomains. The best results in terms of computing tighter error bounds are highlighted in the table. In setting E and F, the error specifications are approximated. Hence, we

expect to get larger worst-case errors which is indeed the case as shown in the tables.

Our results show that settings A, B, C, and D were able to refine the error bounds considering an exact error specification in almost all cases except those marked with '*'. The `traincar4` benchmark has 9 input variables, which means that with our total limit of subdivisions, each input is divided too few times to refine the error. For `traincar3`, PrAn could refine the worst-case error only with setting C and D, i.e., using our full probabilistic error analysis. Using approximate error specifications (settings E and F), PrAn was always able to refine the errors.

For some benchmarks, the full probabilistic error analysis outperforms probabilistic interval subdivision, but for others, it is the other way around. When probabilistic error analysis (settings C and D) is better, it tends to be more significantly better than vice-versa.

In general, which analysis is better is application specific: this depends on the over-approximations committed, which in turn depend on the operations and ranges of an application. A user should consider both variants and use the best result in a portfolio-like approach. Note, however, that for approximate error specifications, a probabilistic error analysis is necessary in order to adequately capture the probabilistic error specification.

We also show the error reduction between the refined and the corresponding worst-case errors in Table 4.3 considering gaussian distributions of inputs. Additionally, we keep setting D with uniform distribution here to compare the full performance of PrAn.

The reductions for the exact error specifications are, on average, 20.9%, 17%, and 16.2% with settings A, C, and D, respectively. The reduction with gaussian inputs in setting C is, in most cases, higher than for uniform inputs. We also see that the reductions are application-specific and, for many benchmarks, more substantial than the averages. In some cases, our probabilistic analysis, even with exact specifications, allows us to report refined errors with nearly half the magnitude as the worst-case (up to 49.8%), with the guaranteed probability of at least 0.85.

Our probabilistic analysis achieves even higher reductions with the approximate error specification (settings E and G) with on average 24.9% and 30.6% smaller refined errors than worst-case, respectively. Even if we double the number of total subdivisions using setting H, the probabilistic interval subdivision can only reduce the error on average by 25.4%, compared to 30.6% using our full probabilistic analysis. While for a few cases, setting H outperforms setting G (because of overapproximations in the probabilistic analysis), overall, our complete probabilistic analysis successfully captures the fact that large errors only occur infrequently.

Finally, we also compare the running times of the analyses in Table 4.4 with settings A, C, and E (averaged over 3 runs). As expected, the full probabilistic error analysis (settings C and E) takes more time than the analysis with probabilistic subdivisions only (setting A) for all benchmarks. Furthermore, setting E with approximate error specification is costlier than the analysis with the exact specifications. This result is also expected, as we have more focal elements for errors with setting E than with setting C. However, we believe that these analysis times are nonetheless acceptable for a static analysis which is run only once.

	worst-case error			refined error		
	A	C	E	A	C	E
#DSI subdiv	0	2	2	0	2	2
#outer subdiv	100	50	50	100	50	50
`sine`	**2.40e-7**	2.41e-7	4.76e-7	**1.56e-7**	1.68e-7	2.66e-7
`sineOrder3`	4.62e-7	4.62e-7	7.34e-7	2.97e-7	**2.67e-7**	3.86e-7
`sqroot`	**1.50e-4**	1.54e-4	2.42e-4	**8.38e-5**	8.89e-5	1.31e-4
`bspline0`	8.69e-8	8.69e-8	1.44e-7	**4.36e-8**	4.44e-8	5.51e-8
`bspline1`	**2.09e-7**	2.10e-7	3.86e-7	**1.96e-7**	1.97e-7	2.67e-7
`bspline2`	2.16e-7	**2.12e-7**	4.21e-7	**1.87e-7**	1.89e-7	2.91e-7
`bspline3`	5.71e-8	5.71e-8	8.44e-8	**3.33e-8**	3.39e-8	3.72e-8
`rigidBody1`	**1.58e-4**	1.73e-4	3.01e-4	**9.99e-5**	1.57e-4	1.98e-4
`rigidBody2`	1.94e-2	**9.70e-3**	1.38e-2	1.06e-2	**8.50e-3**	1.05e-2
`traincar1`	**1.99e-3**	2.00e-3	2.94e-3	1.84e-3	**1.67e-3**	2.52e-3
`traincar2`	1.37e-3	1.37e-3	2.06e-3	1.32e-3	**1.19e-3**	1.83e-3
`traincar3`	2.29e-2	2.29e-2	3.85e-2	2.29e-2*	**2.26e-2**	3.46e-2
`traincar4`	2.30e-1	2.30e-1	4.13e-1	2.30e-1*	2.30e-1*	3.75e-1
`filter2`	1.04e-6	1.04e-6	1.72e-6	8.64e-7	**7.57e-7**	1.13e-6
`filter3`	2.99e-6	**2.87e-6**	4.99e-6	2.62e-6	**2.58e-6**	4.52e-6
`filter4`	6.51e-6	**5.20e-6**	9.16e-6	6.09e-6	**4.96e-6**	8.69e-6
`solveCubic`	**1.83e-5**	2.02e-5	3.35e-5	**1.73e-5**	1.90e-5	2.80e-5
`classID0`	**8.77e-6**	9.10e-6	1.45e-5	7.95e-6	**7.92e-6**	1.20e-5
`classID1`	**4.63e-6**	4.76e-6	7.70e-6	**4.28e-6**	4.38e-6	6.70e-6
`classID2`	**7.32e-6**	7.60e-6	1.25e-5	**6.35e-6**	6.55e-6	1.02e-5
`polyID0`	**5.56e-3**	5.80e-3	9.29e-3	**2.96e-3**	5.32e-3	7.94e-3
`polyID1`	**6.81e-4**	7.56e-4	1.23e-3	**4.51e-4**	7.08e-4	1.12e-3
`polyID2`	**5.05e-3**	5.40e-3	8.73-3	**2.84e-3**	5.08e-3	7.55e-3
`neuron`	**3.22e-5**	7.02e-5	9.87e-5	**3.20e-5**	5.25e-05	7.47e-5

Table 4.1: The worst case and refined error in different settings with guassian inputs (* indicates the worst-case error could not be refined with the setting)

	worst-case error			refined error		
	B	D	F	B	D	F
#DSI subdiv	0	2	2	0	2	2
#outer subdiv	100	50	50	100	50	50
`sine`	2.72e-7	**2.41e-7**	4.76e-07	2.18e-7	**1.83e-7**	2.66e-7
`sineOrder3`	4.62e-7	4.62e-7	7.34e-7	3.29e-7	**2.84e-7**	4.04e-7
`sqroot`	**1.50e-4**	1.54e-4	2.42e-4	**9.02e-5**	9.33e-5	1.39e-4
`bspline0`	8.69e-8	8.69e-8	1.44e-7	**4.60e-8**	4.77e-8	5.80e-8
`bspline1`	2.12e-7	**2.10e-7**	3.86e-7	2.00e-7	2.00e-7	2.81e-7
`bspline2`	2.18e-7	**2.12e-7**	4.21e-7	2.08e-7	**1.93e-7**	2.93e-7
`bspline3`	5.71e-8	5.71e-8	8.44e-8	3.50e-8	3.50e-8	3.99e-8
`rigidBody1`	**1.58e-4**	1.73e-4	3.01e-4	**1.50e-4**	1.57e-4	1.98e-4
`rigidBody2`	1.94e-2	**9.70e-3**	1.38e-2	1.71e-2	**8.55e-3**	1.13e-2
`traincar1`	2.00e-3	2.00e-3	2.94e-3	1.91e-3	**1.67e-3**	2.48e-3
`traincar2`	1.37e-3	1.37e-3	2.06e-3	1.32e-3	**1.19e-3**	1.80e-03
`traincar3`	2.29e-2	2.29e-2	3.85e-2	2.29e-2*	**2.26e-2**	3.46e-2
`traincar4`	2.30e-1	2.30e-1	4.13e-1	2.30e-1*	2.30e-1*	3.75e-1
`filter2`	1.04e-6	1.04e-6	1.72e-6	9.08e-7	**7.74e-7**	1.17e-6
`filter3`	3.07e-6	**2.87e-6**	4.99e-6	2.96e-6	**2.58e-6**	4.26e-6
`filter4`	8.23e-6	**5.20e-6**	9.16e-6	8.17e-6	**4.95e-6**	8.69e-6
`solveCubic`	**1.85e-5**	2.02e-5	3.35e-5	**1.74e-5**	1.90e-5	2.80e-5
`classID0`	9.10e-6	9.10e-6	1.45e-5	8.52e-6	**7.79e-6**	1.20e-05
`classID1`	4.76e-6	4.76e-6	7.70e-6	4.66e-6	**4.35e-6**	6.79e-6
`classID2`	7.60e-6	7.60e-6	1.25e-5	7.44e-6	**6.36e-6**	1.01e-05
`polyID0`	5.80e-3	**5.19e-3**	9.29e-3	**4.17e-3**	4.78e-3	7.94e-3
`polyID1`	7.55e-4	**5.71e-4**	1.23e-3	7.00e-4	**5.04e-4**	5.04e-4
`polyID2`	5.40e-3	**5.19e-3**	8.38e-3	**3.94e-3**	4.78e-3	7.04e-3
`neuron`	**6.40e-5**	7.02e-5	9.87e-5	**3.94e-5**	4.86e-5	6.86e-5

Table 4.2: The worst case and refined error in different settings with uniform inputs (* indicates the worst-case error could not be refined with the setting)

benchmarks	reduction (%)					
	A	C	D	E	G	H
sine	34.9	30.3	24.1	44.1	**52.2**	39.6
sineOrder3	35.7	42.1	38.5	47.4	**52.9**	34.0
sqroot	44.3	42.4	39.5	45.6	**56.6**	45.8
bspline0	49.8	48.9	45.1	61.7	**73.1**	56.6
bspline1	6.2	6.0	4.7	30.8	**40.2**	9.7
bspline2	13.5	11.0	9.4	31.0	**40.6**	17.4
bspline3	41.6	40.7	38.7	56.0	**67.0**	48.6
rigidBody1	36.9	8.8	8.8	34.2	34.8	**38.6**
rigidBody2	45.4	12.4	11.8	24.1	13.5	**49.2**
traincar1	7.5	16.4	16.4	14.4	**20.2**	7.2
traincar2	3.3	12.6	12.7	11.3	**13.6**	1.9
traincar3	0.0	1.3	1.3	10.1	**11.2**	2.2
traincar4	0.0	0.0	0.0	9.2	**8.3**	0.0
filter2	16.9	27.5	25.9	34.5	13.9	**29.7**
filter3	12.4	10.0	10.0	9.6	23.0	**23.8**
filter4	6.5	4.6	4.9	5.1	**47.5**	10.4
solveCubic	5.8	5.8	5.9	16.2	**41.9**	9.3
classID0	9.3	13.0	1.5	17.6	**18.7**	13.6
classID1	7.5	8.0	8.6	12.9	5.3	**10.1**
classID2	13.3	13.8	16.3	18.3	**22.2**	14.8
polyID0	46.8	8.2	8.0	14.5	14.6	**50.1**
polyID1	33.8	6.3	11.7	8.7	10.6	**37.3**
polyID2	43.9	5.9	8.0	13.6	19.6	**49.2**
neuron	0.9	25.3	30.8	24.3	**41.7**	13.9

Table 4.3: Reductions in % in errors w.r.t. the standard worst-case in different settings and analysis time (averaged over 3 runs) in different settings

	A	C	E		A	C	E
sine	9.0	361.0	6551.0	traincar4	0.1	22.0	36.0
sineOrder3	5.0	187.0	406.0	filter2	0.9	107.0	182.0
sqroot	1.0	89.0	2036.0	filter3	0.9	259.0	676.0
bspline0	0.4	22.0	722.0	filter4	16.0	654.0	1719.0
bspline1	0.6	42.0	817.0	solveCubic	16.0	191.0	864.0
bspline2	0.8	50.0	966.0	classID0	23.0	219.0	464.0
bspline3	0.4	18.0	80.0	classID1	2.0	236.0	473.0
rigidBody1	0.1	35.0	76.0	classID2	0.1	219.0	421.0
rigidBody2	0.4	74.0	659.0	polyID0	0.5	1972.0	10905.0
traincar1	0.3	75.0	235.0	polyID1	0.8	2006.0	11599.0
traincar2	0.3	389.0	682.0	polyID2	3.0	2309.0	10658.0
traincar3	0.1	12.0	25.0	neuron	0.5	71.0	257.0

Table 4.4: Analysis time in seconds (averaged over 3 runs) in different settings

4.4 Conclusion

This chapter introduces a novel probabilistic analysis method for tracking errors arising from finite-precision arithmetic in straight-line code. Unlike conventional worst-case error analysis, our approach focuses on computing the probability distribution of roundoff errors at the output. Additionally, to better interpret the error distribution, we have proposed an error metric that identifies a potentially smaller error than the worst-case, occurring with a predefined threshold probability.

This probabilistic analysis is particularly beneficial for applications that can tolerate larger errors as long as their probability is bounded. By considering the probabilistic nature of errors, our refined approach becomes even more valuable, especially in the context of modern approximate hardware, where probabilistic error specifications are common.

5 (Conditional) Verification of Realistic Floating-Point Programs

In addition to roundoff errors, floating-point arithmetic introduces special values such as not-a-number (NaN) and infinities [100]. NaNs emerge as results of invalid operations like division by zero or taking the square root of negative values, while infinity often arises due to overflows in the floating-point format.

Existing techniques for verifying the absence of infinities, NaNs, and large roundoff errors (cancellation) in floating-point applications do not scale well for whole floating-point applications. While static analysis tools can, in theory, verify the absence of floating-point issues, they are either limited to small programs (essentially individual functions) [60, 161, 53, 140, 132, 55] or suffer from significant over-approximations and do not track roundoff errors [26, 135, 89]. Dynamic analyses that attempt to show the presence of such issues [77, 22, 38, 182] either lack scalability or only work with manual intervention. More guided techniques such as symbolic execution [129] rely on a back-end SMT solver, for which floating-point theories have very limited scalability.

Therefore, there is a clear need for more scalable techniques capable of handling larger floating-point programs that can provide an automated, accurate, and preferably sound way of verifying the absence of cancellation errors, infinities, and NaNs.

We observe that often only a relatively small part of a program performs complex numerical computations—we call these parts the *numerical kernels*. Therefore, existing roundoff analyzers can be applied to these kernels if preconditions that bound the kernel input ranges are provided.

However, manually obtaining such preconditions is challenging since the kernels are usually nested in loops as functions. Recall our `nbody` example from Figure 1.4; the `numerical_kernel` function identified on line 9 is one of the most computationally intensive functions, but it is concealed within loops and 2 function calls.

Based on the above observation, we introduce a two-phase analysis that combines different program analyses to conditionally verify the absence of infinity, NaN, and cancellation errors in numerical kernels 'concealed' in large programs. First, we employ a scalable program analysis to infer the ranges of kernel inputs in the context of the containing application. The second phase utilizes state-of-the-art roundoff error analysis that assumes the inferred input ranges to verify the numerical kernels.

The key insight behind this combination is that the first scalable analysis does not need to perform sophisticated floating-point reasoning. Instead, it needs to focus on capturing the ranges of kernel inputs as precisely as possible, which serve as inputs for the subsequent phase

of numerical analysis.

Automatically inferring precise ranges of kernel inputs is the main challenge in our two-phase analysis. Our analysis first attempts to generate ranges fully soundly utilizing tools based on abstract interpretation [135]. If they cannot infer useful (finite) ranges for the kernels, we propose to employ scalable dynamic analysis techniques to generate large kernel input ranges and capture as many feasible inputs as possible. This includes an existing directed greybox fuzzer [27] and a novel guided blackbox fuzzing technique that provides heuristic guidance by directing the fuzzing towards program inputs that are more likely to produce large ranges for kernel inputs.

After inferring the kernel ranges, the second phase utilizes a slightly adapted version of the existing static and sound roundoff error analysis tool Daisy [55] to verify the kernels. In case this analysis produces warnings for special values, we additionally utilize SMT-based bounded model-checking [119] to check for spurious warnings.

The use of dynamic analysis in the first phase implies that we can only obtain approximate ranges for the kernel inputs. As a result, the verification of the kernels can only be done *conditionally* because the verification is performed under the assumption that the input-domain specifications precisely describe possible values of the kernel inputs. Despite the conditional nature of our analysis, it enables us to provide some guarantees for real-world floating-point programs that are beyond the capabilities of current roundoff error analyzers.

We have implemented our adaptation of blackbox fuzzing and our proposed guided blackbox fuzzing in a prototype tool, '*Blossom*'. Our evaluation shows that Blossom is capable of handling programs written in C and C++ (we can also handle Rust programs) with complex programming structures and over 2K lines of code with 24 kernels and can successfully generate ranges of kernel inputs.

Through our two-phase analysis framework, we can conditionally prove that no infinity or NaN occurs for 16 kernels; for 3 of those, the verification is sound. For 14 kernels, we also show the absence of cancellation errors, which are the leading cause of large roundoff errors.

Contributions To summarize, this work makes the following contributions:

a) a two-phase framework that combines dynamic and static analyses to conditionally verify the absence of floating-point special values and large roundoff errors in kernels,

b) a novel guided blackbox fuzzing and an adaption of blackbox fuzzing to infer kernel ranges, implemented in an open-source prototype tool called '*Blossom*' that is publicly available on Zenodo (see `https://doi.org/10.5281/zenodo.8043359`), and

c) an evaluation on a new benchmark set of mid-sized numerical programs.

This chapter is based on our paper titled '*A Two-Phase Approach for Conditional Floating-Point Verification*' [168]. The work is done in collaboration with Clothilde Jeangoudoux, Joshua Sobel, Eva Darulova, and Maria Christakis. Clothilde Jeangoudoux has explored a part of the state-of-the-art analysis fuzzing techniques. Joshua Sobel has contributed a part of the prototype implementation under my supervision. Eva Darulova and Maria Christakis have provided extensive feedback on the writing of the paper.

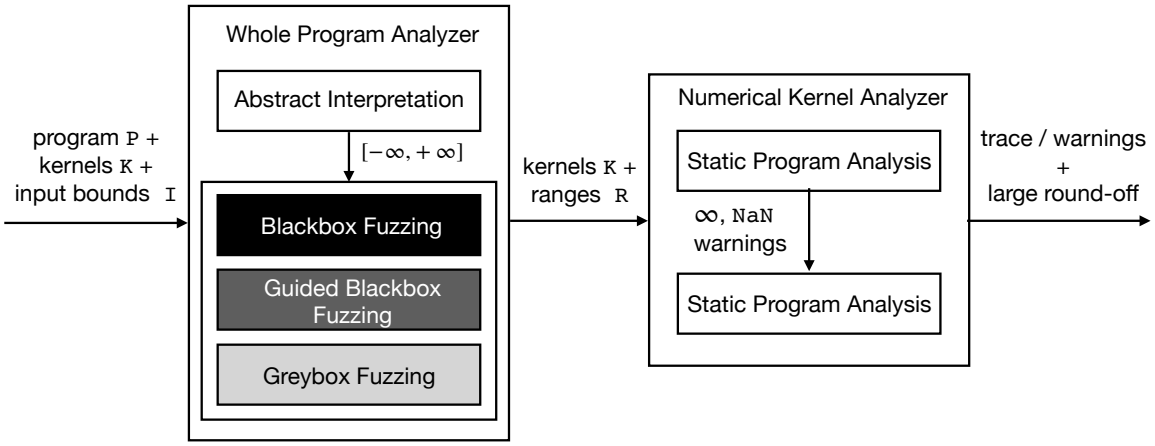

Figure 5.1: Overview of our approach

5.1 Overview

We first provide a brief overview of our two-phase approach that strives to extend the reach of existing floating-point analyses, enabling them to ensure the absence of cancellation errors and special float values in individual floating-point numerical kernels. We observe that such numerical computation-heavy kernels are prevalent in real-world applications across various domains. However, they are often obscured by numerous function calls and other non-numerical codes that are not handled by state-of-the-art roundoff analyzers.

Our approach, depicted in Figure 5.1, consists of two phases. In the first phase, we consider a set of numerical kernels, denoted by $\mathcal{K}$, in a program $\mathcal{P}$ specified by *the user*. Then, we infer bounds on the input variables of these kernels. In the second phase, we utilize these ranges to (conditionally) verify the kernels, meaning we aim to prove the absence of special values and large roundoff errors, subject to the inferred bounds.

5.1.1 First Phase: Whole Program Analysis

In the first phase, we employ a program analyzer that automatically infers bounds $\mathcal{R}$ on the *kernel inputs* by starting from the *program inputs* constrained by $\mathcal{I}$. These bounds are essential because the presence of cancellations and special values directly depends on the range of possible values, and an unbounded input range will typically lead to unbounded roundoff errors and special values.

To obtain the kernel ranges, it is necessary to analyze the entire program or at least up to the numerical kernels. However, computing the exact ranges is usually infeasible, so we aim to approximate them as tightly as possible. We propose using a sound static analysis as a first step, which computes an over-approximation of the actual ranges. These ranges cover all possible inputs, including spurious ones. Therefore, we aim to make these ranges as tight (small) as

possible. If the abstractions necessary for the static analyzer become too large, we propose using dynamic analysis to compute an unsound approximation of the kernel ranges. In this case, the ranges should be as wide as possible to capture as many concrete executions as possible.

Sound Static Analysis Our approach for obtaining a sound over-approximation of the kernel ranges first attempts using abstract interpretation techniques, specifically the industry-strength analyzer Astrée [135]. Astrée is capable of scaling to large programs with complex code and data structures and offers various abstract domains. When selecting an abstract domain in Astrée, one must strike a balance between the amount of over-approximation and the time it takes for the analysis to run. One of the available domains in Astrée is the interval domain, which abstracts a set of concrete variable values by their lower and upper bounds: $[\underline{x}, \overline{x}] := x; |; \underline{x} \leq x \leq \overline{x}$. While interval arithmetic [139] operations are efficient, intervals cannot capture correlations between variables, leading to an over-approximation of the actual behavior (e.g., $x - x \neq 0$ in interval arithmetic). More sophisticated domains, such as octagons, are more precise but less efficient. In our benchmarks, we did not observe significant differences in the results obtained with interval and octagon domains, probably because our benchmarks include many nonlinear operations. Therefore, we chose the interval domain as the abstract numerical domain for our purposes.

Dynamic Analysis If Astrée is unable to compute reasonable ranges for kernel inputs, we turn to dynamic analysis techniques. One such dynamic analysis technique is fuzzing, typically used to find individual inputs that demonstrate undesirable behavior. We propose a different use of the fuzzer: to fuzz a program while monitoring the kernel inputs, recording the lower and upper bounds seen during concrete executions. We instrument each user-specified kernel in the program with a kernel monitor that tracks the smallest and largest values seen for each kernel input. We then repeatedly fuzz the instrumented program and report the minimum and maximum values observed for each kernel input at the end.

The effectiveness of this approach is dependent on the choice of program inputs used for fuzzing. We propose and experimentally compare the blackbox, guided blackbox, and directed greybox fuzzing methods for input selection (as detailed in the experimental evaluation in Section 5.4.3).

Blackbox fuzzing is a simple yet effective technique that randomly selects inputs from the program ranges $\mathcal{I}$ without considering the internal structures of programs. We utilized a blackbox fuzzer that randomly draws inputs from the program ranges $\mathcal{I}$, runs the programs, and keeps track of the kernel ranges.

We introduce a variant of blackbox fuzzing, termed *guided blackbox* fuzzing, which aims to increase the size of the kernel input ranges. We accomplish this by keeping track of inputs that have expanded the kernel ranges and using them as a basis for generating new inputs that are within a small distance of the recorded inputs. This approach is based on the heuristic assumption that these new inputs will continue to widen the monitored ranges.

While blackbox techniques are straightforward to implement, they do not take into account the program structure. We thus evaluate an adaptation of *directed greybox* fuzzing, implemented in state-of-the-art tool AFLGo [27] that can be directed toward specific program locations while

exploring as many different paths in the program as possible. We first fuzz the program to obtain an initial estimate for the kernel input ranges with AFLGo (targeting the kernel). Then, we employ AFLGo in a refinement loop that iteratively attempts to widen the currently seen kernel input ranges. We instrument the kernels with conditional statements that check whether a kernel input is outside of the current kernel range. We use this conditional statement as a target for AFLGo, effectively directing it to find kernel inputs that are outside of the current estimate. If AFLGo finds a program input that widens the current kernel input range, we update it accordingly and iterate the process until a user-defined timeout.

It is important to note that the dynamically computed input ranges may contain more concrete values than those witnessed by the fuzzer. This is because there may be values between the lower and upper bounds that are neither observed by the fuzzer nor feasible. If we were to consider only the values observed at runtime, we could analyze kernels for individual traces, but this would be prohibitively expensive [22]. However, demonstrating that no special values or large roundoff errors occur within a kernel for a given input range would prove this for more executions than could be explored through dynamic testing (since there are usually too many floating-point values to explore exhaustively).

5.1.2 Second Phase: Numerical Kernel Analysis

In the second phase, we employ a static analyzer to analyze the numerical kernels ($\mathcal{K}$) identified by the user based on the inferred ranges ($\mathcal{R}$) from the first phase, as illustrated in Figure 5.1. Our aim is to detect the potential presence of special floating-point values and large roundoff errors or to prove their absence.

To achieve this, we make use of the state-of-the-art numerical analyzer Daisy [55] that we adapt for our purpose. By default, Daisy computes an absolute error bound on the kernel output, which alone is not very informative since it does not consider how this error propagates to the end of the application (although there exist scalable analyses that potentially can compute this information, e.g., [133]). However, for many numerical applications, the exact error bound is not critical as the algorithm itself is approximate. Therefore, we aim to demonstrate that the *roundoff errors are not too large*.

To this end, we modify Daisy to issue a warning whenever it detects a possible *cancellation* that accentuates existing errors (e.g., when two values with close magnitudes are subtracted [85]). These cancellation errors are not immediately apparent from the computed result, so an automated analysis is generally necessary. We present our adaptation of Daisy in Section 5.3. Warnings about special values can be optionally verified by a SAT/SMT-based model checker for spurious cases. Additionally, Daisy includes an optimization procedure that can enhance the accuracy of the kernels by rewriting the arithmetic expressions to produce smaller roundoff errors.

5.1.3 Soundness Guarantees

We utilize the extended Daisy analysis to conditionally verify kernels, ensuring that they do not generate NaN or infinity values and do not result in cancellation errors. When the kernel input ranges are computed soundly using abstract interpretation, our verification is conditional in that we only verify the absence of cancellations for the kernels but not for the rest of the program.

When the ranges are computed using dynamic analysis, we, in addition, cannot guarantee that they cover all possible kernel inputs or even possible inputs in the containing application. Nonetheless, we verify that cancellation does not occur for all concrete inputs within the ranges. While NaN or infinity values are easy to detect, cancellation errors generally cannot be detected by inspecting the computed results. Therefore, our combination is valuable.

5.2 First Phase: Whole Program Analysis

We have considered different program analysis techniques for computing the kernel ranges. We explain each of these techniques in detail in the following sections.

5.2.1 Abstract Interpretation with Astrée

We begin by configuring our framework for sound verification using Astrée. Since Astrée does not assume ranges for global variables directly, we add wrapper functions to provide bounds for these variables. To capture ranges of kernel inputs, we annotate the kernels $\mathcal{K}$ with Astrée's `__ASTREE_log_vars()` construct, which logs about the kernel inputs at the entry of the kernels. Although our instrumentation step is currently manual, it can be automated easily. We then execute Astrée on the annotated program. If Astrée reports a trivial range of $[-\infty, +\infty]$ or a very large range for a kernel, we stop the analysis because roundoff error analyzers do not produce meaningful results for very large input ranges. However, if the computed range set is meaningful, we proceed with kernel analysis by using the annotated kernel inputs.

It should be noted that Astrée's analysis can be parameterized to a great extent using program-specific knowledge, leading to even more precise results. However, this approach requires a significant amount of manual effort and a deep understanding of the program's intricacies. To avoid this, we parameterize Astrée in a generic manner to minimize the need for program-specific knowledge. Specifically, we use semantic loop unrolling up to a defined loop bound to reduce over-approximation in the analysis for all benchmarks.

5.2.2 Fuzzing with Blossom

In cases where Astrée generates highly over-approximated ranges, we turn to fuzzing techniques to compute kernel ranges. To this end, we have developed a prototype tool named '*Blossom*' that

Algorithm 5 Code instrumentation for range computation

1: **procedure** EXECUTE_AND_MONITOR_KERNELS(range $[\mathcal{R}_{min}, \mathcal{R}_{max}]$, input v_i)
2: **if** $v_i < R_{min}$ **then**
3: $R_{min} = v_i$ ▷ stores min val encountered
4: **else if** $v_i > R_{max}$ **then**
5: $R_{max} = v_i$ ▷ stores max val encountered
6: return $[\mathcal{R}_{min}, \mathcal{R}_{max}]$

```
1  double R_min, R_max;
2  // more functions ...
3  numerical_kernel(double v_i) {
4   compute_range ([R_min,R_max], v_i)
5   ... }
6  // more functions...
7  double original_program_main(double i){...} //original main function
8  int main() {
9   double i;
10  do {
11    i = generate_random(i_l, i_h); // randomly generates input within bound
12    original_program_main(i)
13  } while (time_spent <= T); // fuzz the inputs the program until time T
14  return [R_min, R_max]; //returns min and max val of k
15 }
```

Figure 5.2: Instrumentation for blackbox fuzzing

incorporates two fuzzing techniques. Blossom is designed as an LLVM pass and can handle input programs written in C, C++, and Rust, featuring complex programming constructs and data types. Additionally, Blossom can be used with any programming language that compiles to LLVM. The tool requires four inputs: the program $\mathcal{P}$, a configuration file that specifies the ranges of program inputs, the type of fuzzing technique, and a timeout. The LLVM pass automatically inserts code into $\mathcal{P}$ that carries out the specified fuzzing technique and records the ranges of kernel inputs until the timeout is reached. The following section provides an in-depth explanation of the fuzzing techniques employed by Blossom.

Blackbox Fuzzing We will start by discussing the blackbox fuzzing method implemented in Blossom to compute kernel ranges. To instrument the code, Blossom adds an input generator (as shown in Figure 5.2) that uses the `generate_random()` function (it implements `srand()` internally) to randomly generate values of program inputs from the set of input bounds $\mathcal{I}$ (line 12 in Figure 5.2). Blossom also instruments the `execute\_and\_monitor\_kernels()` function in line 4 that is explained in Algorithm 5. It adds conditional statements to check if the current valuation of v_i is smaller than the minimum value R_{min} or larger than the maximum value R_{max} encountered so far. If so, it updates R_{min} or R_{max} accordingly.

Once the code is instrumented, Blossom executes the program and monitors the values of kernel inputs I_k until a specified timeout T is reached. Finally, it returns a set of ranges for kernel inputs. It is worth noting that the program shown in Figure 5.2 has one kernel function,

Algorithm 6 Guided Blackbox Fuzzing

1: **procedure** GUIDED_BLACKBOX(prog $\mathcal{P}$, ranges $\mathcal{I}$, kernels $\mathcal{K}$, timeout T, mutation m, const c)
2: $Q \leftarrow \phi, \{\mathcal{R}_{lo}\}$ = `DBL_MAX`, $\{\mathcal{R}_{hi}\}$ = `DBL_MIN`
3: $\{r_1, \cdots, r_n\} \leftarrow$ `compute_radii`($\mathcal{I}$, c) ▷ generates mutation radii
4: **while** $T \neq 0$ **do**
5: **if** $Q == \phi$ **then** ▷ queue is empty
6: **for** i = 1 to m **do**
7: $Q \leftarrow$ `enqueue(generate_random_input`($\mathcal{I}$)) ▷ generates random inputs
8: **else** ▷ queue has elements
9: $\{v_1, \cdots, v_n\} \leftarrow$ `dequeue`(Q)
10: $[\{\mathcal{R}_{lo}\}, \{\mathcal{R}_{hi}\}] \leftarrow$ `execute_and_monitor_kernels`($\mathcal{K}$)
11: **if** `kernel_range_updated`($[\{\mathcal{R}_{lo}\}, \{\mathcal{R}_{hi}\}]$) **then**
12: **for** i = 1 to m-1 **do**
13: $\{d_1, \cdots, d_n\} \leftarrow$ `mutate`($v_1 \mp r_1, \cdots, v_n \mp r_n$)
14: $Q \leftarrow$ `enqueue`($\{d_1, \cdots, d_n\}$)
15: $Q \leftarrow$ `enqueue(generate_random_input`($\mathcal{I}$) ▷ avoids local max/min
16: return $[\{\mathcal{R}_{lo}\}, \{\mathcal{R}_{hi}\}]$ ▷ returns kernel input ranges

`numerical_kernel()`, with one kernel input v_i, which is typically reached after several function calls in the program (not shown in Figure 5.2).

Guided Blackbox Fuzzing In addition to usual blackbox fuzzing, Blossom implements a guided blackbox fuzzing technique to generate program inputs that maximize kernel ranges. The technique, presented in Algorithm 6, is also implemented as an LLVM-pass instrumentation. It takes as input the program $\mathcal{P}$ with an identified set of kernels $\mathcal{K}$, a set of program input ranges ($\mathcal{I}$), and a timeout (T). The algorithm is also parameterized by the number of mutations m and a constant c that determines the neighborhood radii for all program inputs from which mutants (new program inputs) are drawn. The algorithm returns a set of kernel ranges $[\{\mathcal{R}_{lo}\}, \{\mathcal{R}_{hi}\}]$ (line 16) with the goal of computing this interval as wide as possible within the timeout T.

The algorithm maintains a queue of input values, Q, which is used to execute the program on multiple input values. If the queue is empty, m new random inputs within the specified input range $\mathcal{I}$ are added to the queue (line 6,7). Otherwise, the algorithm dequeues one set of input values $\{v_1, \cdots, v_n\}$ from Q (line 9), executes the program $\mathcal{P}$ on these input values, and updates the kernel ranges as in vanilla blackbox fuzzing (line 10).

If the kernel ranges are updated, indicating that the current input value may have led to kernel inputs being outside of the known range, the algorithm generates $m - 1$ mutants from the dequeued input values by randomly selecting inputs from the neighborhood $\{v_1 \mp r_1, \cdots, v_n \mp r_n\}$ and adds them to the queue (line 12-14). The neighborhood radii $\{r_1, \cdots, r_n\}$, computed once on line 3, determine the distance of a mutant from the original input values and depend on the width of each input range. Larger input ranges correspond to larger neighborhoods.

To avoid getting stuck in a local maximum or minimum, the algorithm also generates one random input value for all variables within the specified input range (line 15). This step ensures that the search process explores a wider range of input values.

5.2.3 Fuzzing with AFLGo

The blackbox does not take into account the program structure. We thus evaluate an adaptation of *directed greybox* fuzzing, which is implemented in the state-of-the-art tool AFLGo [27]. AFLGo performs directed greybox fuzzing and thus can be directed toward specific program locations in order to explore as many different paths in the program as possible.

To use AFLGo to generate the kernel ranges, we need to instrument the input program to only consider numerical values within the specified bounds of program inputs. This is because AFLGo generates inputs, and we want to discard any inputs outside of the specified bounds that are not useful for computing kernel ranges. We instrument the kernel functions in the same way as for blackbox fuzzing as shown in Figure 5.2. We then run AFLGo on this instrumented code for a specified time bound, directing AFLGo towards exploring as many different paths in the program as possible. Finally, we return the set of kernel ranges as specified by $[\mathcal{R}_{min}, \mathcal{R}_{max}]$.

To guide AFLGo's exploration toward inputs that reach the numerical kernels, we specify the beginning of a kernel as a target location. However, traditional AFLGo, which mutates inputs to reach a specific target location while attempting to achieve as much coverage as possible, we use AFLGo to explore all possible inputs that reach a kernel in order to maximize the range of possible kernel inputs. We instrument the kernel functions to monitor all tests that reach the kernel, not just those that achieve new coverage and are added to the final test suite. However, without additional guidance, AFLGo may waste program inputs that do not update the minimum or maximum value of the kernel ranges.

In order to improve the efficiency of the process, we introduce an additional step where we iteratively attempt to extend the ranges using AFLGo. Initially, we run AFLGo on the program to obtain an estimate for the kernel ranges $[\mathcal{R}_{min}, \mathcal{R}_{max}]$. We then instrument code at the beginning of the kernel to check whether the kernel input is smaller than $\mathcal{R}_{min}$ or larger than $\mathcal{R}_{max}$ using a conditional statement, and use these statements as targets for the next step. This effectively directs AFLGo to find kernel inputs that are outside of the current estimate $[\mathcal{R}_{min}, \mathcal{R}_{max}]$. If AFLGo finds a valuation v_i that is smaller than $\mathcal{R}_{min}$ or greater than $\mathcal{R}_{max}$, we update the ranges accordingly and repeat this process until a specified time bound T is reached. Finally, we return the updated kernel ranges $[\mathcal{R}min, \mathcal{R}max]$.

5.3 Second Phase: Static Analysis with Daisy

After computing the kernel ranges $\mathcal{R}$, we use them as kernel input specifications (preconditions) and modify the state-of-the-art roundoff-error analyzer Daisy [55] to check for the absence of cancellation errors and special float values. We chose Daisy because it was easy to extend its analysis for cancellation detection and because it includes an optimization pass based on rewriting that we utilized to reduce the roundoff errors in the kernels as described in Section 5.3.2. The translation of numerical kernels and precondition annotations into Daisy's input language

in Scala is currently done manually but could be automated in the future.

Daisy's core roundoff error analysis involves a forward data-flow analysis that computes ranges and absolute error bounds for each intermediate arithmetic expression in an abstract syntax tree. As part of this analysis, it checks for overflows and invalid expressions that could lead to NaN values, as their absence is essential for a meaningful roundoff-error computation. Daisy employs the interval and affine arithmetic abstract domains as mentioned in the background section of this thesis (see Section 2.2) and also has the option to subdivide the input ranges to further reduce the over-approximations. Ultimately, Daisy provides a worst-case absolute error bound on the result value of the kernel.

Although the absolute error bound is limited in the absence of a global error analysis that propagates errors introduced in individual kernels, Daisy's computation of roundoff errors for all intermediate expressions enables us to verify possible cancellations at every arithmetic operation. To achieve this, we extend Daisy to check at every intermediate expression for possible cancellations using the ranges and absolute error bounds that Daisy computes. At each binary arithmetic operation, we compare the relative errors of the operands with the relative error of the operation result. If the relative error increases by more than a certain factor, we report an error.

To compute the relative error for an intermediate expression x, we divide its worst-case absolute error bound by the smallest value contained in the range of x. If the range of x ($[x]$) contains zero, we divide by a small constant c instead, $\frac{\Delta_x}{\max(c,\min([x]))}$, to ensure that the relative errors are always well-defined. Although this does not produce a sound bound on the relative error, it suffices for our purpose of making relative comparisons.

5.3.1 Analyze Spurious Warnings with CBMC

With our extension on top of Daisy, we can definitively prove the absence of cancellation errors for each kernel, given the specified kernel input ranges. In cases where our extension of Daisy cannot prove the absence, we issue a warning, allowing the user to inspect the reported intermediate expression.

Due to Daisy's over-approximations, our framework can result in both false positives for cancellation warnings and false warnings about potential NaN and Infinity values. While we have not found a tool that can more accurately detect cancellations, we can use a tool like CBMC [119] for precise reasoning to address the latter issue. CBMC can identify possible scenarios that may cause NaN or overflow and provide the user with a counterexample trace if the issue is genuine. If it is, the user can examine this trace to debug issues, if necessary.

5.3.2 Optimize the Kernels

Having access to the kernel input ranges also enables us to optimize the kernel itself. We utilize Daisy's expression rewriting and exploit the non-associativity and non-distributivity of

floating-point arithmetic to improve the numerical kernels locally. It is important to note that the kernel input ranges are essential for this optimization since the optimal re-ordering of arithmetic operations is dependent on the variable ranges.

5.4 Experimental Evaluation

In this section, we provide an overview of our benchmark suite consisting of real-world floating-point programs in Section 5.4.1. We then present a comprehensive survey of the current state-of-the-art static and dynamic analyses in Section 5.4.2 and evaluate their performance on our benchmark suite. Finally, we evaluate our two-phase framework in Section 5.4.3.

5.4.1 Benchmarks

From fluid-dynamics problems to the resolution of dense systems of linear equations, numerical programs are often written in C or C++. Between structure operations, dynamic memory management, and other programming constructs, such code does not only contain arithmetic operations. Instead, floating-point operations are usually clustered in what we call numerical kernels. Furthermore, those kernels may only be reachable through several function calls or loop iterations, which complicate their numerical analysis.

The FPBench benchmark collection [51] is a well-known suite of benchmarks used to evaluate numerical analyzers. However, since it only contains isolated kernels, it is unsuitable for our purpose, which is to analyze kernels within real-world numerical programs. Therefore, we have compiled a new representative set of numerical programs from various application domains to better reflect the complexity of real-world applications.

The selected benchmarks are C and C++ floating-point programs featuring arrays, structures, loops, branching, and function calls and running on a single thread. We consider specified bounds on the input variables of the program (program inputs) and identify a set of numerical kernels containing a large number of arithmetic operations. Table 5.1 provides an overview of the size in terms of the number of lines of code (LOC) and complexity of our benchmarks, as well as the number and arithmetic complexity of the kernels that we chose for verification. We also count the number of trigonometric operations (implemented in library functions) in the kernels, and the 'depth' column shows the number of function calls needed to reach the kernels from the program entry. We slightly modified the benchmarks by removing dynamic memory allocation, pointer arithmetic, and I/O operations, which are challenging for most program analyses. We explain each benchmark in detail below.

`arclength`: This program is a classic example of the arclength function [149] that calculates the distance between two points along an arc. It accepts one input, which is the number of arcs, and calculates the arclength of the specified points along those arcs. The kernel function comprises of 20 arithmetic operations and 1 call to the sine function.

benchmark	lang.	LOC	#in.	#func.	#loops	kernels			
						#	#arith op.	#trig. op.	depth
`arclength`	C	31	1	1	2	1	20	5	1
`linearSVC`	C	32	4	1	3	1	7	-	1
`raycasting`	C	94	2	4	3	1	4	-	4
`nbody`	C	108	21	10	9	2	9, 22	-,-	2, 2
`pendulum`	C	141	4	11	8	2	24, 42	2,11	4, 2
`fbenchV2`	C	215	8	2	5	2	6, 14	-,5	2, 2
`molecular`	C++	323	3	8	13	3	8, 12, 11	-,-,3	1,1, 1
`fbenchV1`	C	380	8	10	8	4	19, 6, 14, 36	-,-,5,-	5, 2, 2, 3
`reactor`	C++	467	4	11	2	3	14, 11, 13	-,2,2	2, 0, 1
`linpack`	C	544	5	12	31	1	8	-	2
`lulesh`	C++	2187	5	43	74	4	109, 77, 14, 41	-,-,-,-	6, 7, 6, 7

Table 5.1: Benchmark statistics

`linearSVC`: This program is derived from the training of a linear Support Vector Classifier using the Python sklearn library on the Iris standard dataset. We obtained the source code from the classifier using sklearn-porter [8]. We identified the predict function, which predicts the classification of test data, as the numerical kernel. The kernel comprises 7 arithmetic operations, and the program has 32 lines of code. It takes 4 inputs.

`rayCasting`: This benchmark program is a C-implementation of the ray-casting algorithm to determine if a point is inside or outside a polygon, sourced from Rosetta Code [5]. The program is written in C, spans 94 lines of code, and takes 2 program inputs. The numerical kernel of the program consists of a function that performs vector multiplication and addition, which is called several times at different stages of the program, after 4 function calls, and in various loops.

`nbody`: This program is also sourced from Rosetta Code [4]. The N-body problem involves simulating the interaction of multiple masses (N bodies) under gravity and illustrating their evolution over time. When $N > 2$, analytical solutions for this problem are not possible, and computer simulations are required. This C code (108 LOC) takes as input the number of masses and their positions (21 inputs for 3 bodies). For this benchmark, we identified 2 kernels, one consisting of 9 arithmetic operations and the other consisting of 22 arithmetic operations. These kernels calculate the velocity and acceleration of the bodies for a one-time step.

`pendulum`: This program simulates the inverted pendulum control problem, where the goal is to keep a pendulum upright. The original Python code [2] was converted into a C program consisting of 141 lines and 4 inputs. Within the function responsible for swinging the pendulum up, we have identified 2 kernels that use trigonometric operations. The first kernel can be reached after 4 function calls.

`fbench`: This benchmark is a ray-tracing algorithm used for optical design, which involves

heavy usage of trigonometric functions [1]. We have created two versions of this benchmark: one with user-defined trigonometric functions (V1) with 380 LOC and another with their library versions (V2) with 215 LOC. The user-defined trigonometric functions are part of the original code. We have identified 4 kernels for V1 and 2 kernels for V2. The two kernels present only in V1 are located inside the user-defined `sin` and `atan` functions, each with 19 and 36 arithmetic operations, respectively. It takes 5 function calls to reach the first kernel.

`molecular`: This C++ program, taken from [3], simulates molecular dynamics by tracking the paths of unconstrained particles that exert a distance-dependent force on each other. The program has 323 lines of code and takes three input variables. We have identified three numerical kernels that perform 8, 12, and 11 arithmetic operations, respectively.

`reactor`: This C++ program [6] simulates the shielding effect of a slab designed to absorb or reflect most of the radiation from a neutron source, demonstrating a Monte Carlo simulation approach. The program consists of 467 lines of code and takes 4 inputs. We have identified 3 numerical kernels with multiple arithmetic and trigonometric operations, with the first kernel reachable after 2 function calls.

`linpack`: Originally designed to benchmark the floating-point computation power of a computer system, this C program performs LU factorization of a dense matrix, uses it to solve a linear system, and measures the speed of the computation. In our context, the benchmark takes 5 input seeds to generate a 5×5 dense matrix. Within the program, we have identified a numerical kernel with 8 arithmetic operations that compute the dot product of two vectors. This kernel is called inside various loops in the program.

`lulesh`: This C++ program exemplifies the numerical algorithms, data movement, and programming styles typical in scientific applications. It approximates the hydrodynamics equations by partitioning the spatial problem domain into volumetric elements defined by a mesh. The code, which is derived from [114], has 2187 lines and 43 functions and takes 5 program inputs. We have identified 4 numerical kernels that compute the area and volume of a mesh. These kernels contain 109, 77, 14, and 41 arithmetic operations and can be reached after 6 or 7 function calls.

5.4.2 State-of-the-art on Floating-Point Programs

We begin by reviewing the current state-of-the-art techniques for analyzing floating-point computations. We broadly classify them into two groups: program verification and automated test generation. Table 5.2 summarizes the types of errors that these techniques can *in principle* prove or disprove, as well as the type of programs they can handle. However, the implementation of these techniques may be more limited than indicated in the table.

In the table, a check mark (✓) indicates that the technique can prove the presence (pres. column) or absence (abs. column) of special float values and large round-off errors, while a cross mark (✗) indicates that it cannot. We also distinguish between techniques that can handle entire programs, those that require specifications (pre- and postconditions) for functions, and those

Technique	Tool	Error Type				Program
		Special Values		Cancellation		
		pres.	abs.	pres.	abs.	
Bounded Model Checking	CBMC	(✓)	(✓)	✗	✗	whole
Abstract Interpretation	Astrée	✗	✓	✗	✗	whole
Deductive Verification	Frama-C	✗	✓	✗	✗	whole + spec
Roundoff Analysis	Daisy, FPTaylor	✗	✓	✗	✓	kernel
Dynamic Symbolic Execution	KLEE-Float	(✓)	(✓)	✗	✗	whole
Directed Greybox Fuzzing	AFLGo	✓	✗	✗	✗	whole
Guided Random Testing	S3FP, ATOMU	✗	✗	✓	✗	whole

Table 5.2: Comparison of different Floating-point Analysis Techniques and Tools

that can work with numerical kernels only.

We then select available representative tools implementing the techniques and evaluate them on our benchmarks w.r.t. applicability and scalability. We review additional existing tools in the related work in chapter 7.

Experimental Setup We selected representative tools for the techniques mentioned in Table 5.2 and used them to analyze our set of benchmarks. To use the tools, we annotated the input ranges for the programs and added assertions at the beginning of the kernel code to check for the absence of Infinity and NaN, using standard library functions like `isinf`' and `isnan`'.

All experiments were conducted on a Debian server system with 2.67GHz and 50 GB RAM, and we used 64-bit precision for the analyses. To ensure a fair comparison of results, we set a time budget of 1 hour for all the tools.

For bounded model checking, we used **CBMC** [119] version 5.12 with MiniSat 2.2.0 as the solver and a bound of 50 loop iterations only to verify the absence or presence of special values. While other Satisfiability Modulo Theories (SMT) solvers like CVC4 or MathSAT could also be used with CBMC, we found that CBMC performed better with MiniSat in our preliminary experiments.

For abstract interpretation, we used **Astrée** [135] to analyze the full program and verify the absence of special values. Astrée cannot detect cancellation errors. We used Astrée's `linux64_b5162300_release` for our experiments.

To use Astrée, we needed to modify the code as it does not allow us to assume the bounds of global variables. For benchmarks with bounds on global variables, we manually wrote wrapper functions that called the main functions with global variables and assumed the global bounds inside the wrappers. We used Astr'ee's interval domain as we did not observe a remarkable difference between the interval and octagon domains for our benchmarks.

Our preliminary experiments showed that Astrée tends to compute a prohibitively large

over-approximation for the benchmarks due to the presence of loops, for which it applies widening to compute a fixpoint. We used semantic loop unrolling for all individual loops to reduce over-approximation and make the analysis more precise. For the experiments, we unrolled all loops up to 50 iterations, i.e., the tool analyzed up to 50 iterations individually and used widening after that.

It is worth noting that Astrée's analysis can be extensively parameterized with knowledge of the program under analysis, which can make the analysis even more precise. However, this requires vast manual effort and knowledge of the intricacies of the program, which we did not perform for our experiments.

For directed greybox fuzzing, we employed **AFLGo** [27] downloaded on June 9, 2020, to detect special values in the numerical kernels. We searched for NaN or Infinity values in the variables of the kernels and created new branches in the program that halt the execution abnormally if such values occur. We configured AFLGo to generate inputs that target these new branches. If AFLGo reaches those branches, it reports a 'crash' along with the inputs that caused the crash.

Since AFLGo generates inputs using bit manipulations, we added checks at the beginning of the program to verify the type of inputs. We only proceeded with the execution if a generated input was a floating-point value within the defined input range.

All of the previously mentioned tools can be directly used on our annotated benchmarks; hence, we compare them in Section 5.4.2. We also attempted to use other tools listed in Table 5.2, but found that they either require extensive annotations or are not applicable to our benchmarks.

For example, Frama-C[116, 43], a state-of-the-art deductive verifier for C programs, requires annotations for all functions in a program and does not support transcendental library functions such as sine and exponent that are commonly used in numerical code, including our benchmarks.

Although the dynamic symbolic execution tool KLEE-Float [129] supports all of our benchmark constructs, it did not find any special float values within the specified time. S3FP [38] and ATOMU [182] are representative tools of guided random testing that target roundoff errors, but we found S3FP difficult to use as it requires the user to generate the high-precision ('real') version of the code for comparison thus requiring manual intervention, and ATOMU did not produce any results for our benchmarks within the specified time bound of 1 hour.

Results and Discussion Using CBMC and Astrée, we could only prove the absence of special float values in `linearSVC` and `rayCasting`, which are among the smallest benchmarks. Astrée was also able to prove the absence of special values in kernels 1 and 5 of `fbenchV1`. However, for all other C benchmarks, Astrée generated warnings for the potential existence of special values, and CBMC did not finish within the time limit. It may be noted that since November 2020, Astrée has added support for C++ programs. However, we leveraged Astrée's version `linux64_b5162300_release`, which did not support C++, and our work was submitted prior to the release featuring C++ support. Hence, our experiments do not include the results for C++ programs.

AFLGo, on the other hand, found special values in kernel 2 of both `nbody` and `pendulum`. These

results prove the presence of special values in those kernels with the given program input-bound specifications and also indicate that greybox fuzzing can be effective in detecting special values. We then modified the input-bound specifications of these programs using AFLGo and generated tighter ranges for these benchmarks to avoid special values. With the new ranges AFLGo did not find errors anymore. However, for both cases, Astrée still reported warnings.

Our findings in this section support the notion that current techniques are not efficient in verifying the absence or testing the presence of NaNs and infinities in real-world numerical programs. Static analysis techniques are either too imprecise or not scalable. While dynamic analysis techniques are more scalable, they cannot prove the absence of these errors. Furthermore, none of these techniques are capable of detecting cancellation errors.

5.4.3 Evaluation of The Two-phase Approach

We proceed to evaluate our two-phase approach and compare it with state-of-the-art tools under a 1-hour time limit. We allocate 50 minutes for generating the kernel ranges and 10 minutes for kernel analysis to ensure a fair comparison. We have empirically evaluated the impact of the time limit and observed that increasing the time does not affect the benchmark results significantly, while a smaller time limit yields worse results.

Comparison of Techniques for Kernel Range Computation The main step in our approach is to compute the kernel ranges, which we compare using four different methods: blackbox fuzzing (BB), guided blackbox fuzzing (GBB) (both implemented in Blossom), AFLGo with our iterative widening (AFLGo), and a combination of BB and AFLGo iterative widening (BB+AFLGo). After empirical testing, we have determined that GBB, with 5 mutants, performs the best for all our benchmarks.

For AFLGo, we first fuzz the program for 5 minutes and then run our iterative widening that employs the fuzzer in a refinement loop to widen the ranges obtained for the next 45 minutes (see Section 5.1.1).

For BB+AFLGo, we use Blossom's blackbox fuzzing for 25 minutes to generate the initial ranges. We then apply our range-widening technique with AFLGo for the next 25 minutes on these ranges.

To compare the computed kernel ranges, we calculate the width of each kernel range ($\overline{x} - \underline{x}$), and we present the average width over all kernel inputs and 5 runs with different random seeds in Table 5.3. Our goal with dynamic analyses is to expand the kernel ranges to cover as many kernel inputs as possible.

In addition to the ranges obtained using our methods, we also include the sound over-approximated ranges computed by Astrée, whenever available. While Astrée produces a warning *inside* the `arclength` kernel, it still computes a finite range for the kernel *input*.

Among the 7 kernels where Astrée reports non-trivial ranges, in 5 of them, our fuzzing techniques compute ranges that are similar to those of Astrée. In fact, for `rayCasting`, the ranges

	kernel-#vars	avg range width					ker.
		Astrée	BB	AFLGo	BB+AFLGo	GBB	anlys
`arclength`	1-1	6.16e+4	**3.14**	**3.14**	**3.14**	**3.14**	✓
`linearSVC`	1-4	3.73	**3.73**	3.71	3.72	**3.73**	(✓)
`rayCasting`	1-5	12.20	**12.20**	**12.20**	**12.20**	**12.20**	✓
nbody	1-6	∞	1.09e+5	6.67e+4	1.21e+5	**1.02e+8**	✓
	2-9	∞	1.25e+4	8.45e+3	1.19e+4	**8.91e+6**	✗
pendulum	1-4	∞	14.80	12.86	**14.82**	14.56	✓
	2-5	∞	22.38	17.61	**22.39**	22.16	✓
fbenchV2	1-5	24.60	**20.46**	**20.46**	**20.46**	**20.46**	✗
	2-5	∞	**21.36**	**21.36**	**21.36**	**21.36**	✗
fbenchV1	1-1	403.00	**0.18**	**0.18**	**0.18**	**0.18**	✓
	2-5	20.50	**20.46**	**20.46**	**20.46**	**20.46**	✗
	3-5	∞	21.36	21.36	**24.76**	21.36	✗
	4-1	1.57	**1.54**	**1.54**	**1.54**	**1.54**	✓
`linpack`	1-8	∞	3.60e+6	4.44e+3	3.60e+6	**2.11e+269**	✗
molecular	1-4	✗	**9.04**	**9.04**	**9.04**	**9.04**	✗
	2-6	✗	**1.86**	**1.86**	**1.86**	**1.86**	✓
	3-7	✗	**12.88**	**12.88**	**12.88**	**12.88**	✓
reactor	1-1	✗	**1.00**	**1.00**	**1.00**	**1.00**	✓
	2 -6	✗	1.43e+2	9.35e+1	1.43e+2	**1.46e+2**	✗
	3-1	✗	**2.50**	**2.50**	**2.50**	**2.50**	✓
lulesh	1-24	✗	**4.97**	4.80	**4.97**	4.95	(✓)
	2-18	✗	**6.09**	5.51	5.50	5.89	✓
	3-9	✗	**3.48**	3.09	3.42	3.25	✓
	4-12	✗	**5.95**	5.49	5.93	5.77	✓

Table 5.3: Comparison of kernel ranges generated by different techniques and settings

computed by both methods are exactly the same. However, in the remaining 2 kernels, Astrée reports large ranges while all the fuzzing techniques compute smaller ranges with the same width. This suggests that Astrée may be over-approximating the ranges or that the fuzzing techniques were not able to discover new inputs within the given time limit.

For the cases where Astrée finds unbounded ranges or cannot compute ranges, we observe that in all but three kernels, the four fuzzing techniques result in similar range widths. However, for three kernels, GBB discovers significantly larger ranges, suggesting that it is capable of finding outlier kernel inputs that other methods miss. Therefore, we conclude that guided

benchmark	kernel-#vars	BB	AFLGo	BB+AFLGo	GBB
`linearSVC`	1-4	-	2.21	-	-
nbody	1-6	121.05	312.93	144.86	181.26
	2-9	155.31	226.10	127.25	206.20
pendulum	1-4	0.69	51.77	0.57	5.25
	2-5	0.69	44.37	0.54	4.48
fbenchV2	1-5	-	-	1.99	-
	2-5	-	0.04	-	-
fbenchV1	1-1	-	0.03	-	-
	2-5	-	-	1.99	-
	3-5	-	0.04	8.85	-
`linpack`	1-8	0.01	100.15	-	114.58
`molecular`	2-6	0.25	8.0	0.15	0.33
reactor	1-1	-	0.01	-	-
	2-6	2.51	11.32	2.91	2.80
	3-1	-	0.01	-	-
lulesh	1-24	1.67	6.76	1.74	2.50
	2-18	4.28	19.73	15.59	6.96
	3-9	7.14	23.25	10.55	11.97
	4-12	3.91	16.13	3.49	5.88

Table 5.4: Variation of computed kernel range widths (from the average width) for our three fuzzing techniques (in %), '-' denotes no variation

blackbox fuzzing is the most suitable method for computing kernel ranges in our benchmarks.

AFLGo often computes the smallest ranges, possibly because its objective is to maximize the number of paths in the program to reach the target locations in the kernels. This approach may prioritize generating values to find new paths over generating values that exercise already-found paths, which could increase the width of the kernel ranges.

Effect of Randomness All fuzzing techniques, including BB, GBB, and AFLGo, rely on randomness. To assess how this randomness affects the computed kernel ranges, we calculate the variation of the range widths compared to the average range width per variable over 5 runs. For 7 kernels, we detect no variation in the range width for any of the methods. The remaining kernels and their variations are listed in Table 5.4. We have noticed that the benchmarks `nbody` and `linpack`, for which GBB has found very large ranges, show significant variations for all the methods, indicating that there may be some corner-case inputs that lead to large kernel ranges, and GBB is the only method that can reliably find them. Moreover, we have observed that AFLGo has larger variations due to randomness for a few additional benchmarks than BB

and GBB, whose variations are relatively small.

Conditional Kernel Verification Out of the 24 kernels, we were able to (conditionally) prove the absence of special floating-point values for 16 of them and (conditionally) prove the absence of cancellation errors for 14 kernels. The results are presented in the last column of Table 5.3. The symbol ✓' indicates that Daisy was able to prove both the absence of special values and cancellation errors for the kernel with the given kernel ranges. The symbol (✓)' indicates that Daisy was able to verify only the absence of special values, while the symbol '✗' indicates that Daisy reported a special-value warning. For the relatively small benchmarks `arclength`, `linearSVC` and `rayCasting`, our verification of the kernels is sound, i.e. unconditional, as we used ranges computed by Astrée.

Special-Value Warnings When Daisy reports a warning, it is not certain that a kernel can actually compute a special-value result for two reasons. Firstly, Daisy over-approximates the concrete program semantics, which means that its analysis may be more conservative than necessary and potentially report false positives. Secondly, the range we compute may contain values that are not feasible during program execution. Therefore, to assist developers in debugging warnings reported by the static analyzer, we use CBMC on those kernels.

CBMC was able to find counterexamples in all kernels for which Daisy reported warnings. However, upon further inspection of the code, we discovered that the counterexamples reported for `nbody` and `fbench` were not accurate for the specific program inputs we used. This was because the true input range for these kernels was discontinuous, and the reported counterexamples were for infeasible inputs. Specifically, for kernel 2 of `nbody`, a NaN could be produced if the simulated bodies collide, which did not occur with our chosen initial conditions. Similarly, the kernels in the ray-tracing algorithm of `fbench` could produce Infinity only if the ray was chosen in a very specific manner, which was not possible with our selected program input ranges.

For `linpack`, the reported arithmetic overflow is indeed genuine. This is because a division by zero can occur before the kernel if the input matrix contains a zero on the diagonal, leading to undefined behavior and a huge range of kernel inputs. Similarly, for `molecular` and `reactor`, arithmetic overflow can occur for a specific position of molecules and a specific value of the angle between the particle's direction and the X-axis, respectively.

It is worth noting that, with the counterexamples generated by CBMC, we could easily distinguish between spurious and genuine warnings. In future work, it may be worth considering refining the kernel monitoring approach to track multiple ranges per kernel, which could help detect discontinuous ranges.

Cancellation Errors Our extension of Daisy reports cancellation-error warnings for one kernel of `linearSVC` and one kernel of `lulesh`. We have used a threshold of 10^3 for reporting cancellation, i.e., if the relative errors of the operands and the result differ by more than three orders of magnitude, we report an error.

We inspected the kernel code and confirmed that the cancellation warnings are genuine, i.e., there are indeed inputs that will result in a large roundoff error. To confirm that these errors also occur for the actual program inputs, we inspected the histograms of the kernel range (running

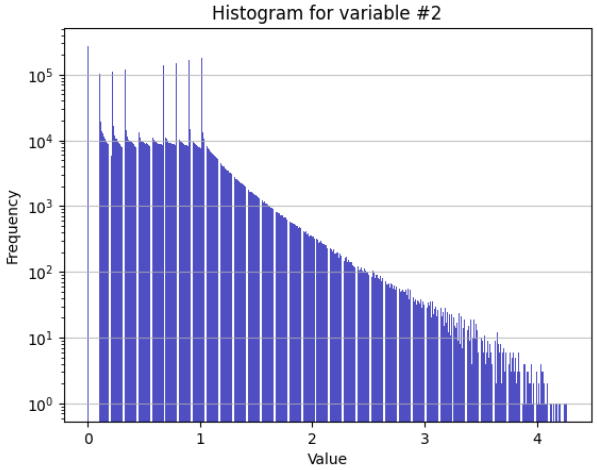

Figure 5.3: Histogram of the input values of one of the inputs of `lulesh`'s kernel 1

fuzzing for a very short time only to not overwhelm the plotting procedure). For example, for the `lulesh` kernel, Daisy raises a warning for a subtraction of two input values. The histogram for one of the 'offending' inputs is shown in Figure 5.3 and confirms that the range of inputs is continuous enough that the cancellation is indeed possible (the small gaps between the bars are an artifact of plotting).

Although the number of cancellations found may seem small, we suspect that this is the case because our benchmarks were mostly developed as reference or example programs. For example, `lulesh` was specifically designed as a representative hydrodynamics simulation code, and we expect it to have been carefully developed and tested.

Kernel Optimization We have additionally applied Daisy's rewriting optimization to kernels for which no possible special values were reported. This allowed us to reduce roundoff errors in eight kernels, with six cases being particularly notable. For example, we reduced the error by 9.5% for `linearSVC`, 7.1% and 3.3% for two outputs of kernel 2 in `pendulum`, 19.8%, 4.0%, 5.8%, and 5.8% for different kernel outputs of `lulesh`, and 33.3% for one output of `molecular`. These results suggest that the ranges inferred in the first phase are indeed useful for the analysis of numerical kernels.

5.5 Conclusion

Despite recent attention to floating-point analyzers, their focus has primarily been verifying or debugging arithmetic kernels. Our review of existing techniques and tools has revealed that few approaches with dedicated floating-point support are suitable for entire programs without significant user expertise. However, we have found that standard greybox fuzzing has been effective in identifying overflows and NaNs. Meanwhile, static analysis techniques for demonstrating the absence of special values and cancellation errors are still restricted to

programs with only a few bounded loops and numerical kernels, respectively.

Instead of attempting to scale up existing roundoff-error analysis tools to analyze entire programs, we proposed a novel approach that combines these tools with more scalable analyses to compute the kernel preconditions necessary for the roundoff analyses to be effective. We demonstrated how relatively small modifications to directed blackbox and greybox fuzzing techniques are sufficient to implement this framework. With these adaptations and changes to an existing roundoff-error analyzer, we were able to conditionally verify the absence of special values and cancellations in several numerical kernels in realistic floating-point programs that are currently beyond the reach of existing analyses. Our analysis is also precise enough to identify cases of cancellations. While our approach is not intended for certifying safety-critical systems, it still provides valuable debugging feedback for many real-world applications.

Sound Mixed Quantization of Neural Networks 6

Finite-precision programs deployed in embedded resource-constrained systems often encounter a tradeoff between accuracy and efficiency. These programs are typically designed with predefined accuracy bounds that specify acceptable levels of errors. However, optimizing the program to effectively use limited resources while staying within these error bounds poses a significant challenge. For example, using shorter precision can help reduce resources such as chip area, but it also introduces larger roundoff errors. Thus, striking the right balance between precision and resource utilization is important.

This work addresses the challenge of scaling up precision versus resource optimization for neural networks in embedded applications. With the increasing use of neural network-based closed-loop controllers in safety-critical applications, such as car and airplane models, adaptive cruise control, and aircraft collision avoidance systems [111], ensuring the correctness while optimizing the efficiency of these applications becomes crucial.

Recently, there has been a lot of work on automatically verifying the safety of limited-size but interesting NN systems [69, 68, 67]. These methods consider NNs trained in high-precision floating-point arithmetic on server-like machines equipped with graphics processing units (GPUs). They aim to estimate safe, reachable states, considering bounded errors stemming from noisy environments or inaccurate implementations. However, these methods do not verify whether the implementations of the NNs remain within the computed bounds.

Directly implementing these high-precision floating-point NNs on resource-constrained embedded systems is often infeasible. The utilization of high-precision floating-point arithmetic can be prohibitively expensive for embedded applications. Additionally, executing floating-point computations may require dedicated co-processors or software-based emulation capabilities, which can be impractical for many embedded applications. Hence, to obtain efficiency, e.g., in terms of area, latency, or memory usage, the trained NNs are quantized to use, for example, low-precision fixed-point arithmetic [92, 87].

The increased efficiency due to low precision comes at the cost of reduced accuracy of the NN computations since each operation potentially incurs a (larger) roundoff error. To ensure the overall correctness of systems with NNs, we thus need to choose the precision for quantization such that the safety of the overall system can still be guaranteed, i.e., the roundoff error is within the application-specific bounds that are established by safety verification.

Numerous approaches have been proposed for the quantization of NNs, demonstrating promising results on standard machine learning benchmarks [92, 87, 120, 163, 159, 146]. However, these techniques are not applicable to safety-critical closed-loop control systems for two primary

reasons. First, they are specifically designed for NN *classifiers*, focusing on dynamically comparing classification accuracy on specific test datasets. Consequently, they cannot handle NN controllers that implement regression tasks and compute (continuous) control values. Secondly, they lack soundness, meaning they cannot guarantee results for all possible inputs, which is essential for safety-critical systems.

On the other hand, existing tools that address sound quantization or precision tuning are primarily intended for general-purpose arithmetic code [39, 54]. These tools face limitations when applied to neural networks, either because they cannot work on fixed-point programs or due to challenges related to scalability and significant over-approximations.

In this chapter, we introduce the first sound and fully automated mixed-precision fixed-point quantizer that efficiently minimizes the number of bits needed to implement the NNs while guaranteeing provided error bounds with respect to idealized real-valued implementations.

Optimizing mixed-precision fixed-point arithmetic efficiently and accurately is challenging for two main reasons. First, fixed-point arithmetic is fundamentally discrete and the continuous abstractions that allow the use of efficient continuous optimization techniques for floating-point arithmetic [39] are not applicable. Secondly, due to the large choice (of combinations) of different precisions for individual operations when implemented on configurable hardware such as FPGAs, the search space for quantization is enormous and heuristic search based techniques [54] becomes impractical.

To address these challenges, we propose formulating the sound fixed-point quantization for feed-forward NNs as a mixed integer linear programming (MILP) problem. However, the direct formulation leads to computationally intractable non-linear constraints. To overcome this, we introduce the use of over-approximations, which relax the problem to linear constraints that can be efficiently solved. Furthermore, we leverage the special structure of NNs to efficiently yet accurately optimize the dot product operations by building on an existing technique for their correctly rounded implementation [61].

We have developed a prototype tool called Aster that takes as input a trained NN written in a small real-valued domain-specific language, a specification of the possible inputs, and an error bound on the result, and outputs a mixed fixed-point precision assignment that is guaranteed to satisfy the error bound (wrt. the real-valued input) and to minimize a (customizable) cost function.

While our implementation specifically targets fully connected feed-forward deep NNs with ReLU and linear activations, the proposed technique is fundamentally not limited to this specific architecture; it can handle other networks that involve (sparse) matrix multiplications and linearized activations.

We conducted a comprehensive evaluation of Aster by applying it to several embedded NN controllers that are commonly used for verification from the literature [131, 111, 90] and compared it with an existing sound quantizer [54]. Our results demonstrate the effectiveness of Aster, as it successfully generates feasible implementations for a significantly larger number of benchmarks than Daisy. When compiled for an FPGA using Xilinx Vivado HLS [171], the

implementations generated by Aster require fewer machine cycles. Moreover, Aster shows substantial improvements in optimization time, particularly when dealing with larger networks comprising thousands of parameters.

Contributions To summarize, in this work, we present:

a) a novel MILP-based mixed fixed-point quantization approach that guarantees a given roundoff error bound,

b) an experimental comparison against the state-of-the-art to demonstrate the effectiveness of our approach on a benchmark set of NN controllers and

c) a prototype tool called 'Aster', which is publicly available on Zenodo (see `https://doi.org/10.5281/zenodo.8123416`).

The content of this chapter is based on our paper '*Sound Mixed Fixed-Point Quantization of Neural Networks* [169], co-authored with Clothilde Jeangoudoux, Anastasia Volkova, and Eva Darulova. Clothilde Jeangoudoux contributed to the initial experimental setup, while Anastasia Volkova and Eva Darulova provided extensive feedback on the problem formalization and the writing and editing of the paper.

6.1 Overview

Before presenting our sound quantization technique in detail, we provide an overview of the problem and our proposed solution with an example.

Unicycle Model Consider a unicycle model of a car [111] that models the dynamics of a car with 4 parameter variables—the Cartesian coordinates (x, y), the speed, and the steering angle. We use a NN that was trained as a controller for this model as our example. This model is a fully connected feed-forward NN with 1 hidden layer. The input of the network is a 4-parameter vector, denoted by $\overline{x}_0 = \begin{bmatrix} x_0^1 & x_0^2 & x_0^3 & x_0^4 \end{bmatrix}$ where each input parameter is constrained by real-valued intervals as shown below specifying valid input values to the network:

$$x_0^1 \in [-0.6, 9.55], \quad x_0^2 \in [-4.5, 0.2], \quad x_0^3 \in [-0.6, 2.11], \quad x_0^4 \in [-0.3, 1.51]$$

The inputs are propagated through each layer by successive application of the dot product operations, bias additions and activation functions. Specifically, the NN performs the following computations:

$$\text{layer 1:} \quad \overline{x}_1 = ReLU(W_1 \cdot \overline{x}_0 + \overline{b}_1) \qquad \text{output:} \quad \overline{x}_2 = Linear(W_2 \cdot \overline{x}_1 + \overline{b}_2) \tag{6.1}$$

$\overline{x}_1$ is the output of the first layer, which is then fed as input to the next output layer. The output of the overall network is $\overline{x}_2$. Each layer is parameterized by a weight matrix W_i, a bias vector $\overline{b_i}$, and an activation function α_i. We have '`ReLU`' and '`linear`' activation functions and the following

weight and bias parameters for the network:

$$layer1: \quad W_1 = \begin{bmatrix} -0.037 & \cdots & 0.129 \\ -0.003 & \cdots & 0.099 \\ -0.128 & \cdots & 0.047 \\ 0.045 & \cdots & -0.166 \end{bmatrix}, \quad \overline{b}_1 = \begin{bmatrix} 0.028 & \cdots & 0.342 \end{bmatrix}$$

$$output: \quad W_2 = \begin{bmatrix} 0.052 & 0.137 \\ \cdots & \cdots \\ -0.200 & 0.154 \end{bmatrix}, \quad \overline{b}_2 = \begin{bmatrix} 0.273 & 0.304 \end{bmatrix}$$

While the network has only a single hidden layer, it has 500 neurons (elided above) and thus results in a non-trivial size of the overall network.

In order to deploy such an NN controller in a safety-critical system, it is necessary to prove the safety before the actual implementation. For instance, one may need to prove that the system reaches a set of safe states within a given time window [111]. Since exact reasoning about finite-precision arithmetic does not scale, existing verification techniques assume real-valued parameters and arithmetic operations for the network [107, 98, 164, 170]. Still, they can typically deal with *bounded* uncertainties from the implementation or measurements. Hence, we assume a bound on the output error ϵ_{target} is provided.

As a controller is primarily executed on resource-constrained hardware, using floating-point arithmetic may be overly expensive, as it requires either additional floating-point processor support or slow software emulations. For this reason, NN controllers are generally quantized to use fixed-point arithmetic.

Fixed-point Quantization In a fixed-point implementation, all program variables and constants are implemented using integers, and have an (implicit) representation $<Q, \pi>$, consisting of the total word length $Q \in \mathbb{N}$ (overall number of bits including a sign bit), and the position of the binary point $\pi \in \mathbb{N}$ counting from the least significant bit. Further details about fixed-point arithmetic and commonly used representation of arithmetic operations can be found in the background section (Section 2.1.2) of this thesis and in the work by Yates et al.[176].

This representation effectively divides the number of overall bits into an *integer part* $I = Q - \pi - 1$, and a *fractional part* π. The integer part must have sufficient bits to accommodate large enough values and avoid overflow, resulting in a representable number range of $[-2^I, 2^I]$. The fractional part controls the *precision* of a variable or an operation, where a higher number of fractional bits allows for more precise value representation. Assuming that each operation uses truncation as the rounding mode, the maximum roundoff error with π fractional bits is $2^{-\pi}$.

It may be noted that in this work, we use truncation as the rounding mode as it is the most commonly used one (default in circuit design and synthesis compilers such as Xilinx) and its runtime efficiency compared to other rounding modes like rounding to the nearest.

In our example (Equation 6.1), the input x_0^1 is in the range $[-0.6, 9.55]$. The number of integer bits required to hold this range, i.e., to represent the maximum absolute value 9.55 without

overflow, is 5 (including the sign bit). Assuming 32 bits are available for wordlength ($Q = 32$), then we have $(32 - 5) = 27$ bits remaining for the fractional part. Hence, the maximum roundoff error for the input is 2^{-27}.

We need to assign each operation enough integer and fractional bits to guarantee *overflow-freedom* and sufficient *overall accuracy* of the final result. To ensure that the overall roundoff error does not exceed a specified target error bound ϵ_{target}, it is necessary to propagate and accumulate the errors that occur in individual operations. However, this process can be complex and challenging to perform manually. Additionally, to optimize resource usage, we want to assign only as many bits as are actually needed.

Mixed-Precision Tuning Using a uniform fixed-point precision, where the same word length is applied to all operations, may not be optimal. For instance, if a single precision is insufficient at a certain point in the program, we need to upgrade *all* operations to a higher precision to guarantee the error bound. However, not all layers or operations have the same impact on the overall accuracy of the program. Therefore, it can be more resource-efficient to assign different precisions (word lengths) to different operations to achieve a target error bound. This approach, known as *mixed precision* optimization, allows for a more flexible and efficient implementation of the model.

Existing sound techniques applicable to fixed-point arithmetic rely on a heuristic search that repeatedly evaluates roundoff errors for different precision assignments [54]. While this technique works well for small programs, the repeated global roundoff error analysis quickly becomes expensive, as we show in our evaluation in Section 5.4. Moreover, these techniques are designed for general-purpose straight-line programs, and do not take into account the unique structure of NNs. To utilize these tools, one must assign individual scalar variables to all weight matrix and bias vector elements and unroll all loops, resulting in a significantly complex straight-line expression.

For the unicycle example, only computing the roundoff error for a uniform 32-bit implementation using the state-of-the-art tool Daisy [54] takes 5.91 minutes, and mixed-precision tuning of this example takes *more than* 2.7 *hours*.

Our Proposed Technique We present an *optimization-based* technique for generating a mixed-precision fixed-point implementation that is guaranteed to meet a given error bound ϵ_{target} and ensures no overflow. We encode this tuning as a mixed integer linear programming (MILP) problem for which efficient solvers exist. The mathematical model for the entire problem will result in fundamentally *nonlinear* constraints. To be able to utilize the efficiency of modern MILP solvers, we perform several over-approximations to generate a linearized problem. We explain the details of linearization in Section 6.2). Our MILP constraints optimize the number of fractional bits such that overflow freedom is ensured even in the presence of errors and a cost function is minimized.

Our approach is parametric in the cost function to be optimized as long as it can be expressed as a linear expression. Since the actual cost is highly dependent on the target application and hardware for the purpose of evaluation we consider the generic cost function implemented in

the state-of-the-art tool Daisy that counts the cost as the total number of bits needed.

Assuming $\epsilon_{target} = 0.001$ as the error bound for the unicycle example, Aster with our MILP-based mixed-precision tuning assigns different precisions (using 18, 19, 20, 21, 24, 30, and 34 bits) to different variables, constants, and operations in just under 50 seconds (i.e. significantly less than the 2.7 hours taken by Daisy). When compiled for an FPGA architecture with Xilinx Vivado HLS tool, Aster's generated code takes *only* 27 *machine cycles* to execute, whereas Daisy's generated code, both uniform and mixed-precision, take 178 cycles.

6.2 MILP-Based Mixed-Precision Tuning

In this work, we specifically target feed-forward NNs with 'ReLU' and 'linear' activation functions that are designed for solving regression problems and computing continuous outputs. Such networks may, for instance, be utilized in closed-loop control systems, where their safety can be proven using various techniques [107, 98, 164, 170] that typically assume real-valued arithmetic operations, inputs, and constants. Our goal is to optimize the resource usage of the *implemented* networks while maintaining safety by ensuring that the computed (control) outputs remain within a specified error bound (that e.g., arises from a safety proof).

Problem Definition Given a real-valued NN architecture, ranges of inputs, and a roundoff error bound at the output, the goal is to generate a fixed-point mixed-precision NN implementation that minimizes the precision of the variables and constants while ensuring that the roundoff errors at the output remain within the specified bound.

Our approach is inspired by successful mixed-precision optimization techniques for *floating-point* programs [39] that phrase precision tuning as an *optimization* problem (which it ultimately is). They rely on the dynamic ranges of floating-point arithmetic that allows bounding floating-point roundoff errors by nonlinear continuous abstractions, which, in turn, enables continuous, purely real-valued optimization techniques for precision tuning.

However, such *continuous* techniques cannot be applied to fixed-point arithmetic programs because in fixed-point, the ranges of individual operations need to be fixed at compile time, i.e., the number of integer bits for each operation has to be pre-determined for all possible inputs. Hence, a continuous abstraction is not possible.

Furthermore, while floating-point arithmetic typically supports only a limited number of precisions (such as 16, 32, or 64 bits), fixed-point arithmetic allows any number of bits to be used for any operation and a more precise encoding is necessary to capture this.

We can capture different numbers of bits precisely by using *integer* constraints. However, at the same time, we also need to guarantee that a target error bound is satisfied and this error is a non-integer. Hence, we choose to encode fixed-point precision tuning as an *Mixed-Integer Linear Programming* (MILP) problem [15]. In MILP, some decision variables are constrained to be integers, and other variables can be real-valued. Integers allow us to directly encode the discrete

decisions about how many bits to use for operations, and real-valued constraints effectively express error constraints.

The primary constraint of our optimization problem guarantees that the total roundoff error remains inside the target error bound. At the same time, we also need to ensure no overflow. Encoding these two conditions together would result in *non-linearity* because of the dot operations in a NN that perform multiplications. Unfortunately, only linear constraints can be efficiently handled by state-of-the-art MILP solvers; nonlinear mixed integer arithmetic is, in general, NP-hard, with non-convex problems being undecidable. Hence, we soundly over-approximate and linearize constraints as and when necessary to make our solution efficient and feasible.

There are two primary sources of non-linearity in the optimization problem: 1) computation of ranges of all arithmetic operations in order to ensure that there is no overflow, and 2) optimization of fractional bits for all variables and constants to guarantee the error bound.

To avoid the first case, we pre-compute a sound over-approximation of real-valued ranges for all operations efficiently using interval arithmetic (Section 6.2.1). From these, we compute the integer bits needed to represent the real-valued ranges. However, we still need to ensure that the finite-precision ranges (real-valued ranges + roundoff errors) do not overflow. We encode this as a linear range constraint in our optimization problem (Section 6.2.2), thus ensuring no overflow even in the presence of errors.

For the latter, instead of optimizing for all intermediate variables and constants individually, we treat the dot product as a single operation and encode only the assignment of the fractional bits of the dot product *results*.

However, to generate the final implementation, we still need to assign precisions to all intermediate variables and constants such that the roundoff errors in intermediate computations do not affect the result. To achieve this, we leverage existing techniques [61, 28] to determine the number of fractional bits required for the intermediate variables in a dot product to guarantee an overall error bound. We discuss these techniques in more detail in Section 6.2.3

Provided that the cost function is also linear, we can then encode the precision tuning problem for NNs with ReLU and linear activation functions with purely linear constraints. Linear constraints can be solved efficiently by state-of-the-art MILP solvers.

As an additional performance optimization, we choose to assign uniform bit lengths for the operations within each individual layer. That is, within a layer, all dot products are assigned the same bit length (and similarly for bias and activations). However, the bit lengths can vary from layer to layer. This choice helps to limit the number of constraints in our approach, leading to improved efficiency. It is important to note that this choice is made for optimization purposes and is not a fundamental limitation of our approach.

Overall, our technique consists of three steps:

1) computing the integer bits of program variables using interval arithmetic,

2) optimizing the fractional bits of dot, bias, and activation operations by reducing it to a

$$\begin{bmatrix} x_1^1 \\ x_1^2 \end{bmatrix} = ReLU\Bigg(\underbrace{\begin{bmatrix} 0.1 & 0.2 \\ 0.2 & 0.15 \end{bmatrix}}_{W_1} \cdot \begin{bmatrix} x_0^1 \\ x_0^2 \end{bmatrix} + \underbrace{\begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix}}_{\overline{b_1}} \Bigg), \quad x_2 = Linear\Big(\underbrace{\begin{bmatrix} 0.1 & 0.2 \end{bmatrix}}_{W_2} \cdot \begin{bmatrix} x_1^1 \\ x_1^2 \end{bmatrix} + \underbrace{\begin{bmatrix} 0.5 \end{bmatrix}}_{\overline{b_2}} \Big)$$

Figure 6.1: Running example for MILP modeling

Mixed Integer Linear Programming problem and

3) computing the precision of all constants and intermediate variables in dot products.

Our tool, Aster, performs all these steps fully automatically. We will explain each of these steps in detail next. Though Aster is primarily designed for continuous feed-forward NN controllers, our proposed technique can be extended to other types of NNs that have sparse matrices and other non-linear activation functions, such as sigmoid that can be piece-wise linearized [175].

Running Example We will illustrate our approach using the following small (artificial) NN presented in Figure 6.1 as our running example: The input to the network is a vector $\overline{x_0}$ consisting of 2 elements, whose ranges are provided by the user: $x_0^1 = [-10, 10]$ and $x_0^2 = [-5, 5]$. The NN produces a single output x_2. To generate an implementation of the network, the user needs to provide the network architecture (i.e., the weight matrices and bias vectors, as well as activation functions for each layer) as input to Aster. Additionally, the user specifies the precision of the input and if the input is represented *exactly*, which incurs no roundoff error in the input. The precision of inputs is typically known from the specification of sensors or similar in the context of embedded systems, but our approach also supports inputs with no initial roundoff errors.

Assume, for the running example, the inputs are exactly represented with 10 bits. The input specification implies that 5 bits are required to represent the maximum absolute value of the ranges, which is 10, leaving 5 bits for the fractional part of the inputs, with no initial roundoff error. The goal is to generate a quantized mixed-precision implementation of the NN such that its cost is minimized and the output error is bounded by a user-specified target error $\epsilon_{target} = 0.1$.

6.2.1 Step 1: Computing Integer Bits

Aster starts by computing the integer bits for all program variables and constants using a forward data-flow analysis that tracks the real-valued ranges at each abstract syntax tree (AST) node. For this purpose, Aster utilizes the widely used and efficient interval arithmetic. We have presented the details of interval arithmetic in Section 2.2.1. In this chapter, we only show how we compute intervals for the activation functions:

$$ReLU(x) = [\mathtt{max}(lo, 0), \mathtt{max}(hi, 0)], \;\; Linear(x) = [lo, hi]$$

Here *lo* and *hi* denote the lower and upper bounds of the interval of x, respectively. With *ReLU*, the resulting interval is the same as the interval of x directly if both *lo* and *hi* are positive; otherwise, it returns 0 for the negative part. After applying the *linear* activation, the interval is the same as the input x.

Given the initial ranges of the inputs, Aster uses interval arithmetic to propagate the intervals through the program and computes an interval for all variables, constants, and intermediate results that soundly over-approximates the real-valued ranges. Finally, Aster uses a function `intBits` [1] that computes, in the two's complement binary representation, the number of integer bits needed for these real-valued ranges such that all possible values can be represented without overflow.

In principle, the finite-precision ranges, i.e., real-valued ranges *with* roundoff errors, may need more integer bits to be represented than the real-valued ranges alone. However, as the errors are usually small, they typically do not affect the integer bits in practice and are thus a good estimate. In addition, our MILP constraints (detailed next) ensure that these initial estimates of the integer bits are sufficient to avoid overflow, i.e., these estimates only need to be good approximations. These integer bits are later added with the fractional bits to compute the total word length required to represent each program variable and constant.

For our running example in Figure 6.1, given the input ranges, Aster computes the real-valued ranges of the first layer as [-3, 3], and [-5, 5] after the dot operation and the bias addition, respectively. From this, Aster determines that 3 and 4 integer bits are needed to represent the signed integer part of the dot operation and bias addition, respectively, at layer 1. It analogously computes the integer bits for all operations in all layers.

6.2.2 Step 2: Optimizing Fractional Bits

Next, we reduce the problem of computing the fractional bits of dot product results, the addition of bias, and the activation functions to a mixed integer linear programming problem. We first provide an overview of the relevant variables that we will use to formulate our MILP problem in Table 6.1. The variables specific to a layer i are referenced along with the subscript i and op can denote the dot product, the bias addition, or the activation function.

We divide these inputs into four categories:

a) **User inputs** The variable k_i, which denotes the number of neurons at layer i, is deduced by Aster from the NN given by the user. The user directly provides the remaining values. The variables Q_{W_i} and Q_{x_0} denote the word length of the maximum weight and the largest input from the input vector $\overline{x}$. The input word length is used as is. However, the word lengths of maximum weights are required in the beginning only to make the optimization problem linear. We later deduce these word lengths automatically considering the result's

[1] `intBit(x) = binary(abs(x).integerPart).numOfDigits`, where it converts the integer part of x into 2's complement binary representation and then determines the number of digits in the binary representation.

user inputs	
k_i	number of neurons at layer i
Q_{W_i}	word length of maximum weight at layer i
Q_{x_0}	maximum word length of inputs
lo	lower bound on fractional bits π
hi	upper bound on fractional bits π
pre-computed inputs	
R_i^{op}	real-valued ranges of operation op at layer i
I_i^{op}	number of integer bits of the finite-precision range of operation op at layer i
A_i^{op}	maximum representable ranges of operation op without overflow at layer i
decision variables	
ϵ_i^{prop}	propagation error at layer i
ϵ_i^{new}	new roundoff error at layer i
γ_i^{op}	cost of an operation op at layer i
π_i^{op}	number of fractional bits of op at layer i
other variables	
b_i^{op}	indicator variable for operation op at layer i
opt_i^{op}	optimal variable for π^{op} at layer i

Table 6.1: Variable notations and definitions (op: dot / bias / activation)

precision of the dot product, which we will explain in Section 6.2.3. The user additionally specifies a range for the fractional bits $[lo, hi]$ (one for all operations and layers) that will be considered during optimization.

b) **Pre-computed inputs** The values of these variables are pre-computed by the first step. R_i^{op} and I_i^{op} denote the real-valued range and the number of integer bits required for the finite-precision range of each operation op at layer i. A_i^{op} denotes the maximum representable range of each operation op without overflow at layer i.

c) **Decision variables:** The decision variables are those for which the MILP problem will be solved. The variables ϵ_i^{prop} and ϵ_i^{new} are the error variables representing the propagation error and newly introduced error at layer i. These variables constitute the error constraint. γ_i^{op} represents the cost of each operation op at layer i where op ranges over dot, bias, and activation function and is used to formulate the objective function. The variables π_i^{op} denote the fractional bits of all operations at layer i.

d) **Other variables:** The other variables are used internally by Aster to linearize non-linear constraints (without over-approximations). The indicator variables are used to select the optimized fractional bit length, and the optimal variables are used to store the corresponding errors. We show these variables in Figure 6.3 and provide a detailed explanation of their purpose when discussing the process of linearization.

$$
\begin{aligned}
&\textbf{minimize:} \quad \gamma \quad = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha} \\
&\text{where,} \quad \gamma_i^{dot} = k_i * k_{i-1} * Q_{W_i} * \pi_i^{dot} \\
&\qquad\quad \gamma_i^{bias} = \max\left(\pi_i^{dot}, \pi_i^{bias}\right) \\
&\qquad\quad \gamma_i^{\alpha} \;= \pi_i^{bias} \\
&\textbf{subject to:} \\
&C_1(\textbf{error}) : \epsilon_n = \sum_{i=1}^{n} (\epsilon_i^{prop} + \epsilon_i^{new}) \leq \epsilon_{target} \\
&\text{where,} \quad \epsilon_i^{prop} \leq 2^{-\pi_0}, \text{ when } i = 1 \\
&\qquad\quad = \left(\max_{j=1}^{k_i} \sum_{l=1}^{k_{i-1}} W_{i,jl}\right) * \epsilon_{i-1}, \text{ when } i > 1 \\
&\qquad\quad \epsilon_i^{new} \leq 2^{-\pi_i^{dot}} + 2^{-\pi_i^{bias}} \\
&C_2(\textbf{range}) : \\
&\forall i \in n, \quad A_i^{dot} \geq R_i^{dot} + \epsilon_i^{prop} + \epsilon_i^{dot} \\
&\forall i \in n, \quad A_i^{bias} \geq R_i^{bias} + \epsilon_i
\end{aligned}
$$

Figure 6.2: The MILP formulation with non-linear constraints

Problem Formulation Our objective is to minimize a cost (function γ) subject to error and range constraints denoted by C_1 and C_2, where the constraints ensure that the user-provided error bound is respected and that no overflow occurs, respectively. Figure 6.2 presents the overall formulation of the MILP constraints for optimizing fractional bits. We provide a detailed explanation of each constraint; we emphasize the references to individual constraints in Figure 6.2 using underlines. We explain each part of these constraints next.

Cost Function Each layer has three operations: first, the dot product is computed, then the bias vector is added, and finally, an activation function is applied. We compute the costs of these operations individually and add them up to compute the total cost of each layer. The total cost over all layers is then computed by adding up the costs of n layers (see γ in Figure 6.2).

This cost function closely follows previous work [109] and computes the total number of bits needed to implement the NN. Our approach supports other cost functions, e.g., capturing performance or energy, as long as they can be expressed as linear expressions. Since the latter is highly hardware-dependent, we implement the cost function used in previous work.

The cost of the dot operation at layer i, denoted by γ_i^{dot}, depends on the number of inputs k_{i-1}, the number of neurons k_i, and the weight matrix W_i of the layer. The dot product at layer i is defined by expanding it into multiplications and additions of the weights and inputs and adding them for all k_i neurons of the layer:

$$dot_i = \sum_{j=1}^{k_i} \left(\sum_{l=1}^{k_{i-1}} W_{i,jl} \times \overline{x}_{i,l} \right)$$

Recall that our approach takes as input the *maximum* word length of the weights Q_{W_i} for each layer. With the Q_{W_i} and the fractional bits of the dot product π_i^{dot} (decision variable), we can over-approximate the total cost of the dot operation at layer i by multiplying it with Q_{W_i}, the number of neurons of the previous layer (k_{i-1}) and the current layer (k_i) (γ_i^{dot} in Figure 6.2).

For the first layer, we consider the number of input variables instead of k_{i-1}. Note that we only use an estimate of the maximum word length of W, as the actual word lengths of W are not known beforehand, and defining them as variables makes the cost non-linear. We later assign precisions for W from the solutions of the MILP problem, which we explain in Section 6.2.3.

As the bias vector is only added with the result of the dot product, the cost of this addition denoted as γ_i^{bias} is the maximum of the fractional bits of the result of the addition and fractional bits of the dot operation (γ_i^{bias} in Figure 6.2). Technically, the '`max`' function is also non-linear. We show linearized constraints for this function in Figure 6.3.

Finally, the cost of the activation is a function of π_i^{bias}. We assume ReLU and linear activation functions for the networks. As these two activation functions are linear functions of π_i^{bias}, the cost is the same as π_i^{bias} (γ_i^{α} in Figure 6.2).

Let us assume for our running example in Figure 6.1 that $Q_W = 8$ for both layers. The network has 2 input variables, 2 neurons in the first layer and 1 neuron in the output layer. With our formulation, the cost of the whole network is defined as follows:

$$\underbrace{(2 * 2 * 8 * \pi_1^{dot}) + \max(\pi_1^{dot}, \pi_1^{bias}) + \pi_1^{bias}}_{\text{layer1}} + \underbrace{(2 * 1 * 8 * \pi_2^{dot}) + \max(\pi_2^{dot}, \pi_2^{bias}) + \pi_2^{bias}}_{\text{output}}$$

Error Constraint The error constraint C_1 in Figure 6.2 states that the overall roundoff error of the full network ϵ_n needs to be bounded by the user-specified error ϵ_{target}. For this constraint, we need to express the roundoff error as a function of the precisions of individual operations.

To compute the roundoff error at each layer i, we need to track the *propagated error* ϵ_i^{prop} from the previous layer and compute the *new roundoff error* ϵ_i^{new} committed by the operations at layer i. The total roundoff error ϵ_n is then defined as the sum of all the errors at all layers.

The initial error ϵ_0 is considered as the propagated error at layer 1. This error ϵ_0 is bounded by the function of fractional bits of inputs (ϵ_1^{prop} in Figure 6.2) that is determined by the word length of the inputs Q_{x_0} and the integer bits needed to represent the range of the inputs R_{x_0} computed using the `intBits` function: $\pi_0 = Q_{x_0} - \texttt{intBits}(R_{x_0})$.

The propagation error at layer > 1 depends on the errors from previous layers as well as the absolute magnitudes of the weights (a weight that is bigger than one will magnify an existing error). Thus, the propagation error ϵ_i^{prop} for layer > 1 is defined as the error at the previous layer ϵ_{i-1} multiplied by the sums of the absolute weights of each neuron. In practice, we sum the absolute weights of each neuron (absolute sum over the number of inputs k_{i-1}) and take the maximum result as a factor to amplify the propagation error (maximum over the number of neurons k_i) (see ϵ_i^{prop} in Figure 6.2).

This is a sound over-approximation of the total propagated error as we assume the maximum magnification of errors for all neurons in the previous layer. For this constraint, Aster computes

the maximum values of the weights over the neurons for each layer beforehand and uses them as constants in the optimization problem, thus preserving linearity.

The new roundoff error at layer i is defined as the sum of the errors for the activation function, for the dot computation, and for the bias addition. The ReLU activation function ($\alpha(x) = \max(0, x)$) and the linear activation function ($\alpha(x) = x$) do not affect the error. Thus the new error is bounded by $2^{-\pi_i^{dot}}$ and $2^{-\pi_i^{bias}}$, where $\pi_i^{dot} = Q_i^{dot} - I_i^{dot}$ and $\pi_i^{bias} = Q_i^{bias} - I_i^{bias}$. For the dot and bias operation, the error is computed considering the fractional bits of these operations.

This error constraint is non-linear ($2^{-\pi_i^{op}}$). We linearize the error constraint for the optimization *exactly* by considering the user-provided range $[lo, hi]$ for π_i^{op}. We explain the linearization constraints in Figure 6.3. We also assume a roundoff error on the *result* of the dot product only and do not account for the roundoff errors of the individual operations of the dot product in our MILP constraints. We do this to avoid nonlinearity in the constraints, and rely on the fact that the dot product can be computed with faithful rounding up to the chosen format using the technique from de Dinechin et.al. [61]. We explain the details of computing the intermediate formats in Section 6.2.3.

Let us derive the error constraint for our running example in Figure 6.1. We considered no input error here: $\epsilon_0 = 0.0$. We first compute the total error in layer 1. This error is then considered as the propagated error for the output layer by magnifying it with the maximum absolute sum of weights: $\max((0.1 + 0.2), (0.2 + 0.15)) = 0.35$. Thus the overall error constraint is (assuming $\epsilon_{target} = 0.1$): $(2^{-\pi_1^{dot}} + 2^{-\pi_1^{bias}}) * 0.35 + (2^{-\pi_2^{dot}} + 2^{-\pi_2^{bias}}) \leq 0.1$.

Range Constraint As we have only considered the real-valued ranges for computing the integer bits of the variables and constants in the first phase, we must ensure that the finite ranges after each operation in each layer do not overflow. Our hypothesis is that the roundoff errors are small enough to keep the integer bits unaffected. Accordingly, in our range constraint C_2 in Figure 6.2, we want to ensure that the number of integer bits required to represent the finite-precision range after each operation I_i^{op} is enough to store the integer bits of the finite-precision result (real-valued ranges together with the roundoff errors).

However, directly implementing this constraint would result in applying the function `intBits` on error decision variables, which is non-linear. Hence, we reduce the problem of computing the integer bits to ensure the ranges of the finite-precision result remain inside the maximum representable range (A_i^{op}) with the integer bits of the real ranges after every operation at layer i.

We pre-compute the integer bits required for the real-valued result and use them to generate the maximum representable range after each operation. For the dot operation, we have the propagation errors ϵ_i^{prop} from the previous layer, and the operation itself introduces the new roundoff error ϵ_i^{dot}. Hence, after the dot operation, the finite range includes these two errors. For bias addition, however, we have the error from the dot operation as a propagated error along with the new roundoff error ϵ_i^{bias} introduced by the addition. The total error after the bias addition is essentially the total error ϵ_i of the layer i as the activation error is zero for the ReLU and linear functions.

Let us consider our running example in Figure 6.1. As there is no initial error, for the first layer, we simply add the roundoff error committed by the dot operation. For the bias, we add a roundoff error for both the dot and the bias operation. The range constraints for the first layer are thus: $[-4.0, 4.0] \geq [-3.0, 3.0] + \epsilon_1^{dot}$ and $[-8.0, 8.0] \geq [-3.0, 6.0] + \epsilon_1^{dot} + \epsilon_1^{bias}$. Similarly, we generate range constraints for all the layers. Given the user inputs, our prototype tool, Aster, automatically encodes the objective cost function, the error, and the range constraints for the optimization problem.

Linearization of Constraints However, we still have nonlinearity in the objective cost function, which includes the nonlinear function `max`. The error constraints are also nonlinear in terms of the number of unknown fractional bits (π).

We linearize these constraints exactly without introducing any approximation. Figure 6.3 presents our linearization constraints for the optimization. Linearization of the cost objective is straightforward. The `max` function of two variables π_i^{dot} and π_i^{bias} (as shown in Figure 6.2) in the form $\gamma_i^{bias} = \max(\pi_i^{dot}, \pi_i^{bias})$ can always be divided into a set of two linear constraints such that γ_i^{bias} is always bigger than or equal to all values of both π_i^{dot} and π_i^{bias}. We utilize this argument to linearize our `max` function as $C_{0(1-2)}$ in Figure 6.3.

$$
\begin{aligned}
&C_{0(1)} : \gamma_i^{bias} \geq \pi_i^{dot} \\
&C_{0(2)} : \gamma_i^{bias} \geq \pi_i^{bias} \\
&C_{1(1)} : opt_i^{dot} = \sum_{j=1}^{m} (T_j \times b_j^{dot}) \\
&C_{1(2)} : opt_i^{bias} = \sum_{j=1}^{m} (T_j \times b_j^{bias}) \\
&C_{1(3)} : \sum_{j=1}^{m} b_j^{dot} = 1 \\
&C_{1(4)} : \sum_{j=1}^{m} b_j^{bias} = 1 \\
&C_{1(5)} : \epsilon_i^{new} \leq opt_i^{dot} + opt_i^{bias}
\end{aligned}
$$

Figure 6.3: The linearization constraints

Linearizing the error constraint is, however, more tricky. To linearize the error constraint, we use *indicator constraints* [29] that uses binary variables to control the values of π_i^{dot} and π_i^{bias}. The nonlinear error constraint is defined in Figure 6.2 as $\epsilon_i^{new} \leq 2^{-\pi_i^{dot}} + 2^{-\pi_i^{bias}}$ where π_i^{dot} and π_i^{bias} are unknown integers in an user-provided integer range $[lo, hi]$. Recall that we want to compute the values of π_i^{dot} and π_i^{bias} that minimize our objective function.

As we already know the integer ranges of the variables, the possible values for π_i^{dot} and π_i^{bias} become finite (m) as they are exactly the same as the number of integers in the given range. With these m values, we define a set of m discrete reals that represent the set of possible values of $2^{-\pi_i^{dot}}$ and $2^{-\pi_i^{bias}}$: $T = [2^{-hi}, 2^{-(hi-1)}, \cdots, 2^{-lo}]$.

We introduce m binary indicator variables b^{dot} and b^{bias} for each valuation of $2^{-\pi_i^{dot}}$ and $2^{-\pi_i^{bias}}$ within the specified range. Intuitively, the indicator variables select only one specific value from the list T. We formulate two indicator constraints $C_{1(1-2)}$ for each layer i as presented in Figure 6.3.

The constraints $C_{1(1-2)}$ select the values of π_i^{dot} and π_i^{bias} that are optimal (opt_i^{dot} and opt_i^{bias}) respectively. They also state that if b_j^{dot} (or b_j^{bias}) is true, we select the value $2^{-(hi-j-1)}$ from the list T for opt_i^{dot} (or opt_i^{bias}). Obviously, we want only one of the b_j^{dot}(or b_j^{bias}) to be true. Hence, we add

another two constraints $C_{1(3-4)}$ in Figure 6.3 to enforce that only one of these binary indicator variables is true. With these new indicator variables and constraints, finally, we linearize the original nonlinear error constraint ϵ_i^{new} in Figure 6.2 as $C_{1(5)}$ in Figure 6.3. Our tool, Aster, encodes these linearization constraints for all layers fully automatically.

Let us assume for our running example in Figure 6.1 that the range is provided as $[lo, hi] = [4, 8]$ which makes the possible values for π_i^{dot} and π_i^{bias}: $T = [2^{-8}, 2^{-7}, 2^{-6}, 2^{-5}, 2^{-4}]$. Next, we define 5 binary indicator variables for b^{dot} and b^{bias} each. The indicator constraints for the dot product in the 1st layer are as follows: $opt_1^{dot} = \sum_{j=4}^{8} 2^{-j} \times b_j^{dot}$ and $\sum_{j=4}^{8} b_j^{dot} = 1$. Similarly, we have indicator constraints for bias. If the solver picks $b_7^{dot} = 1$ and $b_8^{bias} = 1$, the corresponding new error is then bounded by $2^{-7} + 2^{-8}$, and the optimized fractional bit lengths of dot and bias are 7 and 8, respectively.

6.2.3 Step 3: Correctly Rounded Precision Assignment

After solving the MILP, we obtain the fractional bits required for the dot operation and the addition of bias, and we know that the integer bits from the first phase are enough to prevent overflow even in the presence of errors. However, the fractional bits computed for the dot product only apply to the *result* of the dot operation. To generate a complete executable fixed-point implementation, we must also compute the precision of the *intermediate* operations (sum of products of the dot) and the constants of weights. In particular, we need to determine their fractional bits such that the results are rounded correctly up to the precision determined by the MILP.

Our algorithm to compute the intermediate and constant word lengths is based on the fixed-point sum of products by constants (SOPC) algorithm [61, 28]. We first compute the fractional bits for the intermediate computation of a dot. Assume x to be a vector of p fixed-point variables in formats (I_{x_i}, π_{x_i}) and c be a vector of p fixed-point constants in formats (I_{c_i}, π_{c_i}) where I denotes the integer bits. Our goal is to compute $y = \sum_{i=1}^{p} c_i \cdot x_i$ correctly.

The integer bits of the output I_y are already computed in the first step, such that no overflow occurs. The fractional bits of the output π_y are determined by MILP in the second step. These two combined represent the output precision and an accuracy requirement, which ensures that the roundoff error is bounded by $2^{-\pi_y}$.

As it was shown in [61], if the integer bits of output I_y guarantee no overflow, and partial products $s_i = c_i \cdot x_i$ are performed exactly, then performing the summation in an extended format (I_y, π_{ext}) guarantees the output error bound. The number of extended fractional bits depends on the number of terms that need to be added: $\pi_{ext} = \pi_y + \lceil \log_2 p \rceil + 1$. Note that the integer bit positions for the intermediate results are not changed, which might lead to overflows in partial sums during the computation of the dot product. However, because of the properties of 2's complement, these overflows do not influence the result accuracy as long as the output is representable with the output integer bit I_y [28].

Next, we must obtain the fractional bits of the weight constants ($w_i \in W$) such that the

error bound holds. As we have mentioned before, the intermediate results of the partial sums need to be done with π_{ext} fractional bits to ensure correct rounding. Now, in order to ensure that the accuracy of the product $w_i \times x_i$ is up to π_{ext}, the following property needs to hold: $\pi_{w_i} + I_{x_i} \leq \pi_{\text{ext}}$, which implies that the π_{w_i} needs to be at least $\pi_{\text{ext}} + I_{x_i}$.

Finally, the result of the dot product is added to the bias vector. From the MILP, we obtained the fractional bit of the result of the addition π^{bias}. We need to compute the fractional bits of the bias constants ($b_i \in \overline{b}$). To ensure that the roundoff error at the result is bounded by $2^{-\pi^{bias}}$, the fractional bits of b_i is set to be π^{bias}. As the formats of the operands might differ, we set the format of the bias upfront to ensure π^{bias} fractional bits in the result.

6.2.4 Soundness

THEOREM. Given a set of input ranges $\mathcal{R}$ and a specified error bound ϵ_{target} for a NN, if the MILP-based mixed-precision optimization terminates successfully, it returns a fixed-point precision assignment for the network such that for all inputs $i \in \mathcal{R}$, the maximum roundoff error ϵ_n of the network (relative to a real-valued implementation) does not exceed ϵ_{target}, i.e., $\epsilon_n \leq \epsilon_{target}$.

Proof Sketch. Our MILP-based mixed-precision tuning procedure guarantees soundness by construction. We summarize the correctness argument for each of the three steps.

The first step employs sound interval arithmetic [139] to compute the integer bits of all program variables and constants, including intermediate ones. This computation proves the absence of overflow, ensuring that the resulting integer bits are valid and consistent.

The second step assigns the fractional bits of the dot and bias results as the solution of an MILP optimization problem, which bounds the overall roundoff error to be below the user-given error bound. The MILP error constraints soundly over-approximate the true roundoff errors (by assuming worst-case errors and error propagation at each step), and the range constraints ensure that no overflow is introduced due to roundoff errors. The linearization constraints are exact and maintain the soundness of the MILP encoding.

The third and final step assigns precisions to the intermediate variables that store the sum of products of the dot and the weight constants. This is achieved by utilizing the previously determined sound integer bits from the first step and following the fixed-point sum of products by constants (SOPC) algorithm. The correctness of the SOPC algorithm directly follows from [28].

Together, these steps assign both the integer and the fractional bits of all variables, constants, and operations such that no overflow occurs and the overall error bound is satisfied. □

6.3 Implementation

Our tool, Aster, takes as input the network architecture and specification written in a small domain-specific language. The input corresponding to our running example from Figure 6.1 is

```
def runningExample(x:Vector): Vector = {
  require(lowerBounds(x, List(-10, -5)) && upperBounds(x, List(10, 5)))
  val weights1 = Matrix(List(List(0.1, 0.2), List(0.2, 0.15)))
  val weights2 = Matrix(List(List(0.1, 0.2)))
  val bias1 = Vector(List(1.0, 2.0))
  val bias2 = Vector(List(0.5))
  val layer1 = relu(weights1 * x + bias1)
  val layer2 = linear(weights2 * layer1 + bias2)
  layer2
} ensuring(res => res +/- 0.1)
```

Figure 6.4: Network architecture in Aster's input format

shown in Figure 6.4. The 'require' clause specifies the input ranges, and the 'ensuring' clause specifies the overall error bound. In addition, Aster takes the maximum fractional bit length of the input vector, the word length of maximum weights, and the range of the fractional bits for the optimization as command-line user inputs. As explained in Section 6.2.2, we need these values to make the optimization problem linear.

Note that we have implemented Aster to handle feed-forward NNs with `relu` and `linear` activations fully automatically; however, Aster can straightforwardly be extended to handle NNs with sparse matrix multiplication and piece-wise linear activation functions with bounded domains.

Aster generates fully quantized code, including (more) accurate precisions for the weights written in C using the `ap_fixed` library. This code can directly be compiled for an FPGA with the state-of-the-art HLS compiler by Xilinx [171]. Aster's generated code either expresses the matrix operations as for-loops or, alternatively, it can fully unroll these loops.

Integer Bit Computation We observed that a straightforward interval analysis computation of the real-valued ranges quickly becomes expensive with the increasing complexity of the network. We thus leverage the structure of the NN for a more efficient, though over-approximate range analysis. Specifically, we abstract the input variables to each layer by a single range that soundly covers all individual variables and do so similarly for the weights and biases at each layer. In doing so, we can compute the output range of a layer by computing a single dot product with one addition. This over-approximation makes the interval analysis scalable even for large networks with thousands of parameters, while we have not observed it to significantly affect the optimization results.

Choice of MILP Solver For our fixed-point precision optimization, the values we encounter can be very small, e.g., for fixed 32-bit precision, the roundoff errors are on the order of $1e-9$. State-of-the-art MILP solvers use finite precision internally as well, and it is thus crucial to choose a solver that is precise enough to be able to distinguish its own internal roundoff errors from the values in our constraints. In our work, we integrate Aster with the SCIP optimization Suite [79] with the underlying SoPlex solver to solve our mixed-integer linear programming problem. SCIP

internally uses extended precision, which goes beyond the limits of floating-point arithmetic and thus allows us to deal with values as small as $1e-15$. We found that other widely-used industrial solvers (such as CPLEX [101] and Gurobi [94]) have tolerances that are bounded by $1e-6$. These are, unfortunately, too coarse for our fixed-point precision optimization. Note that SCIP's precision is not unlimited, either. If the roundoff error goes beyond the tolerance limits (that is $1e-15$), Aster cannot optimize and will report an error (i.e., it will *not* return an incorrect result).

6.4 Experimental Evaluation

In this section, we conduct an extensive evaluation of Aster across three research aspects:

a) We investigate in Section 6.4.1 how different input parameters affect Aster's results and performance.

b) We compare Aster with the state-of-the-art in terms of implementation cost of NNs in Section 6.4.2.

c) We compare the optimization time of Aster with the state-of-the-art techniques in Section 6.4.3.

Benchmarks We have collected a set of 18 NN controllers, consisting of 15 models from the competition at the Applied Verification for Continuous and Hybrid Systems (ARCH) workshop from the years 2019 [131] and 2020 [111], where the networks were provided by academia as well as industry. Additionally, we included 3 controller models from the VNN-LIB standard benchmark set [90], which is widely used for the verification of NNs. Verification of these controllers is becoming increasingly important due to their usage in safety and operational critical systems. We took all benchmarks for which we could extract all values of weights and biases from the repository. Table 5.1 provides details of the network architectures. We present a brief description of the benchmarks here; a more detailed discussion can be found in [66, 111, 90].

Pendulum Controllers The `InvPendulum`, `SinglePend`, `DoublePend` benchmarks are NN controllers for inverted pendulums with one, one, and two links respectively. We use two versions of the double pendulum, `V2` is more robust than the `V1`. The controller `Acrobot` (originally from [106] where it was proposed as a benchmark for reinforcement learning) is also a two-linked pendulum but with only the second joint actuated.

Car Controllers The `Unicycle` and `MountainCar` benchmarks model a car, where the first is a stabilizing controller that controls a simple unicycle dynamics, and the latter models the dynamics of a car placed between two mountains where the objective is to climb up a mountain on the right. They are originally taken from [68] and [106] respectively. We also have 4 different versions of `ACC`, which models an adaptive cruise control system with 3, 5, 7, and 10 hidden layers that automatically adjust and control the speed of a car considering other cars.

benchmarks	# in	# params	architecture
InvPendulum	4	60	$4 \times 10 \times 1$
MountainCar	2	336	$6 \times 15 \times 15 \times 2$
Acrobot	6	375	$6 \times 20 \times 20 \times 8$
MPC	6	720	$6 \times 20 \times 20 \times 8$
SinglePend	2	775	$2 \times 25 \times 25 \times 2$
DoublePendV1	4	825	$4 \times 25 \times 25 \times 2$
DoublePendV2	4	825	$4 \times 25 \times 25 \times 2$
ACC3	5	980	$5 \times 20 \times 20 \times 20 \times 1$
ACC5	5	1,820	$5 \times 20 \times 20 \times 20 \times 20 \times 20 \times 1$
VCAS	3	1,940	$3 \times 20 \times 20 \times 20 \times 20 \times 20 \times 9$
ACC7	5	2,660	$5 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 1$
AC6	12	3,457	$12 \times 64 \times 32 \times 16 \times 1$
Unicycle	4	3,500	$4\times 500 \times 2$
ACC10	5	3,920	$5 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 1$
AC7	12	12,033	$12 \times 128 \times 64 \times 32 \times 1$
Airplane	12	13,540	$12 \times 100 \times 100 \times 20 \times 6$
ControllerTora	4	20,800	$4 \times 100 \times 100 \times 100 \times 1$
AC8	12	44,545	$12 \times 256 \times 128 \times 64 \times 1$

Table 6.2: Details of benchmark architectures, listing the number of inputs and parameters (considering both weight and bias parameters) as well as the neurons in each layer

Quadrotor controller The MPC benchmark models a six-dimensional control-affine quadrotor and was also introduced in [106]. We use a modified version of the controller with only ReLU and linear activation functions taken from [66].

Rotational Actuator Controller The controllerTora benchmark is an NN controller that controls translational oscillations by a rotational actuator first utilized in [67].

Airplane Controllers The Airplane models a simple dynamical system of a flying airplane. We also use a network, the VCAS, that models the closed-loop aircraft collision avoidance system ACAS X. The network is originally from [112].

Drone Controllers The AC benchmarks are derived from a baseline case study involving the problem of Drone control [90] using reinforcement learning. The controllers regulate a single RPM value across all motors to facilitate takeoff and maintain a fixed hovering position. We have extracted the three largest networks (6, 7, 8) in terms of number of parameters.

We have extracted the safe input ranges, the weight matrices, and bias vectors of these controllers from the competition's repository [66, 9], as well as the VNN-LIB repository [90]. The provided MATLAB files, hierarchical data format (HDF) files, and open format for ML models

(ONNX) files were converted into the input format of Aster. The networks do not come with specified target error bounds, so we choose two target absolute error bounds uniformly for all networks, $1e-3$ and $1e-5$, that we believe to be in a reasonable range for embedded controllers, which are typically implemented in lower precision (16 bits or 32 bits).

Experimental Setup All experiments are done on an Intel Core i5 Debian system with 3.3 GHz and 16 GB RAM. Aster uses SCIP Optimization Suite 7.0.3 as the external MILP solver. We set $1e-15$ as the zero and feasibility tolerance limits for the SCIP as these tolerances are precise and efficient enough for our purpose. We further assume 32 as the initial guess for the word length of the maximum weights to start the optimization. We choose to compare Aster with the state-of-the-art precision tuner Daisy [54], as this is the only available *sound* mixed-precision tuner for fixed-point programs. For the comparison, we use Daisy's version downloaded on 2^{nd} March 2021 (there have been no major commits since). We set a 5-hour time budget for each optimization run in our experiments. We believe that a total of 5 hours is a reasonable time for an analysis to generate a sound implementation that can directly be synthesized on FPGAs. For the synthesis of FPGA designs, we use Xilinx's Vivado HLS tool [171] (version v2020.1) downloaded on 27^{th} May 2020, and set a timeout of 5 hours for our experiments.

6.4.1 Evalutation of Parameter Settings

Aster has several input parameters that are needed to make the MILP optimization tractable, specifically the range of the fractional bits π and the input fractional bit lengths π_0, which determines the input error. These input parameters affect Aster's results and optimization time.

We define 4 different settings that we denote with letters 'A-D' as shown in Table 6.3. Setting A considers a smaller value of π_0, resulting in a larger input error and a wider range of π compared to setting B. Setting C and D do not consider initial errors but have different ranges for π. These two settings are useful when the user knows that the inputs are represented exactly and is only interested in evaluating the roundoff error during internal computations.

settings	π_0	initial error	π range
A	20	2^{-20}	[5, 32]
B	32	2^{-32}	[10, 32]
C	10	0	[5, 32]
D	17	0	[10, 32]

Table 6.3: Aster's settings

Choosing a wide range of π results in more variables and constraints, thus making it harder for the underlying SCIP optimizer to generate results. π_0 needs to be large enough to admit a valid solution to the optimization problem, but too large π_0 will, in general, result in a—potentially unnecessarily—higher overall cost. Note that since Aster computes an over-approximation of the error, the solver may report infeasibility even though a solution to the not-approximated problem may exist.

We determine suitable parameters for our benchmarks with a systematic empirical exploration using settings 'A-D' shown on the left in Table 6.4, considering the two error bounds $1e-3$ and

benchmarks	target error = $1e-3$			target error = $1e-5$		
	A	B	C	A	B	D
InvPendulum	**20404**	**20404**	**20404**	36886	34966	**34006**
MountainCar	**151160**	154934	163486	inf	**207970**	**207970**
Acrobot	175601	**151103**	157546	274887	**248595**	**248595**
MPC	**250952**	**250952**	**250952**	inf	**250952**	**250952**
SinglePend	271918	362100	**263547**	**436565**	438992	438992
DoublePendV1	**352720**	420640	408280	581144	**540042**	**540042**
DoublePendV2	442416	**436810**	453224	inf	508924	**505722**
ACC3	444638	**442438**	**442438**	inf	**446838**	**446838**
ACC5	inf	**702228**	**702228**	inf	**702228**	**702228**
VCAS	inf	inf	inf	inf	inf	inf
ACC7	inf	inf	inf	inf	inf	inf
AC6	2069122	**1986944**	2009218	inf	**2069122**	**2069122**
Unicycle	**1134062**	**1134062**	**1134062**	1736074	**1636072**	**1636072**
ACC10	inf	inf	inf	inf	inf	inf
AC7	inf	**6660220**	**6660220**	inf	**6660220**	**6660220**
Airplane	inf	**5967462**	**5967462**	inf	inf	inf
ControllerTora	**10562548**	11605748	11632956	**13532966**	**13532966**	**13532966**
AC8	inf	21338216	**21261416**	inf	**21338216**	**21338216**

Table 6.4: Parameter evaluation of Aster's with settings 'A-D', inf denotes infeasibility

$1e-5$. Among settings A and B, setting B is expected to be made more benchmarks feasible due to a smaller initial roundoff error, although it may potentially result in a higher cost. Settings C and D set the input error to zero and thus may result in a lower cost.

Table 6.4 shows the overall cost of the precision assignment determined by Aster for these 4 different settings. Setting A is, in general, cost-effective with a larger error bound ($1e-3$), but results in infeasible error bounds for the larger networks. The reason for this is that the bigger initial error gets magnified along with computing new errors at each layer, thus making it impossible to achieve the target error bound. As expected, more benchmarks become feasible with setting B. However, the tradeoff is that setting B mostly computes a larger cost as it considers the largest initial fractional bit length.

Settings C and D are expected to compute the lowest cost, as the input variables in these settings incur no error, along with smaller initial fractional bit lengths. This is indeed the case for 9 and 13 of the benchmarks out of 18, with error bounds of $1e-3$ and $1e-5$, respectively. For the rest of the benchmarks, settings C and D compute higher costs than settings A and B, though the costs are mostly close to those of settings A or B. We have observed that for these

benchmarks, the optimizer finds some specific assignment configurations that work optimally, given the optimization problem. These assignments cost more in the end when we compute the final costs of the whole program after assigning precision to all intermediate program variables and constants. If the user already knows how many bits are required to represent the inputs exactly, setting D is better, as Aster can take that into account.

The benchmarks `VCAS`, `ACC7`, `ACC10` are infeasible with all the settings. Upon closer inspection, we found that this is due to intermediate ranges becoming very large (on the order of $1e+9$). For example, if we reduce the range of one of the input variables of the `VCAS` benchmark slightly (from [-133, -129] to [-130, -129]), Aster's setting B *is* able to generate a precision assignment in 16.45 seconds for the target error bound of $1e-3$. That said since Aster cannot generate a precision assignment for the original input ranges, we remove these benchmarks from the later experiments.

We explore the generic settings A-D for our evaluation, but we expect that when using Aster on a specific application, the user may either know suitable parameter values up front or may run Aster several times to explore different options. The latter is feasible due to Aster's small optimization times (see Section 6.4.3).

We use settings A and B with error bounds $1e-3$ and $1e-5$, respectively, for our comparison with the state-of-the-art in Section 6.4.2 except for `ACC5`, `AC7`, `Airplane`, and `AC8`. For these four benchmarks, we will use setting B for *both* error bounds, as only with setting B could we generate implementations. We do not include settings C and D as the state-of-the-art tuner does not support analysis without initial errors.

6.4.2 Comparison with State-of-the-Art in terms of Implementation Costs

We compare Aster with the state-of-the-art precision tuner Daisy [54] that focuses explicitly on optimizing the precision to satisfy given roundoff error bounds and generates mixed fixed-precision implementations of arithmetic programs that are *sound* considering all possible fixed-point precisions (not only, say, 4 or 8 bits). Note that dynamic quantization tools like SeeDot [87] and Shiftry [120] are not sound and are designed specifically for NN classifiers and do not handle feed-forward NN controllers that solve regression problems that we focus on in this work.

Daisy only works on generic straight-line code without loops and, in particular, does not handle programs with data structures like matrices and vectors that are standard in NNs. To use Daisy on these programs, we completely unroll the loops and data structures, i.e., manually assign individual matrix and vector elements to individual scalar variables.

We use Daisy's following two modes for our purpose:

- **Uniform:** In this mode, Daisy computes the total error bound for a given fixed precision. We determined the lowest uniform precision that satisfies the error bound by manually employing Daisy repeatedly in this mode and checking if the computed error satisfies the

given bound.

- **Mixed-precision:** This mode generates a mixed fixed-point precision assignment using a heuristic search based on delta-debugging [54] that repeatedly evaluates the roundoff errors for different precision assignments, starting from the lowest uniform precision that satisfies the given error bound. However, the computational cost of this heuristic search increases rapidly as the size of the NN grows.

Daisy generates the tuned fixed-point code as a C program in the same format as Aster that can be directly compiled to an FPGA by the Xilinx compiler [171]. Just like the input, the output code of Daisy is fully unrolled straight-line code and potentially very large.

We compare Aster's setting A (fractional bits in the range $[5, 32]$ and the input fractional bit length = 20), setting B (fractional bits in the range $[10, 32]$ and the input fractional bit length = 32), and Daisy with uniform and mixed-precision tuning with a maximum bit length of 32 bits on all our benchmarks, considering the $1e-3$ and $1e-5$ error bounds. For uniform precision, we use the lowest *uniform* precision that satisfies the error bounds. Note that we do not consider settings C and D, as in those settings, we assume no initial error, which Daisy does not support.

Both Aster and Daisy share the common objective of minimizing the total number of bits utilized by the NN. However, they employ entirely different techniques to achieve this goal. Daisy employs a *heuristic* search method that involves multiple iterations of error analysis—one for each candidate precision assignment. In contrast, Aster adopts a global optimization-based approach: it creates a single optimization constraint and solves it for the precision assignment. Although both tools employ cost functions for mixed precision tuning, these functions are similar but not identical.

We cannot use Daisy's cost function directly in Aster as it would lead to non-linear constraints. Specifically, Aster uses assumptions (see Section 6.2.2) regarding the bit lengths of the NN structures (matrices, vectors) and the inputs to maintain linearity. These assumptions make sense when optimizing NNs, for example, to keep the data structures intact in the final implementation. It is also not immediately possible to adapt Daisy to use Aster's cost function. Daisy considers the unrolled code as-is, optimizes each variable individually, and assigns precisions to all variables in a program, including the inputs. Hence, Daisy only has the unrolled code available while optimizing but would need the higher-level data structure information for Aster's cost function. Hence, a fair comparison of Daisy and Aster is impossible as their targeted and possible optimizations differ.

Latency of FPGA Implementations Ultimately, what matters is the actual performance of the generated mixed-precision code. Since Daisy has been used to optimize the latency of fixed-point implementations on FPGAs [109], we compare the tools on that measure. We compile the code generated by Aster and Daisy for a (standard) FPGA architecture using the Xilinx' Vivado HLS tool and compare the running time in terms of machine cycles, i.e., latency, that the compiler reports for the final hardware implementation. (We cannot customize our cost function further for the HLS compiler, as it is commercial and its internal implementation choices are unknown.)

benchmarks	target error: $1e-3$			target error: $1e-5$		
	Daisy		Aster (setting A)	Daisy		Aster (setting B)
	uniform	mixed		uniform	mixed	
InvPendulum	**12**	**12**	**12**	14	14	**13**
MountainCar	27	27	**25**	28	29	**25**
Acrobot	**24**	**24**	25	**24**	25	26
MPC	inf	inf	**35**	inf	inf	**37**
SinglePend	**22**	23	25	**24**	**24**	27
DoublePendV1	29	**28**	**28**	31	31	**28**
DoublePendV2	36	36	**27**	36	36	**30**
ACC3	49	49	**44**	inf	inf	**46**
ACC5	inf	inf	**72**$^{\#}$	inf	inf	**74**
AC6	62	62	**48**	inf	inf	**49**
Unicycle	178	178	**27**	inf	inf	**28**
AC7	TO	TO	**8310**$^{*\#}$	inf	inf	**8310**$^{*\#}$
Airplane	TO	TO	**9001**$^{*\#}$	TO	TO	inf
ControllerTora	TO	TO	**13158***	TO	TO	**13558***
AC8	TO	TO	$\times^{*\#}$	TO	TO	$\times^{*\#}$

Table 6.5: Latencies of implementations generated by Daisy and Aster considering errors $1e-3$ and $1e-5$ respectively (TO: timed out after 5 hours, inf: tools returned infeasible, #: used Aster's setting B, *: compiled with explicit loops (i.e. not unrolled code)), $\times$: Xilinx failed to compile the implementation

Note that this reported latency is exact, and we thus do not measure the noisy runtime on actual hardware.

Table 6.5 shows the latencies of the generated code for our benchmarks considering the target errors $1e-3$ and $1e-5$. It shows that Aster generates feasible implementations for significantly more benchmarks and for benchmarks where both tools successfully generate compilable code, Aster mostly produces code with lower latency than Daisy. Considering the latter benchmarks, we see that Aster matches the performance of Daisy's generated code for two benchmarks and improves on it for 5 benchmarks. Only for two benchmarks, Aster's code's latencies are (slightly) larger. Our results thus show that Aster's optimization is often able to produce faster implementations for NN controllers than the state-of-the-art *heuristic* mixed-precision tuner of Daisy.

Compared with Daisy's uniform precision assignments, we furthermore confirm that mixed precision is indeed overall beneficial for platforms such as FPGAs. This is particularly striking for the Unicycle benchmark, where Aster's reduction in latency is almost 84% considering the $1e-3$ error as Aster is able to optimize some variables much more significantly than Daisy.

The latency, considering the smaller error bound, is only slightly larger, whereas Daisy reports infeasibility.

Our results also show that Aster is more scalable than Daisy, generating a precision assignment for all benchmarks for the larger error bound (Daisy failing on 6), and reporting an infeasible result for one benchmark with the smaller error bound (Daisy failing on 9). Owing to our optimization problem formulation, the number of variables in the MILP grows linearly with the number of layers: one decision variable for the dot product computation and the other for adding bias. This formulation makes it possible for Aster to find a solution even for large networks. In contrast, Daisy's heuristic search becomes intractable with the increasing complexity of the network, leading to many variables and constants. In addition, Daisy performs the error analysis multiple times to ensure that the precision assignment meets the error requirement, thus timing out often.

We run the Xilinx Vivado compiler on the fully unrolled code as we have observed that it leads to smaller latencies (and Daisy generates fully unrolled code). The exceptions are the `AC7`, `Airplane`, `ControllerTora`, and `AC8` benchmarks (marked with a star in Table 6.5), where the generated C code is too large to be compiled within the 5h timeout ($\sim$ 36K, 40K, 62K, and 134K lines of code). Aster can generate smaller programs (54, 58, 54, 54 lines of code) that preserve the original NN structures and loops, and we use these implementations for compilation with Xilinx. The benchmark `AC8`, however, is still too large in terms of the number of loop iterations to be compiled by Xilinx within 5 hours without further customization. Nonetheless, in general, our results show that considering the high-level structure of NNs is beneficial, especially for larger networks.

Comparison of Cost Functions We also compare Daisy's and Aster's results using both cost functions: we generate unrolled code by Aster to use Daisy's cost function, and we use Aster's cost function to evaluate code generated by Daisy. Table 6.6 shows the cost reduction in % achieved by Aster with respect to Daisy's analyses. A negative result signifies that the cost of Aster's implementation is higher than Daisy's.

Not surprisingly, our results show that Aster outperforms Daisy's analyses considering Aster's cost function excluding both `MountainCar` and `Acrobot` with error bound of $1e-3$ and only `Acrobot` with error bound of $1e-5$. However, with Daisy's cost function, Daisy's analyses, both uniform and mixed, outperform Aster, where the cost is computed on the unrolled program, with the exception of the `Unicycle` benchmark with $1e-3$ error bound and the `SinglePend` with uniform precision analysis.

Aster's cost function on Daisy's implementation considers the largest bit lengths, one for weights and the other for bias in a layer, to compute a sound cost. This defeats Daisy's advantage of optimizing all variables, thus resulting in higher costs than Aster. Likewise, using Daisy's cost function for Aster's implementation is also suboptimal for Aster as it considers the same bit length in a layer for linearity and to keep NN structures intact, thus improving scalability. We thus conclude that Daisy and Aster are each better for their intended use cases. Upon closer inspection of Aster's and Daisy's generated precisions, we further observed that Aster's

benchmarks	target error: $1e-3$, setting: A				target error: $1e-5$, setting: B			
	Daisy's cost		Aster's cost		Daisy's cost		Aster's cost	
	u (%)	m (%)	u (%)	m (%)	u (%)	m (%)	u (%)	m (%)
InvPendulum	-6.44	-13.97	**15.99**	**17.80**	-23.52	-37.31	**17.08**	**13.45**
MountainCar	-43.14	-52.47	-2.64	-0.02	-19.68	-31.47	**14.18**	**17.84**
Acrobot	-25.70	-37.30	-21.84	-34.12	-6.10	-19.93	-8.21	-2.12
MPC	*	*	*	*	*	*	*	*
SinglePend	**1.89**	-11.12	**11.95**	**2.58**	**0.96**	-2.57	**10.81**	**10.69**
DoublePendV1	-16.36	-24.33	**6.00**	**3.61**	-14.12	-16.11	**5.01**	**10.01**
DoublePendV2	-32.61	-40.95	**0.92**	**6.22**	-40.71	-47.05	**24.56**	**24.38**
ACC3	-62.01	-65.50	**38.37**	**19.35**	*	*	*	*
ACC5	inf	inf	inf	inf	*	*	*	*
AC6	-66.95	-64.96	**18.13**	**17.45**	*	*	*	*
Unicycle	**24.10**	**9.50**	**48.15**	**39.20**	*	*	*	*
AC7	×	×	×	×	*	*	*	*
Airplane	inf	inf	inf	inf	inf	inf	inf	inf
ControllerTora	×	×	×	×	×	×	×	×
AC8	×	×	×	×	×	×	×	×

Table 6.6: Reduction (%) in cost using Aster's settings A and B w.r.t. Daisy's uniform (u) and mixed (m) analyses considering $1e-3$ and $1e-5$ error bounds using Daisy's and Aster's cost functions (×: Daisy times out after 5 hours, *: Daisy returns infeasible but Aster generates an implementation, inf: Aster returns infeasible)

assignment of precisions in the dot product is sometimes overly and unnecessarily conservative and can be improved in the future. That said, as we have seen in Table 6.5, unrolled code can become impractical for hardware implementations of large NNs. In these cases, Aster's optimization and cost function are better suited, with which Aster performs significantly better than Daisy.

6.4.3 Comparison with State-of-the-Art in terms of Running Time

We compare the optimization time of Daisy's uniform and mixed-precision tuning with Aster's setting A for the target error of $1e-3$ and Aster's setting B for the target error of $1e-5$. For Daisy's uniform precision, we only show the time to run the roundoff error analysis once *after* we find the lowest uniform precision satisfying the error bound using Daisy's mixed precision analysis (we could not find a better way to generate this information). Also, Aster generates infeasibility with setting A for the ACC5, AC7, Airplane, and AC8 benchmarks with target error

benchmarks	error: $1e-3$			error: $1e-5$		
	Daisy (unif)	Daisy (mix)	Aster (A)	Daisy (unif)	Daisy (mix)	Aster (B)
InvPendulum	1.72	4.19	**1.66**	1.70	5.42	**1.62**
MountainCar	5.16	43.68	**2.22**	5.11	38.79	**2.15**
Acrobot	6.25	196.93	**2.31**	6.04	95.41	**2.24**
MPC	-	-	**3.50**	-	-	**3.44**
SinglePend	17.63	237.83	**3.52**	17.23	115.25	**3.53**
DoublePendV1	19.42	127.96	**3.69**	19.69	201.13	**3.84**
DoublePendV2	20.55	246.64	**3.80**	20.66	334.03	**3.72**
ACC3	27.13	292.05	**7.28**	-	-	**4.84**
ACC5	-	-	**12.97**$^{\#}$	-	-	**12.85**
AC6	311.37	2033.23	**51.01**	-	-	**51.04**
Unicycle	354.71	9980.65	**49.92**	-	-	**45.78**
AC7	3999.54*	TO	**727.40**$^{\#}$	-	-	**729.05**
Airplane	**906.93***	TO	1060.92$^{\#}$	TO	TO	-
ControllerTora	12128.83*	TO	**2875.95**	TO	TO	**2521.21**
AC8	TO	TO	**13771.43**$^{\#}$	TO	TO	**13794.66**

Table 6.7: Optimization time (in seconds) averaged over 3 runs; TO: timed out after 5 hours, *: used 32 bit uniform precision analysis, #: used Aster's setting B instead of A, -: returns infeasible)

$1e-3$, we show the running time with setting B. Table 6.7 shows the end-to-end optimization time measured by the bash time command of Daisy and Aster in seconds averaged over 3 runs.

In general, running uniform precision roundoff analysis once is quick for smaller benchmarks, but with the increased size of the network, it also becomes slower and even times out after 5 hours for the largest benchmark AC8 with both target error bounds. The running times are substantial even though Daisy assigns a uniform given bit length because it still needs to run the error analysis to ensure overflow freedom and appropriately assign integer bits. Daisy's mixed-precision tuning is usually a magnitude slower than the uniform precision analysis.

Aster outperforms Daisy's analyses substantially in terms of running time for both error bounds for almost *all* benchmarks with the exception of Airplane, where running Daisy's uniform analysis once is the fastest.

With setting B, Aster uses a smaller range of fractional bits ($[10, 32]$). Hence, the number of constraints in this setting is less than in setting A, resulting in a smaller running time in general.

Due to our formalization of the optimization problem, the number of constraints and variables only depends linearly on the number of layers and the range of fractional bits (π) for Aster's settings. Solving an MILP problem with a limited number of variables is very efficient. The Airplane, ControllerTora, and AC8 are the largest benchmarks with the most number of variables

and constraints. For these MILP problems, where a solution is feasible (except `AC8` with setting A and `Airplane` in both settings), it was found in a maximum of 0.15 seconds. Our experiments thus confirm that constraint optimization is not a bottleneck of our mixed precision tuning and it is indeed a viable solution for fixed-point computation, even for large networks with many parameters.

6.5 Conclusion

This work introduces a novel sound quantization approach that assigns fixed-point mixed precision to NNs for regression tasks, guaranteeing a user-provided roundoff error bound. By formulating the precision optimization problem as an MILP problem and incorporating efficient over-approximations, we are able to solve it effectively and efficiently using a state-of-the-art solver. Our experiments demonstrate that our method is fast, even for large networks with thousands of parameters. This efficiency indicates that constraint optimization is not a bottleneck here. Our proposed technique is suitable for quantizing NNs that appear as verified embedded controllers. It is able to handle more and larger benchmarks than existing fixed-point tuners and mostly generates more efficient code for custom hardware such as FPGAs. While our focus has been on regression models for NN controllers, there is potential to extend Aster's applicability to quantizing classifiers with further exploration to bridge the gap between numerical and classification accuracy.

7 Related Work

We have reviewed essential techniques for analyzing and optimizing finite-precision programs and the tool Daisy [55] in the background section (chapter 2) that form the basis of our proposed approaches. In this section, we present a selection of other methods, some with similar objectives and others with differing goals compared to ours, which we consider complementary to our work.

7.1 Finite Precision Analysis

Analyzing finite-precision programs involves analyzing accuracy, handling exceptions, and identifying bugs. We categorize the finite-precision analyses into static and dynamic techniques and provide a concise overview of the current state-of-the-art techniques in both categories.

7.1.1 Static Analysis

Accuracy Analysis As outlined in chapter 2, there are two primary types of static analyses for computing the accuracy of finite precision programs: those based on data-flow analysis and those based on global optimization. These tools compute the roundoff error in the output and are capable of proving the absence of large roundoff errors (cancellations).

In our work, we have leveraged the data-flow based roundoff error analysis techniques implemented in Daisy [55]. This decision was motivated by several factors. Daisy is open-source, it supports both floating-point and fixed-point computations, and it incorporates error refinement features such as interval subdivision and the use of an SMT solver. Moreover, it offers the potential for seamless integration with state-of-the-art data-flow-based probabilistic analysis, as demonstrated in chapters 3 and 4. While not directly relevant to our work, it is worth noting that proof certificates for Daisy's error bounds can be generated and validated by tools like FloVer [20].

In the subsequent part, we present a concise overview of alternative tools to Daisy that employ data-flow analysis and tools that employ optimization-based techniques.

- *Other Data-Flow based Approaches* – The tools Gappa [60], Fluctuat [64], and Rosa [53] utilize data-flow based analysis for computing the worst-case roundoff errors, and in principle, could have been used for our purpose.

  Gappa employs interval arithmetic to reason about properties about ranges and errors

and provides a checkable proof from source code. Fluctuat utilizes zonotopes, a higher-dimensional affine representation (similar to Daisy's affine arithmetic), for both range and error analysis. To further refine error bounds, it employs techniques like range subdivision and Taylor expansion for unary non-linear operations. Rosa's analysis is similar to Daisy's, employing interval arithmetic for range computations and affine arithmetic for error estimations. Additionally, similar to Daisy, it supports interval subdivision and utilizes an SMT solver to systematically tighten error bounds.

These tools primarily target multivariate straight-line code, often using a specialized syntax. However, some extend their capabilities further. For instance, both Fluctuat and Rosa provide support for conditional branches by computing the rounding errors of each branch and the errors arising from diverging executions between the 'if' and 'else' branches. Additionally, Fluctuat can handle loops by employing techniques like loop unrolling, widening, or a combination of both. However, loop unrolling requires the number of loop iterations to be known beforehand, and widening often returns highly over-approximated error bounds. Rosa, on the other hand, offers a method to bound rounding errors in specific types of bounded loops, although it requires prior knowledge of invariants for ranges of variables.

A recent addition to this domain is DS2L [105], which builds on the foundation of Daisy. DS2L extends Daisy's capabilities to support complex list-like data structures, such as matrices and vectors, typically written in a functional programming style.

- *Global Optimization based Approaches* – In addition to data-flow based techniques, there exist optimization-based techniques. Tools like FPTaylor [161], real2Float [132], PRECiSA [140], and Satire [56] fall into this category. All these tools formulate roundoff errors for floating-point programs as maximization objectives and solve the problem using different techniques. They, too, aim to compute only the worst-case errors and mostly focus on simple straight-line floating-point programs.

  The tool FPTaylor introduces symbolic Taylor's expansions for the objective. It employs branch-and-bound techniques to maximize the objective and, finally, computes a sound bound on higher-order terms through the use of interval arithmetic. Similar to FPTaylor, Satire leverages symbolic Taylor terms for computing precise error bounds but adopts an incremental abstraction of numerical expressions, thus scaling better for large straight-line programs. The tool, Real2Float, also relies on symbolic Taylor expansion to create the maximization objective but employs a semi-definite programming optimization technique to solve the problem.

  Alternatively, PRECiSA employs a symbolic expression to represent the error, followed by the application of a branch-and-bound solver to precisely quantify the error. It also provides support for computing errors due to divergent executions between 'if' and 'else' branches, along with evaluating errors for individual branches.

None of these tools are tailored to effectively handle large non-functional programs in languages like C and C++, which involve complex data and program structures like arrays, structures, branching, loops, and function calls that we address in our work in chapter 5.

In contrast to our work presented in chapters 3 and 4, which incorporates the probability distribution of inputs to provide a more nuanced and realistic analysis of control flow and roundoff errors, all these tools focus on computing worst-case errors, and consequently, provide too pessimistic results to be useful in the scenarios we considered.

There exists a constraint-based approach [44], which heuristically generates test cases where floating-point and real-valued results deviate, especially when there are discrepancies in control flow [180]. This methodology complements our approach for computing wrong path probabilities. When our technique computes an exceptionally high probability, this method of generating test cases can be useful for debugging purposes.

There exist a few techniques that focus on computing probability bounds on errors. Daumas et al. [58] compute bounds on the probability that accumulated floating-point roundoff errors exceed a given threshold, using known inequalities on sums of probability distributions and distributions on the individual errors. This approach requires manual proofs and, unlike our techniques, does not consider input distributions.

The tool Chisel [137] considers and bounds the probability of errors occurring due to approximate hardware and is thus, in spirit, similar to our probabilistic error analysis approach presented in chapter 4. Chisel, however, only tracks the probability of a large error occurring, whereas our approach provides a more precise probability distribution and furthermore tracks the variable ranges probabilistically as well.

Following the publication of our tool PrAn for sound probabilistic roundoff error analysis (chapter 4), a subsequent effort has been made to compute sound probabilistic errors for floating-point programs. The tool PAF [45] focuses on soundly bounding the output distributions of floating-point programs and providing an over-approximation on roundoff errors, conditioned on the computation falling within an interval that encapsulates most, if not all, outputs.

PAF builds upon a technique similar to the one used in PrAn [11] but improves it by incorporating an SMT solver for reasoning about variable dependencies. It effectively removes regions from the state space with zero probability, resulting in more precise error bounds. However, it is important to note that the SMT solver highly influences the runtime of this tool, as it requires solver calls after each operation to determine these dependencies. Hence, the runtime is often larger than PrAn, as demonstrated in the paper.

Exception Analysis The static accuracy analysis tools are capable of proving the absence of special values (NaN and infinity) by computing an abstraction of the possible behaviors. However, as previously noted, their scalability is rather constrained when applied to real-world applications.

Scalable static analysis techniques have been developed for larger floating-point programs. *Abstract interpretation*-based methods are particularly suitable for verifying the absence of special values in these contexts. For example, Apron [110] offers a library of numerical abstract domains

that maintain soundness concerning floating-point arithmetic. This includes domains like polyhedra [37] and octagon [134], which, while more precise than the interval domain, are also notably more computationally intensive. ELINA [160] provides performance-optimized implementations of numerous numerical abstract domains. However, its polyhedra domain lacks support for floating-point arithmetic.

In our two-phase analysis, presented in chapter 5, we opted for the abstract interpretation-based tool Astrée [135]. This choice stems from its status as an industrial-strength, scalable tool that accommodates a wide spectrum of both C and, since November 2020, C++ programs [1]. However, these methods are expensive and incur high over-approximations without manual annotations.

In the space of *deductive verification*, substantial progress has been made in incorporating support for floating-point operations. The Why3 deductive verification framework [72] with front-ends for C and Ada programming languages through Frama-C [48] and SPARK [78] respectively and the Boogie intermediate verification language [125] support floating-point operations. These verifiers allow users to annotate floating-point programs with pre and post-conditions. From these annotations, verification conditions are automatically generated and then discharged with automated SMT solvers like Z3 [63].

A more recent work [10] added floating-point support to the KeY deductive verifier for Java programs that specifically focuses on proving the absence of the special values and compares multiple SMT solvers (Z3, CVC4 [17] and MathSAT [42]) with floating-point support. These methods could have been used for our purpose to verify the absence of special floating-point values like NaN and infinity; however, they require manual efforts for annotations, and the floating-point point support of SMT solvers is still quite limited.

In the domain of *bounded model checking* (BMC) [25] based on SAT and SMT theories, the typical approach involves performing a bit-precise translation of an input program, usually annotated with assertions and loops unrolled to a specified depth, into a formula whose satisfiability is then checked using SAT and SMT solvers. The BMC tools that handle floating-point programs are CBMC [119] and JBMC [47]. CBMC is tailored for checking C and C++ programs, while JBMC is specialized for the verification of Java bytecode. In theory, these tools can detect the presence of, or prove the absence of, special floating-point values up to a bounded program depth. However, it may be noted that these tools do not scale for large floating-point programs due to the inherent complexity of the arithmetic.

Other Probabilistic Analyses Our proposed probabilistic analysis technique is based on the propagation of uncertain probability distributions, using *probabilistic affine arithmetic*, following the work presented in [11]. This probabilistic affine arithmetic has also been extended by incorporating concentration of measure inequalities [31] to reduce the over-approximations. Furthermore, to reduce the combinatorial explosion of the size of the representation (e.g., number of focal elements), polynomial approximations of non-linear distributions based on polynomial

[1] It is worth noting that for our experiments, we leveraged Astrée for C programs, and our work was submitted prior to the release featuring C++ support.

arithmetic have also been proposed [157]. These methods have been used for tracking real-valued ranges in a program (but not errors). That said, these methods could potentially be used to improve the accuracy and reduce overapproximations of our approach.

In a broader context, the work by Sankaranarayanan et al. [156] explores the verification of probabilistic properties in programs with numerous paths by examining a finite subset. A combination of symbolic execution and volume computation provides bounds on the probability of the property holding. This work only considers linear programs and does not consider the effects of finite precision while handling more general programs and properties.

In chapter 3, we compared our proposed probabilistic analysis with an *exact symbolic probabilistic inference* technique [82]. This approach was used to generate the distribution of the output. However, we found that this inference method is not scalable for higher-order numerical programs. Since then, there have been a few advancements in this domain. For instance, the tool λPSI [83] is now capable of handling higher-order probabilistic programs with continuous distributions. It would be interesting to evaluate this tool's applicability for our purpose. Another recent tool, SPPL [150], translates probabilistic programs into symbolic sum-product expressions, effectively representing the joint distribution over all program variables. This allows for precise solutions to probabilistic inference queries. However, this tool is unsuitable for programs involving multivariate numeric transformations with real coefficients.

7.1.2 Dynamic Analysis

In contrast to sound analysis tools, dynamic analysis tools involve the execution of programs. Consequently, they are better suited for handling large programs with complex program and data structures. Within this category, various techniques and tools are available for floating-point programs to detect errors using fuzzing techniques, performing shadow executions, and sampling for probabilistic estimates. In the following section, we provide a concise overview of these techniques relevant to the work presented in this thesis. It is worth noting that, due to the fundamental nature of dynamic analysis, these tools only detect the presence of bugs in the executions observed and do not provide guarantees for all inputs.

Fuzzing One of the simplest yet effective fuzzing techniques is *Blackbox fuzzing*, which randomly generates (fuzzes) inputs without considering the internal program structure. This technique has been explored in the context of floating-point programs to find large roundoff errors by searching through the input space, guided either by heuristics like binary guided random testing [38] or through a genetic search algorithm [181]. Unfortunately, these tools are still limited to small floating-point programs and thus are not applicable for finding special floating-point values in large programs. Also, due to the inherent nature of blackbox fuzzing, they may not be able to find corner cases or paths in a reasonable time.

Guided greybox fuzzing presents an improvement over blackbox fuzzing by employing lightweight runtime monitoring, which enables the distinction of different program paths. It

also accesses the program's internal structure for coverage calculation. This method proved to be more effective in error detection than blackbox fuzzing, even though it might be somewhat less efficient in input generation.

In our work presented in chapter 5, we have utilized the guided greybox fuzzing tool AFLGo [27] to guide the program execution towards specific program locations where floating-point special values (NaN and infinity) may arise. Apart from AFLGo, there is a wide range of other targeted greybox fuzzers, such as those targeting specified program locations [36], rare branches [126], unexplored branches [127, 172], or potential vulnerabilities [80, 96, 18, 128].

The most effective (but expensive) technique is *whitebox fuzzing*, which utilizes the complete structures of programs, employing either concolic execution [84] or symbolic execution [16, 129, 91] to detect errors.

In the context of finding floating-point bugs, tools like KLEE-Float [129], FPGen [91], and Ariadne [16] exist that use symbolic execution techniques. These tools are capable of identifying issues in floating-point code, including overflows, large precision loss, and cancellation. KLEE-Float relies on floating-point SMT decision procedures, while Ariadne approximates the path constraints and uses the real-valued theory. FPGen introduces specialized inaccuracy checks to detect cancellations.

Additionally, white-box fuzzing has also been used for detecting overflows [77] and large roundoff errors [182] by phrasing the search as a mathematical optimization problem and by adapting the notion of condition numbers, respectively. Unfortunately, all these tools have been designed for relatively small programs and do not scale for real-world programs.

Shadow Execution Another compelling technique for detecting numerical errors involves shadow execution with high-precision computation [23, 22, 155, 40]. These techniques execute a program with a specific floating-point precision in parallel with a shadow execution in high precision, often using arbitrary precision arithmetic, to represent ideal real-valued execution. Here, each floating-point value is shadowed by a corresponding high-precision value, which is then compared with the original at various points of execution to detect errors. These tools instrument both during compilation [40] and in already compiled binaries [22, 155].

The tool FPDebug [22] has been shown to scale beyond small programs and detect cancellation and roundoff errors. In addition to cancellation, the tool PFPSanitizer [40] can also detect branch divergence and special floating-point values efficiently by executing shadow instructions marked by the user in parallel. It may be noted that shadow executions in high precision may incur high performance overhead. Once a cancellation has been identified, the tool Herbgrind [155] can help to locate its root cause, which may be in a different instruction than where the error becomes significant. Herbgrind is complementary to our work presented in chapter 4 and may additionally be used to locate the root causes of potential cancellation errors reported.

Probabilistic Approaches In addition to exact symbolic analysis, there exist sampling-based approaches for *probabilistic inference*. While they may not offer guaranteed bounds, they are known for their efficiency. We provide a brief overview of these approaches here, as we view them as complementary to our probabilistic analysis techniques.

The tool R2 combines MCMC sampling with program analysis to enhance efficiency [142]. Stan's Hamiltonian sampling can generate samples from high-probability regions [35]. Infer.NET implements various algorithms, including expectation and belief propagation, as well as Gibbs sampling [136]. It achieves efficiency by compiling graphical models into executable code. MAYHAP [154] statically constructs a distribution from a probabilistic program, which simplifies the Bayesian network before evaluating it through sampling.

7.2 Optimization of Finite-Precision

Optimizing the precision of finite-precision programs is crucial for improving accuracy, efficiency, or striking a balance between both. Various techniques have been developed for optimizing finite-precision programs. In this thesis, we specialize optimization for efficiently quantizing neural networks. Hence, this section provides a brief overview of numerical optimization and a discussion of the state-of-the-art quantization techniques for neural networks.

7.2.1 Numerical Optimization

Expression Rewriting In our work (chapter 5), we employed Daisy's expression rewriting technique [54] to reduce error bounds in numerical expressions. Daisy leverages a genetic search-based approach to systematically and soundly rewrite expressions, resulting in a notable reduction of error ranging from 3.3% to 33.3% in our context. Another static analysis tool, Salsa [50], adopts Sardana [103] internally for expression rewriting using abstract equivalence graphs.

Apart from sound analysis techniques, tools like Herbie [144], AutoRNP [177], and the tool by Wang et al. [173] focus on repairing numerical programs. They propose to generate partitions (regimes) of the input domain, detect an input subdomain that triggers high floating-point errors using dynamic analysis techniques, and then apply rewrites in order to repair high rounding errors. For rewriting, Herbie employs a greedy hill-climbing search guided by a dynamic error evaluation function, while other tools derive approximations with more specialized rewrite rules. It is worth noting that none of these tools completely preserve real-valued semantics.

In addition to the fully automated approaches, a recent tool, Odyssey [138], has introduced an interactive mode. This feature allows users to identify particularly problematic inputs with dynamic analysis, adjust input domains in real-time, and fine-tune the optimized expressions. This work is complementary to our fully automated approach.

Mixed-Precision Tuning For *sound* mixed precision tuning, there are tools available, such as FPTuner [39], Salsa [49], and Daisy [54]. FPTuner specializes in tuning floating-point arithmetic by solving a nonlinear interval-valued optimization problem. However, this continuous optimization approach is not applicable to fixed-point arithmetic due to its non-dynamic range. Alternatively, Salsa uses a combination of backward static analysis and SMT solving for mixed-

precision tuning of floating-point programs. Only Daisy can handle both floating-point and fixed-point arithmetic by employing an iterative search based on delta debugging. However, we have shown in our work (chapter 6) that the iterative search becomes intractable with the increased size of the program.

Other iterative approaches have also been explored for mixed-precision tuning for fixed-point arithmetic, for example, Min+1 or Max-1 [33], and a combination of Bayesian optimization and Min+1 [95]. Iterative approaches have been proposed to overcome difficulties in phrasing and, more importantly, solve sound global optimization problems. Alternatives are relaxing the integer to real-valued optimization [73], but these methods only work for Digital signal processors (DSPs). Generally, these techniques treat errors as uncorrelated and additive or evaluate them dynamically with simulation and thus do not guarantee soundness. A more recent tool, POP [12], finds mixed-precision assignments for floating-point arithmetic code by phrasing the problem as an ILP problem. POP uses dynamic analysis to infer variable ranges and thus also does not guarantee complete soundness.

Combined Techniques Some tools employ combined techniques for optimization. For example, Daisy [54] and Pherbie [151], built on top of Herbie, combine expression rewriting and mixed precision tuning. Daisy first rewrites the input expression into one that is equivalent but accurate under reals. It then follows up with mixed-precision tuning to generate code in mixed precision. Pherbie, on the other hand, interleaves accuracy optimization with precision tuning, resulting in improved running time. Like Herbie, Pherbie also does not completely preserve real-valued semantics. However, both these tools are applicable to both floating- and fixed-point arithmetic.

An attempt to combine sound and unsound optimizers Daisy and Herbie has been made in [19]. In this context, Daisy serves as a backend for validating Herbie's optimizations. This work also uncovered some limitations of both tools.

A recent tool, REGINA [148], partitions the input domain and systematically optimizes each part using established sound mixed-precision tuning or rewriting optimization routines. This approach effectively improves either the performance or accuracy, depending on the specific optimization, for floating-point programs.

While we primarily used rewriting to enhance the accuracy of numerical kernels in chapter 5, a combined technique could potentially offer further improvements.

Custom Operations There have been several other efforts to build custom operators for sums of products by constants with an accuracy requirement. For instance, the work of Dinechin et al. [61] presents an approach that permits to use of only a necessary amount of bits for intermediate products based on worst-case error analysis, and we have leveraged it in our ILP-based mixed fixed-precision tuner, Aster, as discussed in chapter 6.

Another noteworthy technique involves a multiplier-less approach where constant multiplications are performed using bit shifts and additions. Intermediate terms are also shared when several constants are multiplied [121], thus reducing the overall cost of the implemented hardware. A more recent work [81] presents an ILP-based solution for sound truncated multipliers, which takes as input the result's format and minimizes the number of full adders in the final

implementation. Our technique for neural network quantization (chapter 6) could potentially also benefit from this approach by further reducing costs, but the scalability might be a concern for word lengths beyond 16 bits.

7.2.2 Neural Network Quantization

We have tailored mixed fixed-point optimization for neural networks in chapter 6, which is typically called quantization in this domain. Quantization is a widely employed technique to reduce latencies and memory footprints. It involves converting high-precision floating-point weights, biases, and network activations into more cost-effective, low-precision operations.

While there have been numerous prior works on neural network quantization, most of these techniques have been applied to neural network classifiers in non-safety-critical contexts. They are based on dynamic analysis techniques and thus do not offer any accuracy guarantees. We provide a brief overview of these approaches here. However, it is important to note that they are fundamentally different from sound quantization for regression problems that we address.

The state-of-the-art quantization techniques typically focus on selecting a particular uniform (custom) floating-point [118, 147] or fixed-point precision [93, 130, 87] and showing empirically that it performs well on a particular data/benchmark set. For mixed-precision quantization, several approaches have emerged.

Particularly, there exists a technique [104] that performs mixed-precision tuning for interpolator networks. It uses dynamic range analysis and formulates the tuning as an ILP problem. However, it is limited to floating-point arithmetic. The tool Shiftry [120] specializes in mixed fixed-point quantization, which aligns closely with our work. It employs an automated process to iteratively decrease the precision of variables in recurrent neural networks, transitioning from 16 to 8 bits, and to dynamically compare classification accuracy with the original model. This enables the quantized models to run efficiently on memory-constrained hardware. Another approach to mixed-precision is to dynamically adapt to outliers, i.e., particularly large values, in inputs or weights by providing specialized hardware architectures [146, 159, 163].

For efficient implementations of neural networks, alternative number representations have also been considered. These include stochastic computing [70] for convolutional neural networks and posits [34] for deep neural networks. Additionally, tunable floating-point (TFP) precision has been explored, which allows for adjustments in the exponent and mantissa widths. These techniques either aim to minimize power consumption while maintaining acceptable accuracy during both the training and inference phases of deep neural networks [76] or to maximize dynamic range at network layers during inference phase [165].

In our work, we have analyzed the generated neural network implementation using the standard High-Level Synthesis (HLS) tool Xilinx [171]. For the most refined FPGA-based dot products, it is possible to substitute Xilinx with specialized hardware code generators like the FloPoCo tool [59]. This tool can optimize the implementation of mathematical operators tailored to a particular FPGA design, generating VHDL code.

8 Conclusion

Finite-precision programs are widespread, especially in safety-critical systems with constrained resources. Thus, analyzing and improving the accuracy of these programs while ensuring optimal resource utilization is crucial. Various techniques exist for sound analysis and optimization for finite-precision programs. However, they have traditionally focused on worst-case analysis, mostly in the context of straight-line code with limited support for conditionals and loops.

Throughout this thesis, we have presented our work on extending these analyses to consider input uncertainties and improving the scalability of both the analysis and optimization techniques. Below, we present our concluding remarks.

Remark 1: When extending the state-of-the-art data-flow-based probabilistic analyses in the context of finite-precision program executions and errors, we have observed significant overapproximations in the resultant distributions. To this end, techniques like pruning irrelevant input domains using an SMT solver or subdividing intervals into subintervals can be effective, but at the expense of increased runtime. Additionally, refining granularity, such as increasing discretizations, improves accuracy but also raises the complexity of the analyses. Therefore, striking a balance between accuracy and complexity is important for effectively utilizing the analyses.

Remark 2: Scaling finite-precision analyses to large programs is a challenging task. Fortunately, one analysis for the whole program with sophisticated finite-precision reasoning may not always be necessary. Typically, the numerically intensive parts (kernels) are small but concealed in large programs. Therefore, a more focused technique can be employed that analyzes only the kernels precisely and employs numerically imprecise but scalable techniques for the entire program, thus increasing scalability.

Remark 3: Instead of generic numerical analysis and optimization, specializing these techniques within specific application contexts can be highly beneficial. For instance, neural network controllers involving operations with matrices, vectors, and loops over data structures can be soundly quantized using static optimization-based methods with linearization and abstractions. We implemented these techniques to generate fixed-point mixed-precision NN codes that are guaranteed to satisfy user-provided error bounds and customized for specific hardware, such as FPGAs.

In summary, our work expands the horizons of state-of-the-art finite-precision analysis and

optimization, making them more suitable for practical scenarios. Nevertheless, there remains substantial work ahead.

Future Work In the domain of static analysis, we focused on probabilistic analysis of small numerical programs written in functional-style syntax and scaling up worst-case analysis for larger non-functional programs. As a natural next step, it would be imperative to explore methods to scale up probabilistic analysis in the context of larger non-functional programs. This step may require finding a balance between the accuracy and efficiency of the analysis.

Scaling up analyses and optimization for large programs poses significant challenges. In this thesis, we explored one direction to increase scalability through a combination of static and dynamic analyses. Exploring further combinations for analysis and optimization, such as integrating machine learning techniques with static analysis, where machine learning guides static analysis for a more focused approach, holds promise for further improvements in scalability and applicability.

We have specialized our optimization efforts in the field of neural network quantization. Further specialization in other domains, such as precision-aware control implementations in noisy environments, is a promising direction. Additionally, adopting more realistic models of resource costs specific to the application domain can help finely tune programs in the context of applications and the targeted hardware.

Moreover, optimizing based on worst-case roundoff errors can be excessively pessimistic, as errors do not consistently reach their worst-case magnitude. Furthermore, applications may be robust to some bounded noise in their execution. Consequently, introducing probabilistic analysis into the optimization process can be an exciting avenue, potentially enabling more aggressive optimization for specific program segments and, thus, enhancing overall resource utilization.

Finally, there exists substantial potential to expand numerical analysis and optimization in the context of heterogeneous high-performance computing (HPC) systems, integrating GPUs. Our work only focused on single-threaded numerical programs executed on CPUs with accessible source code. However, modern HPC systems introduce a whole new set of unique challenges, ranging from diverse number formats and the absence of exception status registers to limited access to source code and the prevalence of highly mixed precision options [86]. While recent efforts have been made towards detecting floating-point exceptions in GPU-based systems [123, 122], there remains significant work ahead to effectively verify, test, and optimize numerical programs running on these platforms.

In this thesis, we took a step forward toward analyzing and optimizing applications with realistic specifications, highlighting some potential future directions. Yet, the journey to practicality stretches far beyond, with many unexplored avenues ahead!

List of Algorithms

List of Figures

List of Tables

Bibliography

[1] FBench: Trigonometry Intense Floating Point Benchmark. https://www.fourmilab.ch/fbench/fbench.html. Accessed: 2020-10-05. Cited on page 79.

[2] Inverted-pendulum Control Problem. http://www.toddsifleet.com/projects/inverted-pendulum. Accessed: 2020-10-05. Cited on page 78.

[3] Molecular Dynamics. https://people.math.sc.edu/Burkardt/cpp_src/md/md.html. Accessed: 2020-10-05. Cited on page 79.

[4] N-body Problem. https://rosettacode.org/wiki/N-body_problem#C. Accessed: 2020-10-05. Cited on pages 7 and 78.

[5] Ray-casting Algorithm. https://rosettacode.org/wiki/Ray-casting_algorithm#C. Accessed: 2020-10-05. Cited on page 78.

[6] Simulated Test of Reactor Shielding. https://people.math.sc.edu/Burkardt/cpp_src/reactor_simulation/reactor_simulation.html. Accessed: 2020-10-05. Cited on page 79.

[7] GLPK (GNU Linear Programming Kit). https://www.gnu.org/software/glpk/, 2012. Cited on pages 34 and 58.

[8] Project Sklearn-porter. https://github.com/nok/sklearn-porter, 2018. Cited on pages 44, 59, and 78.

[9] ARCH-COMP2020 Github Repository, 2020. URL https://github.com/verivital/ARCH-COMP2020. Cited on page 107.

[10] R. Abbasi, J. Schiffl, E. Darulova, M. Ulbrich, and W. Ahrendt. Deductive Verification of Floating-Point Java Programs in KeY. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2021. Cited on page 120.

[11] A. Adje, O. Bouissou, J. Goubault-Larrecq, E. Goubault, and S. Putot. Static Analysis of Programs with Imprecise Probabilistic Inputs. In *Verified Software: Theories, Tools, Experiments (VSTTE)*. Springer, 2014. Cited on pages 6, 27, 44, 119, and 120.

[12] A. Adjé, D. Ben Khalifa, and M. Martel. Fast and Efficient Bit-Level Precision Tuning. In *Static Analysis (SAS)*. Springer, 2021. Cited on page 124.

[13] A. Anta and P. Tabuada. To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems. *IEEE Transactions on Automatic Control*, 55(9), 2010. Cited on page 5.

[14] A. Anta, R. Majumdar, I. Saha, and P. Tabuada. Automatic Verification of Control System Implementations. In *International conference on Embedded software (EMSOFT)*, 2010. Cited on pages 16, 29, and 37.

[15] C. Artigues, O. Koné, P. Lopez, and M. Mongeau. Mixed-Integer Linear Programming Formulations. In *Handbook on Project Management and Scheduling Vol. 1*. Springer, 2015. Cited on page 94.

[16] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic Detection of Floating-Point Exceptions. *ACM Sigplan Notices*, 48(1), 2013. Cited on page 122.

[17] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification (CAV)*. Springer, 2011. Cited on page 120.

[18] A. Basak Chowdhury and R. K. Medicherla. VeriFuzz: Program Aware Fuzzing: (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2019. Cited on page 122.

[19] H. Becker, P. Panchekha, E. Darulova, and Z. Tatlock. Combining Tools for Optimization and Analysis of Floating-Point Computations. In *Formal Methods (FM)*. Springer, 2018. Cited on page 124.

[20] H. Becker, N. Zyuzin, R. Monat, E. Darulova, M. O. Myreen, and A. Fox. A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018. Cited on page 117.

[21] D. Ben Khalifa, M. Martel, and A. Adjé. POP: A Tuning Assistant for Mixed-Precision Floating-Point Computations. In *Formal Techniques for Safety-Critical Systems (FTSCS)*. Springer, 2020. Cited on page 25.

[22] F. Benz, A. Hildebrandt, and S. Hack. A Dynamic Program Analysis to find Floating-Point Accuracy Problems. *ACM SIGPLAN Notices*, 47(6), 2012. Cited on pages 3, 67, 71, and 122.

[23] F. Benz, A. Hildebrandt, and S. Hack. A Dynamic Program Analysis to Find Floating-Point Accuracy Problems. *ACM SIGPLAN Notices*, 47(6), 2012. Cited on page 122.

[24] D. Berleant and C. Goodman-Strauss. Bounding the Results of Arithmetic Operations on Random Variables of Unknown Dependency using Intervals. *Reliable computing*, 1998. Cited on page 22.

[25] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Handbook of Satisfiability*, 185, 2009. Cited on page 120.

[26] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Programming Language Design and Implementation (PLDI)*, 2003. Cited on pages 3 and 67.

[27] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed Greybox Fuzzing. In *Computer and Communications Security (CCS)*, 2017. Cited on pages 8, 68, 70, 75, 81, and 122.

[28] S. Boldo, D. Gallois-Wong, and T. Hilaire. A Correctly-Rounded Fixed-Point-Arithmetic Dot-Product Algorithm. In *Computer Arithmetic (ARITH)*, 2020. Cited on pages 95, 103, and 104.

[29] P. Bonami, A. Lodi, A. Tramontani, and S. Wiese. On Mathematical Programming with Indicator Constraints. *Mathematical programming*, 151(1), 2015. Cited on page 102.

[30] O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot. A Generalization of P-boxes to Affine Arithmetic. *Computing*, 94(2-4), 2012. Cited on pages 23, 31, and 32.

[31] O. Bouissou, E. Goubault, S. Putot, A. Chakarov, and S. Sankaranarayanan. Uncertainty Propagation Using Probabilistic Affine Forms and Concentration of Measure Inequalities. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2016. Cited on pages 46, 49, and 120.

[32] R. Braojos, G. Ansaloni, and D. Atienza. A Methodology for Embedded Classification of Heartbeats Using Random Projections. In *Design, Automation & Test (DATE)*. IEEE, 2013. Cited on page 27.

[33] M.-A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie. An Automatic Word Length Determination Method. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2001. Cited on page 124.

[34] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi. Deep Positron: A Deep Neural Network Using the Posit Number System. In *Design, Automation & Test (DATE)*. IEEE, 2019. Cited on page 125.

[35] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A Probabilistic Programming Language. *Journal of Statistical Software*, 76 (1), 2017. Cited on pages 37, 49, and 123.

[36] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Computer and Communications Security (CCS)*, 2018. Cited on page 122.

[37] L. Chen, A. Miné, and P. Cousot. A Sound Floating-Point Polyhedra Abstract Domain. In *Asian Symposium on Programming Languages and Systems (APLAS)*. Springer, 2008. Cited on page 120.

[38] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient Search for Inputs Causing High Floating-point Errors. In *Principles and Practice of Parallel Programming (PPoPP)*, 2014. Cited on pages 3, 67, 81, and 121.

[39] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić. Rigorous Floating-Point Mixed-Precision Tuning. *ACM SIGPLAN Notices*, 52(1), 2017. Cited on pages 4, 25, 90, 94, and 123.

[40] S. Chowdhary and S. Nagarakatte. Parallel Shadow Execution to Accelerate the Debugging of Numerical Errors. In *Foundations of Software Engineering (FSE)*, 2021. Cited on page 122.

[41] S. Chowdhary and S. Nagarakatte. Fast Shadow Execution for Debugging Numerical Errors using Error Free Transformations. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2): 1845–1872, 2022. Cited on page 3.

[42] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2013. Cited on page 120.

[43] M. Claude and Y. Moy. *The Jessie plugin for Deductive Verification in Frama-C, Tutorial and Reference Manual*. INRIA Saclay-Île-de-France & LRI, CNRS UMR 8623, 2018. URL `http://krakatoa.lri.fr/jessie.html`. Cited on page 81.

[44] H. Collavizza, C. Michel, and M. Rueher. Searching Critical Values for Floating-Point Programs. In *International Conference on Testing Software and Systems (ICTSS)*. Springer, 2016. Cited on page 119.

[45] G. Constantinides, F. Dahlqvist, Z. Rakamarić, and R. Salvia. Rigorous Roundoff Error Analysis of Probabilistic Floating-Point Computations. In *Computer Aided Verification (CAV)*. Springer, 2021. Cited on page 119.

[46] G. F. Cooper. The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks. *Artificial Intelligence*, 42(2-3), 1990. Cited on page 37.

[47] L. Cordeiro, D. Kroening, and P. Schrammel. JBMC: Bounded Model Checking for Java Bytecode: (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2019. Cited on page 120.

[48] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A Software Analysis Perspective. In *Software Engineering and Formal Methods (SEFM)*. Springer, 2012. Cited on pages 3 and 120.

[49] N. Damouche and M. Martel. Mixed Precision Tuning with Salsa. 2018. Cited on pages 25 and 123.

[50] N. Damouche and M. Martel. Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. *Kalpa Publications in Computing*, 5, 2018. Cited on page 123.

[51] N. Damouche, M. Martel, P. Panchekha, J. Qiu, A. Sanchez-Stern, and Z. Tatlock. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *NSV*, 2016. Cited on page 77.

[52] N. Damouche, M. Martel, and A. Chapoutot. Improving the Numerical Accuracy of Programs by Automatic Transformation. *International Journal on Software Tools for Technology Transfer (STTT)*, 19, 2017. Cited on pages 17, 19, and 51.

[53] E. Darulova and V. Kuncak. Towards a Compiler for Reals. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(2), 2017. Cited on pages 2, 17, 19, 20, 27, 36, 51, 67, and 117.

[54] E. Darulova, E. Horn, and S. Sharma. Sound Mixed-Precision Optimization with Rewriting. In *International Conference on Cyber-Physical Systems (ICCPS)*, 2018. Cited on pages 4, 9, 24, 25, 90, 93, 108, 110, 111, 123, and 124.

[55] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2018. Cited on pages 2, 7, 17, 19, 20, 28, 30, 38, 50, 51, 67, 68, 71, 75, and 117.

[56] A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, and P. Panchekha. Scalable yet Rigorous Floating-Point Error Analysis. In *High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2020. Cited on pages 17 and 118.

[57] M. Daumas and G. Melquiond. Certification of Bounds on Expressions Involving Rounded Operators. *ACM Transactions on Mathematical Software (TOMS)*, 37(1), 2010. Cited on pages 2 and 27.

[58] M. Daumas, D. Lester, É. Martin-Dorel, and A. Truffert. Improved Bound for Stochastic Formal Correctness of Numerical Algorithms. *Innovations in Systems and Software Engineering*, 6, 2010. Cited on page 119.

[59] F. de Dinechin. Reflections on 10 Years of FloPoCo. In *Computer Arithmetic (ARITH)*. IEEE, 2019. Cited on page 125.

[60] F. De Dinechin, C. Q. Lauter, and G. Melquiond. Assisted Verification of Elementary Functions Using Gappa. In *ACM Symposium on Applied Computing*, 2006. Cited on pages 17, 19, 67, and 117.

[61] F. de Dinechin, M. Istoan, and A. Massouri. Sum-of-Product Architectures Computing Just Right. In *Application-Specific Systems, Architectures and Processors (ASAP)*, 2014. Cited on pages 90, 95, 101, 103, and 124.

[62] L. H. De Figueiredo and J. Stolfi. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms*, 37, 2004. Cited on pages 17 and 18.

[63] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008. Cited on pages 35 and 120.

[64] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. In *Formal Methods for Industrial Critical Systems (FMICS)*. Springer, 2009. Cited on pages 17, 19, 20, and 117.

[65] M. Dhiflaoui, S. Funke, C. Kwappik, K. Mehlhorn, M. Seel, E. Schomer, R. Schulte, and D. Weber. Certifying and Repairing Solutions to Large LPs How Good are LP-solvers? In *Symposium on Discrete Algorithms (SODA)*, 2003. Cited on pages 35 and 58.

[66] Diego Manzanas Lopez and Patrick Musau. ARCH-2019 Github Repository, 2019. URL `https://github.com/verivital/ARCH-2019`. Cited on pages 106 and 107.

[67] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Learning and Verification of Feedback Control Systems using Feedforward Neural Networks. *IFAC-PapersOnLine*, 51(16), 2018. Cited on pages 4, 9, 89, and 107.

[68] S. Dutta, X. Chen, and S. Sankaranarayanan. Reachability Analysis for Neural Feedback Systems using Regressive Polynomial Rule Inference. In *Hybrid Systems: Computation and Control (HSCC)*, 2019. Cited on pages 4, 89, and 106.

[69] M. Everett, G. Habibi, and J. P. How. Efficient Reachability Analysis of Closed-Loop Systems with Neural Network Controllers. In *International Conference on Robotics and Automation (ICRA)*. IEEE, 2021. Cited on pages 4 and 89.

[70] S. R. Faraji, M. H. Najafi, B. Li, D. J. Lilja, and K. Bazargan. Energy-Efficient Convolutional Neural Networks with Deterministic Bit-Stream Processing. In *Design, Automation & Test (DATE)*. IEEE, 2019. Cited on page 125.

[71] S. Ferson, V. Kreinovich, L. Ginzburg, D. S. Myers, and K. Sentz. Constructing Probability Boxes and Dempster-Shafer Structures. Technical report, Sandia National Laboratories, 2003. Cited on page 21.

[72] J.-C. Filliâtre and A. Paskevich. Why3—Where Programs Meet Provers. In *European Symposium on Programming Languages and Systems (ESOP)*. Springer, 2013. Cited on page 120.

[73] P. D. Fiore. Efficient Approximate Wordlength Optimization. *IEEE Transactions on Computers*, 57 (11), 2008. Cited on page 124.

[74] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2), 2007. Cited on page 34.

[75] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library With Correct Rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2), 2007. Cited on pages 6 and 58.

[76] M. Franceschi, A. Nannarelli, and M. Valle. Tunable Floating-Point for Artificial Neural Networks. In *International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2018. Cited on page 125.

[77] Z. Fu and Z. Su. Effective Floating-Point Analysis via Weak-Distance Minimization. In *Programming Language Design and Implementation (PLDI)*, 2019. Cited on pages 3, 67, and 122.

[78] C. Fumex, C. Marché, and Y. Moy. Automating the Verification of Floating-Point Programs. In *Verified Software Theories, Tools, and Experiments (VSTTE)*. Springer, 2017. Cited on page 120.

[79] G. Gamrath, D. Anderson, K. Bestuzheva, and W.-K. C. et al. The SCIP Optimization Suite 7.0. Technical report, 2020. URL `http://www.optimization-online.org/DB_HTML/2020/03/7705.html`. Cited on page 105.

[80] V. Ganesh, T. Leek, and M. Rinard. Taint-Based Directed Whitebox Fuzzing. In *International Conference on Software Engineering (ICSE)*. IEEE, 2009. Cited on page 122.

[81] R. Garcia, A. Volkova, and M. Kumm. Truncated Multiple Constant Multiplication with Minimal Number of Full Adders. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2022. Cited on page 124.

[82] T. Gehr, S. Misailovic, and M. Vechev. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification (CAV)*. Springer, 2016. Cited on pages 27, 30, 38, 49, and 121.

[83] T. Gehr, S. Steffen, and M. Vechev. λPSI: Exact Inference for Higher-Order Probabilistic Programs. In *Programming Language Design and Implementation (PLDI)*, 2020. Cited on page 121.

[84] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security (NDSS)*, volume 8, 2008. Cited on page 122.

[85] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM computing surveys (CSUR)*, 23(1), 1991. Cited on pages 1, 13, and 71.

[86] G. Gopalakrishnan, I. Laguna, A. Li, P. Panchekha, C. Rubio-González, and Z. Tatlock. Guarding Numerics Amidst Rising Heterogeneity. In *International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2021. Cited on page 128.

[87] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma. Compiling KB-Sized Machine Learning Models to Tiny IoT Devices. In *Programming Language Design and Implementation (PLDI)*, 2019. Cited on pages 4, 89, 110, and 125.

[88] E. Goubault and S. Putot. Static Analysis of Finite Precision Computations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011. Cited on pages 2, 3, 27, and 51.

[89] E. Goubault and S. Putot. Robustness Analysis of Finite Precision Implementations. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2013. Cited on pages 2, 27, and 67.

[90] D. Guidotti, S. Demarchi, A. Tacchella, and L. Pulina. The Verification of Neural Networks Library (VNN-LIB), 2023. URL `https://www.vnnlib.org`. Cited on pages 10, 90, 106, and 107.

[91] H. Guo and C. Rubio-González. Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution. In *International Conference on Software Engineering (ICSE)*, 2020. Cited on page 122.

[92] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep Learning with Limited Numerical Precision. In *International Conference on Machine Learning (ICML)*, 2015. Cited on pages 4 and 89.

[93] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep Learning with Limited Numerical Precision. In *International conference on machine learning (ICML)*. PMLR, 2015. Cited on page 125.

[94] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL `https://www.gurobi.com`. Cited on page 106.

[95] V.-P. Ha and O. Sentieys. Leveraging Bayesian Optimization to Speed Up Automatic Precision Tuning. In *Design, Automation & Test (DATE)*. IEEE, 2021. Cited on page 124.

[96] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for {Overflows}: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security Symposium*, 2013. Cited on page 122.

[97] L. Hatton and A. Roberts. How Accurate Is Scientific Software? *IEEE Transactions on Software Engineering*, 20(10), 1994. Cited on page 17.

[98] C. Huang, J. Fan, W. Li, X. Chen, and Q. Zhu. ReachNN: Reachability Analysis of Neural-Network Controlled Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s), 2019. Cited on pages 92 and 94.

[99] C. S. IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008. Cited on page 49.

[100] C. S. IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019*, 2019. Cited on pages 14 and 67.

[101] I. ILOG. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation*, 46(53), 2009. Cited on page 106.

[102] W. R. Inc. Mathematica, Version 10.4. `https://www.wolfram.com/mathematica/`, 2018. Champaign, IL. Cited on pages 31 and 38.

[103] A. Ioualalen and M. Martel. Sardana: An Automatic Tool for Numerical Accuracy Optimization. In *Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN)*, 2012. Cited on page 123.

[104] A. Ioualalen and M. Martel. Neural Network Precision Tuning. In *Quantitative Evaluation of Systems (QEST)*. Springer, 2019. Cited on page 125.

[105] A. Isychev and E. Darulova. Scaling up Roundoff Analysis of Functional Data Structure Programs. In *Static Analysis Symposium (SAS)*, 2023. Cited on page 118.

[106] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee. Verisig: Verifying Safety Properties of Hybrid Systems with Neural Network Controllers. In *Hybrid Systems: Computation and Control (HSCC)*, 2019. Cited on pages 106 and 107.

[107] R. Ivanov, K. Jothimurugan, S. Hsu, S. Vaidya, R. Alur, and O. Bastani. Compositional Learning and Verification of Neural Network Controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s), 2021. Cited on pages 92 and 94.

[108] A. Izycheva and E. Darulova. On Sound Relative Error Bounds for Floating-Point Arithmetic. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2017. Cited on pages 19 and 52.

[109] A. Izycheva, E. Darulova, and H. Seidl. Synthesizing Efficient Low-Precision Kernels. In *Automated Technology for Verification and Analysis (ATVA)*, 2019. Cited on pages 4, 99, and 111.

[110] B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification (CAV)*. Springer, 2009. Cited on page 119.

[111] T. T. Johnson, D. M. Lopez, P. Musau, H.-D. Tran, E. Botoeva, F. Leofante, A. Maleki, C. Sidrane, J. Fan, and C. Huang. ARCH-COMP20 Category Report: Artificial Intelligence and Neural Network Control Systems (AINNCS) for Continuous and Hybrid Systems Plants. In *ARCH*, EPiC Series in Computing, 2020. Cited on pages 4, 10, 89, 90, 91, 92, and 106.

[112] K. D. Julian and M. J. Kochenderfer. A Reachability Method for Verifying Dynamical Systems with Deep Neural Network Controllers. *arXiv preprint arXiv:1903.00520*, 2019. Cited on page 107.

[113] W. Kahan. IEEE Standard 754 for Binary Floating-Point Arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776), 1996. Cited on page 1.

[114] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. LULESH Programming Model and Performance Ports Overview. Technical Report LLNL-TR-608824, 2012. Cited on pages 1 and 79.

[115] C. Keil. Lurupa-Rigorous Error Bounds in Linear Programming. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006. Cited on pages 35 and 58.

[116] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, 27(3), 2015. Cited on page 81.

[117] M. J. Kochenderfer. *Decision Making Under Uncertainty: Theory and Application*. MIT press, 2015. Cited on page 1.

[118] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof, et al. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017. Cited on page 125.

[119] D. Kroening and M. Tautschnig. CBMC–C Bounded Model Checker: (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2014. Cited on pages 3, 8, 68, 76, 80, and 120.

[120] A. Kumar, V. Seshadri, and R. Sharma. Shiftry: RNN Inference in 2KB of RAM. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020. Cited on pages 4, 89, 110, and 125.

[121] M. Kumm. Optimal Constant Multiplication Using Integer Linear Programming. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(5), 2018. Cited on page 124.

[122] I. Laguna, X. Li, and G. Gopalakrishnan. BINFPE: Accurate Floating-Point Exception Detection for GPU Applications. In *State Of the Art in Program Analysis (SOAP)*, 2022. Cited on page 128.

[123] I. Laguna, T. Tirpankar, X. Li, and G. Gopalakrishnan. FPChecker: Floating-Point Exception Detection Tool and Benchmark for Parallel and Distributed HPC. In *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2022. Cited on page 128.

[124] D. U. Lee, A. A. Gaffar, R. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-Guaranteed Bit-Width Optimization. *IEEE Transactions on Computer Aided Design of Integrated Circuits Systems (TCAD)*, 25(10), 2006. Cited on page 37.

[125] K. R. M. Leino. This is Boogie 2. 2008. URL `https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/`. Cited on page 120.

[126] C. Lemieux and K. Sen. FAIRFUZZ: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Automated Software Engineering (ASE)*, 2018. Cited on page 122.

[127] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. Steelix: Program-State Based Binary Fuzzing. In *Foundations of Software Engineering (FSE)*, 2017. Cited on page 122.

[128] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, and R. Beyah. V-Fuzz: Vulnerability Prediction-Assisted Evolutionary Fuzzing for Binary Programs. *IEEE Transactions on Cybernetics*, 52(5), 2020. Cited on page 122.

[129] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zahl, and K. Wehrle. Floating-Point Symbolic Execution: A Case Study in N-Version Programming. In *Automated Software Engineering (ASE)*. IEEE, 2017. Cited on pages 3, 67, 81, and 122.

[130] D. Lin, S. Talathi, and S. Annapureddy. Fixed Point Quantization of Deep Convolutional Networks. In *International conference on machine learning (ICML)*. PMLR, 2016. Cited on page 125.

[131] D. M. Lopez, P. Musau, H.-D. Tran, and T. T. Johnson. Verification of Closed-loop Systems with Neural Network Controllers. *EPiC Series in Computing*, 61, 2019. Cited on pages 10, 90, and 106.

[132] V. Magron, G. Constantinides, and A. Donaldson. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Transactions on Mathematical Software (TOMS)*, 43(4), 2017. Cited on pages 17, 67, and 118.

[133] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve. Minotaur: Adapting Software Testing Techniques for Hardware Errors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019. Cited on page 71.

[134] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation (HOSC)*, 19, 2006. Cited on page 120.

[135] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *Embedded Real Time Software and Systems (ERTS)*, 2016. Cited on pages 3, 8, 67, 68, 70, 80, and 120.

[136] T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.6, 2014. Microsoft Research Cambridge. http://research.microsoft.com/infernet. Cited on pages 37, 49, and 123.

[137] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability and Accuracy-Aware Optimization of Approximate Computational Kernels, volume = 49, year = 2014. *ACM Sigplan Notices*, (10). Cited on pages 2, 49, 57, and 119.

[138] E. Misback, C. Chan, B. Saiki, E. Jun, Z. Tatlock, and P. Panchekha. Odyssey: An Interactive Workbench for Expert-Driven Floating-Point Expression Rewriting. *arXiv preprint arXiv:2305.10599*, 2023. Cited on page 123.

[139] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009. Cited on pages 17, 70, and 104.

[140] M. Moscato, L. Titolo, A. Dutle, and C. A. Munoz. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *Computer Safety, Reliability, and Security (SAFECOMP)*. Springer, 2017. Cited on pages 2, 17, 51, 67, and 118.

[141] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011. Cited on page 3.

[142] A. Nori, C.-K. Hur, S. Rajamani, and S. Samuel. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *Conference on Artificial Intelligence (AAAI)*, 2014. Cited on page 123.

[143] A. V. Nori, C.-K. Hur, S. K. Rajamani, and S. Samuel. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *Conference on Artificial Intelligence (AAAI)*, 2014. Cited on pages 37 and 49.

[144] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically Improving Accuracy for Floating Point Expressions. *ACM SIGPLAN Notices*, 50(6), 2015. Cited on page 123.

[145] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The Limitations of Deep Learning in Adversarial Settings. In *European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016. Cited on page 44.

[146] E. Park, D. Kim, and S. Yoo. Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation. In *International Symposium on Computer Architecture (ISCA)*, 2018. Cited on pages 4, 89, and 125.

[147] K. Pradeep, K. Kamalavasan, R. Natheesan, and A. Pasqual. EdgeNet: SqueezeNet like Convolution Neural Network on Embedded FPGA. In *International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2018. Cited on page 125.

[148] R. Rabe, A. Izycheva, and E. Darulova. Regime Inference for Sound Floating-Point Optimizations. *Transactions on Embedded Computing Systems (TECS)*, 20(5s), 2021. Cited on page 124.

[149] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning Assistant for Floating-point Precision. In *High Performance Computing, Networking, Storage and Analysis (SC)*, 2013. Cited on page 77.

[150] F. A. Saad, M. C. Rinard, and V. K. Mansinghka. SPPL: Probabilistic Programming with Fast Exact Symbolic Inference. In *Programming Language Design and Implementation (PLDI)*, 2021. Cited on page 121.

[151] B. Saiki, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock. Combining Precision Tuning and Rewriting. In *Computer Arithmetic (ARITH)*. IEEE, 2021. Cited on page 124.

[152] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. *ACM SIGPLAN Notices*, 46(6), 2011. Cited on pages 2, 49, and 57.

[153] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and Verifying Probabilistic Assertions. In *Programming Language Design and Implementation (PLDI)*, 2014. Cited on page 49.

[154] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and Verifying Probabilistic Assertions. In *Programming Language Design and Implementation (PLDI)*, 2014. Cited on page 123.

[155] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock. Finding Root Causes of Floating Point Error. In *Programming Language Design and Implementation (PLDI)*, 2018. Cited on page 122.

[156] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. In *Programming Language Design and Implementation (PLDI)*, 2013. Cited on pages 49 and 121.

[157] S. Sankaranarayanan, Y. Chou, E. Goubault, and S. Putot. Reasoning about Uncertainties in Discrete-Time Dynamical Systems Using Polynomial Forms. *Advances in Neural Information Processing Systems (NeurIPS)*, 33, 2020. Cited on page 121.

[158] G. Shafer. *A Mathematical Theory Of Evidence*, volume 42. Princeton university press, 1976. Cited on pages 20 and 21.

[159] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *International Symposium on Computer Architecture (ISCA)*, 2018. Cited on pages 4, 89, and 125.

[160] G. Singh, M. Püschel, and M. Vechev. Fast Polyhedra Abstract Domain. In *Principles of Programming Languages (POPL)*, 2017. Cited on page 120.

[161] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *Formal Methods (FM)*, 2015. Cited on pages 36, 67, and 118.

[162] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(1), 2018. Cited on pages 2, 17, and 27.

[163] Z. Song, B. Fu, F. Wu, Z. Jiang, L. Jiang, N. Jing, and X. Liang. DRQ: Dynamic Region-based Quantization for Deep Neural Network Acceleration. In *International Symposium on Computer Architecture (ISCA)*, 2020. Cited on pages 4, 89, and 125.

[164] X. Sun, H. Khedr, and Y. Shoukry. Formal Verification of Neural Network Controlled Autonomous Systems. In *Hybrid Systems: Computation and Control (HSCC)*, 2019. Cited on pages 92 and 94.

[165] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. Rush, D. Brooks, and G.-Y. Wei. Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference. In *Design Automation Conference (DAC)*. IEEE, 2020. Cited on page 125.

[166] **D. Lohar**, E. Darulova, S. Putot, and E. Goubault. Discrete Choice in the Presence of Numerical Uncertainties. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 37(11), 2018. Cited on page 28.

[167] **D. Lohar**, M. Prokop, and E. Darulova. Sound Probabilistic Numerical Error Analysis. In *Integrated Formal Methods (iFM)*. Springer, 2019. Cited on page 50.

[168] **D. Lohar**, C. Jeangoudoux, J. Sobel, E. Darulova, and M. Christakis. A Two-Phase Approach for Conditional Floating-Point Verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2021. Cited on page 68.

[169] **D. Lohar**, C. Jeangoudoux, A. Volkova, and E. Darulova. Sound Mixed Fixed-Point Quantization of Neural Networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 22(5s), 2023. Cited on page 91.

[170] H. Tran, F. Cai, D. M. Lopez, P. Musau, T. T. Johnson, and X. D. Koutsoukos. Safety Verification of Cyber-Physical Systems with Reinforcement Learning Control. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s), 2019. Cited on pages 92 and 94.

[171] Vivado Lab Solutions. Vivado Design Suite, 2021. URL `https://www.xilinx.com`. Cited on pages 90, 105, 108, 111, and 125.

[172] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. In *International Conference on Software Engineering (ICSE): Companion Proceeedings*, 2018. Cited on page 122.

[173] X. Wang, H. Wang, Z. Su, E. Tang, X. Chen, W. Shen, Z. Chen, L. Wang, X. Zhang, and X. Li. Global Optimization of Numerical Programs via Prioritized Stochastic Algebraic Transformations. In *International Conference on Software Engineering (ICSE)*. IEEE, 2019. Cited on page 123.

[174] Q. Xu, T. Mytkowicz, and N. S. Kim. Approximate Computing: A Survey. *IEEE Design Test*, 33(1), 2016. ISSN 2168-2356. Cited on page 49.

[175] Z. Xu, Y. Liu, S. Qin, and Z. Ming. Output Range Analysis for Feed-Forward Deep Neural Networks via Linear Programming. *IEEE Transactions on Reliability*, 2022. Cited on pages 10 and 96.

[176] R. Yates. Fixed-Point Arithmetic: An Introduction. *Digital Signal Labs*, 81(83), 2009. Cited on pages 1, 16, and 92.

[177] X. Yi, L. Chen, X. Mao, and T. Ji. Efficient Automated Repair of High Floating-Point Errors in Numerical Libraries. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019. Cited on page 123.

[178] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. The Fuzzing Book, 2019. Cited on page 3.

[179] H. Zitoun, C. Michel, M. Rueher, and L. Michel. Search Strategies for Floating Point Constraint Systems. In *Principles and Practice of Constraint Programming (CP)*. Springer, 2017. Cited on page 44.

[180] H. Zitoun, C. Michel, M. Rueher, and L. Michel. Search Strategies for Floating Point Constraint Systems. In *Principles and Practice of Constraint Programming (CP)*. Springer, 2017. Cited on page 119.

[181] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In *International Conference on Software Engineering (ICSE)*, volume 1, 2015. Cited on page 121.

[182] D. Zou, M. Zeng, Y. Xiong, Z. Fu, L. Zhang, and Z. Su. Detecting Floating-Point Errors via Atomic Conditions. *Proceedings of the ACM on Programming Languages*, 4(POPL), 2020. Cited on pages 3, 67, 81, and 122.