Using Runtime Floating Point Accuracy Feedback to Make Automated

Precision/Performance Improvements or Tradeoffs

by

Ralph Nathan

Department of Electrical and Computer Engineering
Duke University

Date:_______________________

Approved:

___________________________
Daniel Sorin, Supervisor

___________________________
Andrew Hilton

___________________________
Benjamin Lee

___________________________
Helia Naeimi

___________________________
Xiaobai Sun

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University

2015
ABSTRACT

Using Runtime Floating Point Accuracy Feedback to Make Automated Performance/Precision Improvements or Tradeoffs

by

Ralph Nathan

Department of Electrical and Computer Engineering
Duke University

Date: ________________________

Approved:

___________________________
Daniel Sorin, Supervisor

___________________________
Andrew Hilton

___________________________
Benjamin Lee

___________________________
Helia Naeimi

___________________________
Xiaobai Sun

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University

2015
Abstract

In this thesis, we design frameworks for efficient and accurate floating point computation. The principle underlying our frameworks is making information usually discarded in the hardware, specifically, in the floating point unit, visible to the programmer. The programmer, or automated tools that we developed, can use this information to make accuracy/performance improvements or tradeoffs.

We make the error of floating point additions architecturally visible to programmers and experimentally demonstrate that programmers can use this error to improve the accuracy of their applications or improve the application’s performance without affecting the accuracy of the final result. To free programmers from having to manually instrument their code, we develop a compiler pass to automate this process.

We also design a framework to profile applications to measure undesirable numerical behavior at the floating point operation level. We develop a debugger that programmers can use to find variables with “bad” behavior. In addition, we present a profile driven mixed precision analysis framework that heuristically determines the precision of all variables in an application based on their numerical behavior. We experimentally evaluate the mixed precision analysis to show that it can generate a range of results with different accuracies and precisions.
## Contents

Abstract ......................................................................................................................................... iv

List of Tables .................................................................................................................................. xiv

List of Figures .................................................................................................................................. xv

Acknowledgements ................................................................................................................... xxi

1. Introduction ................................................................................................................................ 1

2. Background .................................................................................................................................... 4

    2.1 IEEE-754 Standard .................................................................................................................. 4

    2.2 Hardware Support ..................................................................................................................... 6

    2.3 Software Libraries to Improve Precision ............................................................................... 8

        2.3.1 Double-Double Arithmetic ............................................................................................... 9

        2.3.2 Simulating Arbitrary-Precision FPU ................................................................................... 9

    2.4 Changing Algorithms to Improve Accuracy ............................................................................. 10

    2.5 Detecting Numerical Errors .................................................................................................... 11
3.5.2 Kernel Microbenchmarks

3.5.2.1 Summation of Long Sequence of Numbers

3.5.2.2 Dot-Product

3.5.3 Benchmarks

3.5.3.1 LU Decomposition

3.5.3.2 N-Body Simulation of Electrical Charges

3.5.3.3 Financial Pricing Simulation

3.6 REBits vs. Software Numerical Methods

3.6.1 Comparison to Software-Only Schemes

3.6.2 REBits to Streamline Double-Double Arithmetic

3.7 Related Work

3.8 Conclusions

4. Profile Driven Mixed Precision Analysis

4.1 Introduction

4.2 Numerical Profiling
4.2.1 Behaviors ................................................................................................................ 57

4.2.1.1 Size of Rounding Error ................................................................................ 57

4.2.1.2 Cancellation ................................................................................................. 60

4.2.1.3 Adding Small Numbers to a Large Number ............................................. 61

4.2.1.4 Closeness to Infinity ..................................................................................... 62

4.2.2 Limitations ............................................................................................................. 63

4.2.3 Implementation ..................................................................................................... 63

4.2.4 Hardware Accelerated Profiling ......................................................................... 66

4.3 Profile Driven Numerical Debugging .................................................................... 66

4.3.1 Numerical Debugger ............................................................................................ 67

4.3.2 Example .................................................................................................................. 69

4.3.3 Usage ....................................................................................................................... 71

4.4 Profile Driven Automated Mixed Precision .......................................................... 75

4.4.1 Why Automated Mixed Precision? ................................................................. 75

4.4.2 Determining Bad Behaviors ............................................................................... 77
4.4.2.1 Cancellation Percentage ................................................................. 78
4.4.2.2 Range Percentage ........................................................................ 78
4.4.2.3 Percentage Threshold Exceeded ................................................. 79
4.4.2.4 Error Ratio .................................................................................... 79
4.4.2.5 Exponent Difference ................................................................. 80
4.4.3 Classifying Instructions ............................................................... 80
4.4.3.1 Single Instruction Promotion Bin ............................................... 80
4.4.3.2 No-Error Bin ............................................................................... 81
4.4.3.3 Cancellation Bin ......................................................................... 81
4.4.3.4 "Other" Bin ................................................................................. 82
4.4.3.5 Instructions in Multiple Bins ........................................................... 82
4.4.4 Rewriting the IR .......................................................................... 83
4.4.4.1 Cancellation Pass ...................................................................... 84
4.4.4.2 Single Instruction Promotion Pass ......................................... 86
4.4.5 Benefits of Profile Driven Mixed Precision Analysis ................. 87
4.5 Choosing Threshold Vectors.................................................................87

4.5.1 Result Equivalence.............................................................................88

4.5.2 Instruction Change Equivalence.........................................................89

4.5.3 Prime Threshold Vectors.................................................................90

4.6 Experimental Methodology ................................................................93

4.6.1 Methodology: Profiler..........................................................................93

4.6.2 Methodology: Automated Mixed Precision.........................................94

4.6.3 Evaluation Environment......................................................................94

4.7 Experimental Evaluation.....................................................................95

4.7.1 Case Study 1: Numerical Integration.................................................96

4.7.1.1 Mixed Precision Results Range......................................................96

4.7.1.2 Equivalences..................................................................................97

4.7.1.3 Performance....................................................................................102

4.7.2 Case Study 2: Gravitational N-Body Simulation..............................104

4.7.2.1 Mixed Precision Results Range......................................................105
**List of Tables**

Table 1: IEEE-754 Layouts .................................................................................................................. 5

Table 2. Schemes Compared ............................................................................................................. 31

Table 3. Results: Sea Heights ......................................................................................................... 39

Table 4. Improving software methods using $REBits$ ........................................................................ 48

Table 5: Evaluation of Rump’s Polynomial ..................................................................................... 70

Table 6: Threshold Values Range ................................................................................................... 88

Table 7: Unique Result Counts for each Accuracy Metric for QR Decomposition with Learning Matrix Size 100 x 100 drawn from U[-1, 1] ........................................................................ 122

Table 8: Profiling Overheads ........................................................................................................... 144
List of Figures

Figure 1: Single Precision Floating Point Layout ................................................................. 5

Figure 2. MixedHW operations required for vectorized software................................. 19

Figure 3. REBits for Floating Point Addition. Hardware extensions shaded............. 20

Figure 4. 2-Norm Software. REBits extensions highlighted in bold text.................... 26

Figure 5: Simplified LLVM IR before and after REBits Instrumentation’ ................. 29

Figure 6. REBits-32 Code for Sum of Random Numbers ............................................ 34

Figure 7. Results: Summation of Random Numbers (32-bit Inputs) ....................... 37

Figure 8: Summation of Random Numbers (64-bit Inputs) ........................................ 38

Figure 9. Results: Dot-product ...................................................................................... 41

Figure 10. Results: LU decomposition ......................................................................... 43

Figure 11. Results: N-Body simulation ........................................................................ 44

Figure 12. Results: Financial Pricing ............................................................................. 46

Figure 13. Original Double-Double addition ................................................................ 50
Figure 39: Prime Result Equivalent Threshold Vectors for N-Body Simulation with Learning at 64 Bodies ................................................................. 113

Figure 40: Instruction Change Equivalence for N-Body Simulation with Learning at 64 Bodies ......................................................................................... 114

Figure 41: Double Precision Instruction Counts for N-Body Simulation with Learning at 64 Bodies ................................................................................. 115

Figure 42: Best Results at Different Learning Dataset Sizes for N-Body Simulation ..... 116

Figure 43: Performance vs. Accuracy for N-Body Simulation with 1000 Bodies with Learning at 64 Bodies .............................................................. 118

Figure 44: Range of Mixed Precision Results for QR Decomposition with Learning Matrix Size 100 x 100 drawn from U[-1, 1] ......................................................... 121

Figure 45: Result Equivalence for QR Decomposition with Learning Matrix Size 100 x 100 drawn from U[-1, 1] ................................................................. 124

Figure 46: Prime Result Equivalent Threshold Vectors for QR Decomposition with Learning Matrix Size 100 x 100 drawn from U[-1, 1] ............................... 125

Figure 47: Instruction Change Equivalence for QR Decomposition with Learning Matrix Size 100 x 100 drawn from U[-1, 1] ......................................................... 126
Figure 48: Double Precision Instruction Precision Counts for QR Decomposition with Learning Matrix Size 100 x 100 drawn from U[-1, 1] ........................................................... 127

Figure 49: Effect of Size for U[0, 1] .......................................................................................... 129

Figure 50: Effect of Size for U[-1e6, 1e6] ................................................................................ 130

Figure 51: Effect of Size for U[-1, 1] .................................................................................... 131

Figure 52: Effect of Size for N(0, 1e6) .................................................................................... 132

Figure 53: Performance vs. Accuracy for QR Decomposition for Matrix Size 100 x 100 with Learning Matrix Size 100 x 100 drawn from U[-1, 1] .................................................. 134

Figure 54: Range of Mixed Precision Answers for LU with Learning Matrix Size 100 x 100 drawn from U[-1e6, 1e6] ................................................................................................... 136

Figure 55: Cached Performance for Scalar LU with Matrix Size 500 x 500 with Learning Matrix Size 100 x 100 drawn from U[-1e6, 1e6] ................................................................. 138

Figure 56: Cached Performance for Vectorized LU with Matrix Size 500 x 500 with Learning Matrix Size 100 x 100 drawn from U[-1e6, 1e6] ................................................................. 139

Figure 57: Scalar Streaming Performance for LU with Matrix Size 4000 x 4000 with Learning Matrix Size 100 x 100 drawn from U[-1e6, 1e6] ................................................................. 141
Figure 58: Vectorized Streaming Performance for LU with Matrix Size 4000 x 4000 with Learning Matrix Size 100 x 100 drawn from U[-1e6, 1e6] .......................... 142
Acknowledgements

There were so many people who helped me get my thesis to where it is today. In this section, I would like to acknowledge some of their many contributions. Firstly, I would like to thank my parents for their support throughout the years.

Violeta, my wonderful girlfriend, I do not think words can express my gratitude for the support you have provided me, both as an undergrad and as a graduate student. You were always there to support and encourage me, during both good and bad times. You inspire me to push myself and have played a huge role in molding me into the person I am today. Without your encouragement and support, I could not have done this. Also, thank you for proofreading the entirety of this dissertation.

I would like to thank my advisor, Daniel Sorin, who introduced me to computer architecture and research. From you, I learned the benefits and importance of good organization and timekeeping. Thank you for the research freedom you gave me to explore different areas and find projects that excite me (even ones that did not seem fruitful at first glance). I am truly grateful for your mentorship. You have made me a better engineer, researcher, and writer. In addition, the many delicious cookies you gave me helped me push towards deadlines.

I would like to express my gratitude to Helia Naeimi, my mentor at Intel Corporation. Helia, I have learnt so much from you about industrial research. The wisdom and the insights you have given me, have, and will prove, invaluable. You
opened so many doors for me and gave me many opportunities at Intel. Thanks to you, I had two tremendous experiences at Intel! Shih-Lien, I would like to thank you for giving me the opportunity to work in your group twice. I really enjoyed the chats we had in Santa Clara and Hillsboro and wish we could have had more. John Gustafson, thank you for introducing me to the area of research that ended up being the topic of my dissertation.

To the other members of my committee, Andrew Hilton, Xiaobai Sun, and Benjamin Lee, thank for you for being patient and working through the many scheduling constraints. Ben, thank you for allowing me to turn to you for a different perspective on important decisions in my research career. I enjoyed our conversations and I really appreciate all the honest advice you gave me. Xiaobai, thank you for helping me work in Numerical Analysis. You patiently answered all my questions and gave invaluable assistance in finding benchmarks and setting up experiments. Drew, thank you for you feedback and guidance on my research.

To the graduate students in Hudson 213, thank you for the many interesting conversations that have helped me remain sane during submission deadlines. Blake, thank you for letting me bounce half-baked ideas off of you. Adam, thank for you being my gym and tennis partner. Marisabel, thank you for working with me on class projects. Luwa, I really enjoyed our many philosophical conversations. Weidan, thank you for being my office mate, especially during the stressful thesis writing season. Lastly, I
would like to thank the all students in the office, both past and present, for their support.
1. Introduction

Computers are used in many important applications including scientific simulations, financial models, and defense. These applications often have both accuracy and temporal requirements, i.e. it is important for these applications to get the right answer and to get it quickly.

Calculations performed on computers suffer from limited precision. Indeed, rounding errors may be introduced after each step in the calculation. Over the sequence of a long computation, this error can significantly affect the accuracy of the final result\(^1\).

This rounding error can sometimes have deadly consequences as in the case of the Patriot missile that accrued rounding error when computing the time since boot[1][2]. This rounding error caused the missile to miss an incoming Scud missile which struck a barracks killing twenty-eight people.

Modern architectures allow for tradeoffs to be made between accuracy and performance by changing the precision of the computations. Thus programmers can sacrifice some accuracy for performance and vice-versa. However, making these tradeoffs can be difficult due to the complexity of modern applications, as well as the

\(^1\) http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html
large number of calculations in the application. Ideally, we would want to design automated tools to allow novice programmers to make these decisions easily.

Making information about the accuracy and inner workings of floating point computations, available at the operation level, in excess of that currently visible can be used by either programmers, or automated tools, to improve the accuracy and/or performance of numerical applications.

In this work, we develop frameworks to allow for more efficient and accurate floating point computations. The common thread in the frameworks presented in this dissertation is to make information usually discarded in the hardware, specifically, in the floating point unit, visible to the programmer. The programmer can then use this information to improve the accuracy and performance of his/her applications. In addition, we develop tools that automatically use the information garnered from the frameworks to allow even novice programmers to make these complex accuracy and performance tradeoffs.

In Chapter 3, we introduce Recycled Error Bits (REBits). REBits makes the error of floating point operations, specifically additions, architecturally visible to the programmer. The programmer can use this to improve the accuracy of their applications. Alternatively, REBits can be used to improve performance without significantly affect the quality of the results. In addition, we develop a compiler pass to automatically instrument code to use REBits thereby freeing the programmer.
In Chapter 4, we present a framework for Profile Driven Mixed Precision Analysis. This framework determines the precision of all variables in an application based on their numerical behavior. We profile the application to obtain the floating point behavior of all computations. Variables with “bad” behavior are executed in higher precision while those with “good” behavior are executed in lower precision. Programmers can then use this to automatically make the tradeoffs between accuracy and performance.

In this thesis, we make the following contributions:

   
   We evaluate \textit{REBits} to show that it provides accuracy comparable to existing methods at a lower cost. We allow for programmers to automatically instrument their applications to benefit from \textit{REBits}.

2. We develop a numerical profiling framework to detect “bad” behavior at the floating point operation level. We provide a debugger to \textit{read} the profile to detect variables that have “bad” behavior. In addition, we design a tool to automatically increase the precision of variables with “bad” behavior.
2. Background

2.1 IEEE-754 Standard

Computers commonly represent real numbers using floating point numbers. Floating point numbers approximate real numbers with limited precision and range. The limitations on floating point arithmetic are due to the fixed size of floating point numbers in computers. The IEEE-754 standard [3] specifies the size of floating point numbers, the operations that can be performed, as well as the rounding modes associated with those operations. In this section, we present a brief overview of the IEEE-754 standard; for a more complete review, we refer readers to Goldberg [4].

IEEE-754 floating point numbers consist of a sign bit, an exponent field, and a mantissa field as shown in Figure 1. The mantissa field consists of an implicit one before the stored leading one. The IEEE-754 2008 standard specifies the allowable sizes of the exponent and the mantissa fields. The total floating point sizes are half precision (16-bits), single precision (32-bits), double precision (64-bits), and extended precision (128-bits). The sizes of the exponent and mantissa fields are given in Table 1. The value given by the number is

\[ value = -1^{sign} \times 2^{exponent} \times mantissa \]

Increasing the size of the floating point number increases the size of both the exponent, which improves the range, and the mantissa, which improves the precision.
Because floating point numbers are an approximation of real numbers and have limited precision, operations on floating point numbers may involve rounding. During rounding, error is introduced into the calculation. Increasing the precision of the floating point numbers reduces the rounding error for each operation thereby allowing the calculation to obtain better accuracy.

In this thesis, we consider only binary 32-bit and 64-bit IEEE-754 floating point numbers as they are currently most commonly implemented in hardware.
2.2 **Hardware Support**

Though the IEEE-754 standard allows four precisions, current CPUs, particularly x86 CPUs, implement only single and double precision natively in hardware. While other architectures offer the half and extended precision, in these works, we limit the discussion to single and double precisions. x86 currently emulates half and extended precision in software. In addition to the floating point precisions specified in the IEEE-754 standard, x86 implements the x87 floating point extension. x87 is an 80-bit non-IEEE-754 compliant floating point format.

Modern architectures, both CPUs and GPUs, have vastly different performance characteristics for applications in single and double precisions. A look at datasheets for the Intel Datasheets show a 2X performance difference for CPUs [5] and greater than 2X for the latest GPUs\(^1\) [6]. Although, we focus on CPUs in this thesis, we note that similar analyses can also be performed for GPUs.

The main reason for the disparity in single and double precision throughputs is bandwidth, both compute and memory. We divide compute-bound applications into two cases; vectorized and scalar. Fully vectorized code uses the all throughput offered by the hardware by utilizing the entire width of the Single Instruction Multiple Data (SIMD) lane. Instructions operate in lockstep on all lanes in a SIMD processors. For such

---

\(^1\) [http://www.nvidia.com/object/tesla-servers.html](http://www.nvidia.com/object/tesla-servers.html)
applications, single precision performs much better than double precision code because 2X more single precision FPUs can find an n-bit wide SIMD lane when compared to double precision FPUs. However, if the code is not vectorized, then single and double precision are expected to have similar performance characteristics for compute-bound applications. This lack of vectorization maybe due to limited data level parallelism in the algorithm or the inability of the programmer to optimize their code to maximize the utilization of the hardware. The reason scalar single and double precision compute bound applications have roughly the same performance is that the latency of most single precision floating point operations is the same as that of double precision operations\(^2\).

For memory bound applications, this analysis is slightly more complicated as it depends whether the application’s working dataset is cache resident or is streamed in from main memory. Cache resident applications may be limited by both bandwidth to the cache as well as the cache capacity. For cache resident applications, moving from single precision to double precision could cause the size working dataset to exceed the size of the cache and cause some of the data to be pushed into the next level in the cache hierarchy which would negatively affect performance. In the worst scenario, moving to double precision could cause the working set to spill out of the L3 and into main

\(\text{\(^2\) For most common operations (add/sub/mul) this statement is true. Obviously some of the more complicated floating point operations may exhibit lower cycle counts for single precision operations.}\)
memory. Streaming applications are primarily limited by bandwidth, not storage. For bandwidth limited applications, either those that are cache bandwidth limited or stream from main memory, 2X more single precision values can be transferred per unit time than for double precision values. As such, single precision is expected to perform significantly better when compared to double precision for streaming applications.

The above analysis holds true for direct methods but not for iterative methods. Iterative applications, such as the Conjugate Gradient method to solve a system of linear equations, may converge faster, with fewer iterations, with double precision instead of single precision. As such, the performance implications of precision are harder to quantify.

Thus, for vectorised applications, double precision is significantly slower than single precision while offering higher precision. Programmers then have the ability to trade-off performance for accuracy.

2.3 Software Libraries to Improve Precision

Sometimes the maximum precision provided by the hardware is insufficient, leading to the final result being inaccurate. In this case, programmers can use numerical libraries that provided higher precision than that provided natively in hardware. We divide these numerical libraries into two broad classes; ones that use the hardware floating point unit and ones that are implemented entirely in software.
2.3.1 Double-Double Arithmetic

One can increase precision by using software to stitch together multiple floating point values and computations. A notable example is the Double-Double arithmetic developed by Bailey et al. [7]. A double-double floating point value is represented by two doubles, and the value of the double-double is the sum of these two doubles. If A and B are both double-doubles (pairs of 64-bit values), then adding A+B produces a double-double C. The challenge is that adding two double-doubles requires more than two hardware floating point additions. In fact, adding two double-doubles requires 20 64-bit hardware additions! Thus Double-Double arithmetic requires substantially more work and thus takes more time. Natural extensions of Double-Double, such as Quad-Double, extend the accuracy/cost trade-off. We show later, in Section 3.6.2, how to use REBits to accelerate Double-Double operations and make them more energy-efficient.

2.3.2 Simulating Arbitrary-Precision FPU

An extreme way to achieve accuracy is to simulate an arbitrary-precision FPU and not use the floating point hardware at all. A well-known GNU library, called GMP (GNU Multiple Precision: www.gmplib.org), uses integer and bit-manipulation instructions to perform arbitrary-precision floating point computations. GMP thus provides the desired accuracy, but at an extremely large cost in terms of time and energy.
2.4 **Changing Algorithms to Improve Accuracy**

There are several well-known problems that have multiple mathematically equivalent, but numerically distinct, algorithms that may be used to solve the problem. One well-known example of such a problem is the generation of an orthonormal set of basis vectors in an inner product space. The Gram-Schmidt process, which is used to generate these basis vectors, suffers from significant rounding errors causing the resulting vectors to not be orthonormal. However, a modification of the algorithm leads to the Modified Gram-Schmidt process that is mathematically equivalent, but suffers from less rounding errors.

Another example is summation of floating point numbers which is a very common operation in scientific computing in general or linear algebra. Summation suffers from poor accuracy because rounding error accumulates easily due to the high dynamic range and the oscillatory nature of the input data. Thus, there exists a large amount of work that attempts to improve the result of summations. Algorithms by Dekker [8], Kahan [9], Knuth [10], and Priest [11] improve the results of numerical summations.

Ansel et al. develop Petabricks [12], a compiler framework that tunes applications to tradeoff accuracy and performance. Petabricks allows programmers to make algorithmic choices while meeting accuracy thresholds.
2.5 Detecting Numerical Errors

Detecting numerical errors is a well-studied problem, and as such, there exists a large amount of work that attempts to solve it. These works can be broadly divided into static analysis and dynamic analysis. One example of a static analysis work is [13] which measures the error accrued by using limited precision instead of using real numbers on all possible values on the nodes of an application without executing it. In this work, we focus on dynamic analyses methods since floating point error accumulation is input dependent.

One form of detecting errors dynamically is interval arithmetic [14][15][16]. Interval arithmetic attempts to establish a maximum bound on the error by performing computations on intervals instead of a single number. Thus, the result of a calculation is not just one number but two; the lower bound and the upper bound of the interval guaranteed to contain the result. After each calculation, the size of the interval increases to account for any rounding error accrued in that operation. While the interval contains the correct result,\(^3\) it can be arbitrarily large and not convey much information. For example, if the result of a computation is the interval \([-\infty, \infty]\), this interval contains the correct answer, it does not provide any useful information. This is because interval arithmetic measures the worst-case rounding error which might be significantly larger

\(^3\) Infinite Result – result computed at infinite precision
than the errors seen with typical datasets. In addition to interval growth, interval arithmetic also suffers from significant performance penalties. However, there exist hardware accelerators [17] that attempt to mitigate the performance impact.

Affine arithmetic [18][19][20][21] represents a number as a linear combination of intervals between negative one and positive one. It is an improvement over interval arithmetic because it corrects the interval growth in interval arithmetic. The interval in affine arithmetic accounts for the rounding error in the computations that lead to that result. For more details, we refer readers to [22].

2.6 Prior Work in Mixed Precision

A large amount of prior work has explored the use of Mixed Precision – selectively assigning certain variables to be in double precision and others to be in single precision – including works by Baboulin et al. [23], Buttari et. al. [24], Buttari et. al. [25], Hogg and Scott [26], Li et al. [27], Strzodka and Goeddeke [28]. In these works, the authors develop mixed precision routines, intended primarily for linear solvers. This work has been applied to other fields like flow problems [29], finite element methods [30] among others. While in Chapter 4, we focus on applications run on CPUs, prior work has explored the implications of mixed precision of GPUs [31][32][33][34][35]. Prior works are not necessarily limited to software only as Strozka [36] develops an FPGA implementation of a mixed precision PDE solver.
For some applications, primarily graphics, lower precision than that in the IEEE-754 standard, implemented using fixed point, can have significant speedups as shown by Hao et. al. [37]. Jenkins et. al. [38] develop a scheme to split up floating point numbers along byte boundaries Their scheme allows programmers to trim the mantissa of floating point numbers to reduce the memory and bandwidths impacts.
3. **Recycled Error Bits**

3.1 *Introduction*

Developers of numerical software must consider the finite precision of floating point hardware in processor cores and the potential for small inaccuracies to snowball into glaring—and silent—inaccuracies over the course of a long sequence of computations [39]. This is because the small rounding error inherent in each operation can accumulate and affect the accuracy of the final result.

In this chapter, we propose extending the floating point unit to provide the programmer with the rounding error for each operation. The programmer can then use this rounding error to improve the accuracy of their applications. We call our idea *Recycled Error Bits (REBits)*.

REBits makes the following contributions:

- We discuss the design space of options for making the rounding error architecturally visible, and we highlight the option that is easiest to implement in a modern core.

- We show how to write software that uses the rounding error in running sums (very common idioms in numerical software) and to accelerate *Double-Double* [7].

- We develop compiler support for *REBits* that automatically transforms software to take advantage of the rounding error.
We perform an experimental evaluation of REBits on a real hardware platform. This is the first evaluation of any scheme that provides the rounding error to software. At the 32-bit granularity, REBits achieves accuracy that is comparable to 64-bit precision with performance and energy that are superior to MaxHW (64-bit) and MixedHW for vectorized benchmarks. At the 64-bit granularity, REBits achieves accuracy that is comparable to Double-Double [7], with performance and energy that are vastly better than Double-Double.

While the idea of REBits applies to all floating point operations, we focus exclusively on addition in this paper. Software that sums a large number of values—which is a common feature in scientific code—is particularly vulnerable to accumulated rounding error. We specifically address the error that can accumulate in such sums. We consider only addition because a running sum of computed products (or quotients or numbers computed otherwise) can be considered a running sum of numbers that are given. Another reason to consider only addition is the fact that, for rounding error to be “significant,” in most cases a long computational chain (e.g., a running sum) needs to exist.

We motivate the need for REBits over existing work in Section 3.2. In Section 3.3, we present our new architectural support for low-cost extended precision floating point math and review how to extend floating point hardware to provide the rounding error in each addition. We next show how to develop software that utilizes REBits in Section
3.4. In Section 3.5, we experimentally evaluate the accuracy, energy-efficiency, and performance of REBits, in comparison to existing approaches. Lastly, we compare REBits against well-known software-only schemes to improve accuracy and show how REBits can be used to improve the performance and energy-efficiency of one such scheme in Section 3.6.

3.2 Motivation

There are many traditional approaches that numerical software developers use for handling this well-known problem. In this section, we briefly summarize the prior methods to motivate the use of REBits.

**Maximum Hardware Precision:** The simplest approach to maintaining accuracy while dealing with hardware’s finite precision is to use as much of that precision as possible. If the hardware supports 32-bit and 64-bit floating point, then a programmer can choose to *always* use the 64-bit hardware by declaring every variable as a 64-bit double-precision *double* instead of a 32-bit single-precision *float*. The primary drawbacks to this approach, which we refer to as *MaxHW*, are its energy and performance overheads. Many general-purpose processors—including processors from Intel, AMD, and IBM—provide significantly greater throughput for 32-bit precision than
64-bit precision [40]. GPUs also tend to provide greater throughput for 32-bit precision than for 64-bit precision.\(^1\)

**Mixed Hardware Precision:** A clever programmer, possibly with automated help [41][42], can selectively choose which precision to use for each variable [23][43]. This approach, which we refer to as MixedHW, reduces many of the costs of MaxHW, but requires a more sophisticated numerical analysis and still incurs significant performance and energy overheads due to vectorization.

With MixedHW, the programmer declares most variables to be 32-bit floats except for some variables (e.g., running sums) that are selectively declared as 64-bit doubles. The MixedHW approach can potentially achieve the “best of both worlds,” if the programmer can effectively identify which variables should be 32-bit floats and which should be 64-bit doubles [23][43]. Automation can help the programmer in this process [41][42]. On scalar (non-vectorized) software, we have observed that MixedHW does indeed achieve the desired accuracy at low overheads. However, for vectorized software, which comprises the majority of numerical software, MixedHW suffers significant performance and energy overheads. As illustrated in Figure 2, the culprits are the additional casts (from float to double) and packing/unpacking of vectors. A simple

\(^1\) http://www.nvidia.com/object/tesla-servers.html
single precision implementation of the example code would include just the vector load and the vector add.

**Emulating Greater Precision with Software:** When even MaxHW provides insufficient precision, programmers must resort to software to emulate greater precision than available in the hardware floating point units; however, this emulation incurs steep performance and energy costs.

All of these traditional techniques have significant drawbacks, and they have inspired the observation, made by two recent papers [44][45], that a software developer could benefit from observing the rounding error that occurs in each operation. Both of these papers show that a core’s floating point unit (FPU) can be extended, at low cost in hardware and power, to provide not just the result but also the rounding error. DeHon [44] evaluates how, in theory, using the rounding error could accelerate parallel summations. Dieter et al. [45] present a way in which the rounding error could be made architecturally visible, and they claim—correctly, but without evidence—that one could use the rounding error to accelerate a specific software scheme called Double-Double [7]. These papers highlight a potential opportunity, but they lack architectural analysis, software to exploit the rounding error, and experimental evaluations.
Figure 2. MixedHW operations required for vectorized software
3.3 Recycled Error Bits

In this section, we present our architectural support for numerical accuracy, called Recycled Error Bits (REBits).

3.3.1 Big Picture

The key idea of REBits is to make the rounding error in each floating point operation architecturally visible, such that it can be used by the programmer. Numerical analysts have long known that having the rounding error is useful, which is why many software schemes try to infer the rounding error and why two recent papers proposed floating point unit hardware extensions to compute the rounding error [44][45].

In today’s cores, the FPU discards information when it rounds, and our goal is to recycle this information to help programmers achieve greater accuracy when desired. We illustrate this high-level view of REBits in Figure 3.

Floating point arithmetic has some edge cases such as infinity and Not-A-Number (NaN). These situations do not directly affect REBits, because the rounding
error can never be infinity or NaN. If the answer is infinity or NaN, the error does not matter. We do not currently handle sub-normal numbers (“denorms”).

### 3.3.2 Architecture

*REBits* makes the rounding error of each floating point addition *architecturally* visible as an IEEE-754-compliant floating point number. There are three ways we considered doing this, and we settled on the last option:

**Dedicated Register:** As proposed by Dieter et al. [45], we could have every floating point add instruction (fpadd) write the error to a dedicated register (in addition to writing the sum, as usual, to an architectural register specified in the instruction). We call this register FPERR, and its value is overwritten by every fpadd. Thus, a programmer who wishes to use this information must move it from FPERR to a regular floating point register before the next fpadd instruction. The drawback to this design is the implicitly written FPERR register that needs to be renamed and bypassed. Dieter et al. also suggested a stack of FPERR registers, but that approach has additional complexities.

**New Add Instruction:** We could add an fpadderr instruction whose result is the rounding error that results from adding the two input operands. The drawback to this design is that it requires two extra arithmetic instructions resulting in a significant performance degradation.
**New Fused AddError Instruction:** We add a new instruction that is analogous to a fused multiply-add (FMA) and that we call fused generate-error-and-accumulate (FGEA). This FGEA instruction accepts three operands, produces the rounding error from adding the first two operands and adds this rounding error to the third operand. We believe this architectural interface is preferable to the other two options above.

### 3.3.3 FGEA Implementation

*REBits* extends the functionality of the floating point adder such that it recycles the error that is normally discarded. Recycling is an apt analogy, because most of the logic required to determine the error is already in the FPU and we simply need to maintain some bits that are otherwise ignored.

We propose modifying the floating point adder to produce both the result and the accumulated error. The baseline adder has two inputs, $a$ and $b$, and produces the result $s$. As the FGEA instruction has three inputs, we add another input, $e$, which is the accumulated error. The FGEA hardware will produce the error of adding $a$ and $b$ to produce the intermediate error, $e_i$. This intermediate error is then added to the input error, $e$, to produce the result which is the accumulated error. Both the FPADD and the FGEA instructions can use the same hardware, with the FPADD instruction being able to complete somewhat more quickly.
3.3.3.1 Rounding Errors Produced by Hardware

Floating point addition is performed in stages. Error is introduced only in the alignment and rounding stages, while the remaining stages are lossless. In the two stages where error is introduced, we track the error using our hardware modifications. In this section, we assume that we are adding two numbers, $A$ and $B$. For ease of explanation, we assume that $A$ is greater than $B$ and that both numbers are positive.

Error is produced when we denormalize the smaller number, $B$ in this case, in order to align its exponent $A$. In the process of aligning the exponents, $B$’s mantissa is shifted to the right. The second step in which error can be produced is during the final rounding step. It is in this step that the intermediate result’s mantissa (which can be a few bits larger than the rounded mantissa) is rounded to be the size specified by the IEEE-754 standard.

We need to make some modifications to the floating point unit in order to capture the error and convert it into an IEEE-754 compliant floating point number. The bits that are shifted out of the mantissa of $B$ in the process of aligning the mantissas of $A$ and $B$ are saved as $B'$. During the rounding stage, we save the lost bits $R'$. Using $B'$ and $R'$, we can construct the mantissa of the error. We then normalize it and use the information from the normalization and the mantissa of the result to reconstruct the exponent of the error. The sign on the error is dependent on the sign of the result and whether or not the floating point unit rounded the intermediate value up or down in the
rounding stage. This is similar to work done by Kadric et al. in [44]. Kadric et al. modify a floating point adder to produce the additive error, as well as the result of the addition.

### 3.4 Utilizing REBits in Software

The goal of this work is to provide low-cost architectural support for extending the precision of floating point. So far we have described the architectural support, and now we turn our attention to how software uses it. In Section 3.4.1 we present a software interface to REBits for use by advanced programmers with an example provided in Section 3.4.2. In Section 3.4.3, we present compiler support for REBits to automatically instrument code.

#### 3.4.1 Programming Interface

There are several ways in which we could make the rounding error visible to an application programmer. Although it is possible to provide access via a library call in a HLL (High Level Language), most numerical software developers are willing and able to use lower-level interfaces, including inline assembly code and compiler intrinsics. In this paper, we assume compiler intrinsics, but other choices are viable. The intrinsic _mm_getErr_ss returns the rounding error of performing the addition of the two operations. If this intrinsic is used in isolation (i.e., not used to accumulate the rounding error), the third operand of the FGEA instruction is set to zero. (Recall that FGEA adds the rounding error of the addition of the first two operands to the third operand.)
3.4.2 Example REBits Software

We now illustrate how we use REBits in actual software, using a simple example: a function that computes the 2-norm of a vector of numbers:

\[ 2\text{norm}(\vec{V}) = \sqrt{\sum_i V_i^2} \]

In Figure 4, we show four implementations of 2-norm code. In the first row, from left to right, we present the accuracy-unaware HLL code and then the HLL code with REBits. In the second row, we present the accuracy-unaware assembly and the REBits assembly. (For the assembly, we show only the instructions within the for-loop.) The REBits code requires minimal modification to the accuracy-unaware code, and understanding this modification does not require complex numerical analysis. However, it does require the use of either assembly or compiler intrinsics. In the next section, we present compiler support for REBits that would enable it to be used by even novice programmers.
```c
float result = 0;
float[N] v;
for (i=0; i<N; i++){
    result = result + v[i] * v[i];
}
result = sqrt(result);
```

```c
float result = 0;
float err = 0;
float[N] v;
for (i=0; i<N; i++){
    float t = v[i] * v[i];
    err = err + _mm_getErr_ss(result, t);
    result = result + v[i] * v[i];
}
result = sqrt(result + err);
```

<table>
<thead>
<tr>
<th>HLL Naïve accuracy-unaware code</th>
<th>HLL REBits code</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmovss (%rsp,%rax,4),%xmm1</td>
<td>vmovss (%rsp,%rax,4),%xmm1</td>
</tr>
<tr>
<td>incq %rax</td>
<td>incq %rax</td>
</tr>
<tr>
<td>vmulss %xmm1,%xmm1,%xmm2</td>
<td>vmulss %xmm1,%xmm1,%xmm2</td>
</tr>
<tr>
<td>vaddss %xmm2,%xmm0,%xmm0</td>
<td>vfgadd %xmm3,%xmm2,%xmm0</td>
</tr>
<tr>
<td>cmpq $10, %rax</td>
<td>// %xmm3 = %xmm3 + Err(%xmm2+%xmm0)</td>
</tr>
<tr>
<td>j1 ..B1.4</td>
<td>vaddss %xmm2,%xmm0,%xmm0</td>
</tr>
<tr>
<td></td>
<td>cmpq $10, %rax</td>
</tr>
<tr>
<td></td>
<td>j1 ..B1.4</td>
</tr>
</tbody>
</table>

**Figure 4.** 2-Norm Software. REBits extensions highlighted in bold text.

### 3.4.3 Compiler Support

In Section 3.4.1, we presented some ways that REBits could be accessed and used by high level language programmers. However, the approaches presented in Section 3.4.1 require programmers to modify their applications by hand and to explicitly consider floating point accuracy. We seek to automate the insertion of REBits such that even novice programmers can easily utilize it for running sums using loops. We design a compiler pass that identifies running sums and instruments the code to insert instructions to use REBits. We use the LLVM (Low Level Virtual Machine) framework [46] developed at the University of Illinois. LLVM provides multiple front-ends and
allows REBits to work with different languages. We have evaluated our compiler pass with C, C++, and FORTRAN applications.

Our compiler pass operates at the Intermediate Representation (IR) level. LLVM IR is Static Single Assignment (SSA) based and offers a clean interface between assembly and the HLL. The compiler pass identifies running sums in loops and insert instructions for REBits. We define running sums as follows: a variable \( V \) is a running sum in a particular loop \( L \) if and only if the definition of \( V \) in \( L \) is of the form \( V = V + \text{expr} \) (and there is at least one definition of \( V \) in \( L \), and the only uses of \( V \) in \( L \) are in the statements that re-define it. In SSA semantics, we detect a running sum if there any exists a \( \phi \) node that dominates a floating point add with a back-edge from the floating point add to the \( \phi \) node. We use the following heuristic to identify running sums given our definition:

1. An input to a floating point \( \phi \) node is used before it is defined. A \( \phi \) node is an IR-only construct that is used to select a value based on the basic block that preceded the \( \phi \) node.

2. The floating point value is produced by an add instruction.

These conditions ensure that reductions involving loops that are running sums are selected for instrumentation. If a variable is used in a \( \phi \) node part of a loop and is produced by an add instruction in that loop, it is a running sum.

Once the compiler pass has identified, using the above conditions, a running sum for potential REBits instrumentation, it inserts an FGEA instruction before the add
instruction. We use an example running sum in the top of Figure 5 to illustrate the instrumentation process. The middle row in Figure 5 demonstrates the insertion of an FGEA instruction. The extra LLVM IR instructions added for \textit{REBits} are bolded. The inputs to the FGEA instruction are the inputs to the add instruction as well as the running error. Also, the compiler pass ensures that the running error computation path has the same control and data flow as the original running sum. After the for-loop completes, the running error is added to the running sum to form the higher accuracy running sum (\texttt{\%temp} in bottom right of Figure 5). All future uses of the original running sum are replaced with this new value.
double sum = 0;
for (int i = 0; i < N; i++){
    sum += v[i];
}
printf("\%f\", sum);

<table>
<thead>
<tr>
<th>HLL Naïve accuracy-unaware code</th>
</tr>
</thead>
<tbody>
<tr>
<td>%sum.phi = phi double 0.0, %add</td>
</tr>
<tr>
<td>%3 = load double* %arrayidx</td>
</tr>
<tr>
<td>%add = fadd double %sum.phi, %3</td>
</tr>
<tr>
<td>%sum.phi = phi double 0.0, %add</td>
</tr>
<tr>
<td>%add = fadd double %sum.phi, %3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Naïve Simplified LLVM IR for loop body</th>
<th>Simplified LLVM IR with \textit{REBits} for loop body</th>
</tr>
</thead>
<tbody>
<tr>
<td>%sum.lcssa = phi double 0.0, %add</td>
<td></td>
</tr>
<tr>
<td>%call = tail call @printf(%sum.lcssa)</td>
<td></td>
</tr>
<tr>
<td>%call = tail call @printf(%sum.lcssa)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Naïve Simplified LLVM IR for end of loop</th>
<th>Simplified LLVM IR with \textit{REBits} for end of loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>%call = tail call @printf(%sum.lcssa)</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Figure 5: Simplified LLVM IR before and after \textit{REBits} Instrumentation'\hspace{1cm}}

29
3.5 Experimental Evaluation

We now experimentally evaluate REBits, both on kernels and then on complete benchmarks. The summation and the dot-product kernels that we use for evaluation here are the central parts of the vast majority of HPC and numerical applications [47][48]. Hence, improving the performance and accuracy of these kernels, especially dot product, is essential in improving the efficiency and accuracy of complete HPC benchmarks. We also show the effect of using REBits in three complete programs that are typical, commonly-used numerical software: LU Factorization, N-body Simulation, and Monte Carlo Simulation.

3.5.1 Methodology

We evaluate REBits in comparison to uniform hardware precision using the native FPU (denoted Native), MixedHW, and Double-Double. In Table 2, we list all schemes evaluated and whether we use them for benchmarks with 32-bit or 64-bit inputs. We also evaluate against legacy x87 (discussed in Section 2). Some benchmarks have two versions, one with 32-bit inputs and one with 64-bit inputs, but we have no benchmark that takes both 32-bit and 64-bit inputs in the same experiment. When using Native-64 on 32-bit inputs, we cast the final result, just before printing, to 32-bits to enable fair comparisons to 32-bit schemes.
Table 2. Schemes Compared

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Description</th>
<th>Benchmark Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native-32</td>
<td>all 32-bit floats</td>
<td>32-bit</td>
</tr>
<tr>
<td>Native-64</td>
<td>all 64-bit doubles</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>MixedHW</td>
<td>mix of floats and doubles</td>
<td>32-bit</td>
</tr>
<tr>
<td>x87</td>
<td>Intel x87 (not vectorizable)</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Dbl-Dbl</td>
<td>Double-Double (Section 2.3.1)</td>
<td>64-bit</td>
</tr>
<tr>
<td>REBits-32</td>
<td>32-bit floats with 32-bit rounding errors</td>
<td>32-bit</td>
</tr>
<tr>
<td>REBits-64</td>
<td>64-bit floats with 64-bit rounding errors</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

**Experimental Platform:** We run all experiments on actual (not simulated) SandyBridge-E x86 hardware, using only the Intel® AVX (Advanced Vector eXtensions) units (except when explicitly evaluating x87). The system is an Intel® Xeon® E5-2680 with two sockets and 8 cores in each socket. The cores run at 2.70 GHz. Each chip has 20MB of L3 cache, and the system has 64GB of DDR3 DRAM. The system has an idle power consumption of 12W measured at the socket using Intel® RAPL (Running Average Power Limit) counters. The system runs Linux 3.0.13-0.27-default. We run each
experiment 5 times to account for the possibility of variability across experiments, although, in practice, we observe much less than 1% variation across experiments.

**Compilation/Vectorization:** We use the LLVM software pass presented in Section 3.4.3 to identify and instrument running sums within our benchmarks. To get the maximum benefit from the vectorization, we performed the vectorization by hand on both baseline and proposed codes with compiler intrinsics using icc with the “-O2” optimization level. We also use “–fp-model precise” to ensure that the compiler does not make optimizations that would affect the result of floating point computations. Because AVX offers 256-bit SIMD registers, we have 8-wide vectorization for 32-bit computations and 4-wide vectorization for 64-bit computations.

**Accuracy:** We compare the numerical results for existing hardware/software and 
REBits to numerical results produced by the arbitrary-precision GMP library using 1000 bits of precision. GMP-1000 is our “gold standard” of accuracy. To simulate the REBits FPU, we use the open-source SoftFloat\(^2\) software simulator of an IEEE-754 compliant FPU. The FPU’s adder supports all rounding modes supported by IEEE-754. We modified the SoftFloat FPU’s adder to produce the rounding error.

**Performance:** We measure runtimes using a wall clock. For evaluating the runtime of REBits code, we must consider the latency required for performing the FGEA

\(^2\) http://www.jhauser.us/arithmetic/SoftFloat.html
instruction, yet our actual hardware does not have the FGEA instruction. We mimic the latency and dataflow dependence of the FGEA by inserting an assembly instruction, specifically floating point multiply (FMUL), to mimic the performance of FGEA such that the functionality of the program is not affected and the dataflow is preserved. In fact, FMUL conservatively over-estimates the latency and energy consumption of FGEA.

**Energy:** We measure power using the RAPL on-chip performance and power counters on the chip. We use the Intel® PCM (Performance Counter Monitor³) tool that interfaces with RAPL to measure the amount of energy required to run a benchmark. As with the performance experiments, we mimic the FGEA instruction using FMUL, which approximates the energy consumption of the FGEA that would occur in REBits.

### 3.5.2 Kernel Microbenchmarks

We analyze two kernels—summation and dot-product—that are ubiquitous in HPC and numerical benchmarks [47][48].

#### 3.5.2.1 Summation of Long Sequence of Numbers

We perform two different summation experiments: one with randomly chosen numbers and one with data from a real scientific experiment.

```c
float sum=0, err=0;
```

float[N] v;
for i=0; i<N; i++){
    err = err + _mm_getErr_ss(sum, v[i]);
    sum = sum + v[i];
}
result = result + err;

Figure 6. REBits-32 Code for Sum of Random Numbers

Summation of Random Numbers

We perform the summation of 1048576 ($2^{20}$) random floating point numbers sampled from a uniform distribution between -1x10^6 and +1x10^6. Each experiment consists of a large number (100,000) of computations of the sum, to minimize the effects of initialization. We perform each experiment multiple times with different random seeds, and we observe negligible differences between experiments.

For REBits-32, we use the algorithm in Figure 6. For REBits-64, simply replace floats with doubles. We experiment with both scalar and vectorized code. Although the focus of this paper is on numerical applications, which are generally vectorizable, we want to compare how the schemes perform on scalar code for one benchmark. The comparison reveals the impact of vectorization on both REBits as well as MixedHW. A scalar benchmark also allows us to compare to x87 which is not vectorizable. (All other benchmarks in this paper are typical, vectorized applications.)

In Figure 7 and Figure 8, we plot the results summation with 32-bit and 64-bit inputs respectively. Each figure contains data for both scalar and vectorized code. In
each graph, there are two y-axes. Accuracy is on the right y-axis, measured in the
number of correct significant digits in the result (i.e., how many digits match GMP-
1000). The left y-axis is cost as measured in energy*delay^2 (ED^2), and it is normalized to
Native-32 and Native-64 for 32-bit inputs and 64-bit inputs, respectively.

The results provide several insights:

- For 32-bit inputs and scalar code, MixedHW, REBits-32, Native-64, and x87 achieve
  the same accuracy. However, REBits-32 achieves this accuracy at a much higher cost
  than the other schemes because the extra latency due to FGEA (emulated by FMUL)
  cannot be hidden in this simple loop body. We show later that the latency of FGEA
  can be hidden in loops that perform more work per iteration.

- For 32-bit inputs and vectorized code, REBits-32 achieves better accuracy than
  MixedHW at approximately the same cost. This experiment quantifies the cost of
  vectorizing MixedHW. However, Native-64 is still preferable to REBits-32 (better
  accuracy at similar cost); as with scalar summation, the cost of REBits-32 is high in
  this tight loop, which is why its cost approaches that of Native-64.

- For 64-bit inputs and scalar code, REBits-64 suffers for the same reasons as REBits-32.
  The x87 approach, which is not vectorizable, does best in this experiment.

- For 64-bit inputs and vectorized code, REBits-64 achieves the same accuracy as
  Double-Double at less than 2% of the cost. REBits-64 provides three more significant
digits than Native-64 at only 2.5 times the cost.
A simple running sum with no other computation in the loop is the worst-case scenario for REBits, in terms of its cost, and these results highlight that cost. In all other benchmarks, the cost of REBits can be more easily overlapped with the other computation done in the loop.
Figure 7. Results: Summation of Random Numbers (32-bit Inputs)

- **Vector Summation (32-bit Inputs)**

- **Scalar Summation (64-bit Inputs)**
Figure 8: Summation of Random Numbers (64-bit Inputs)

Summation of Sea Heights

A prior paper [49] on numerical accuracy in scientific simulations, focusing on the problem of summations, presented a stress-test kernel that sums a remarkably small, yet problematic, 2D (120x64) array of 64-bit sea heights. This kernel is similar to the one in the previous section, except the data is real (not randomly generated). The authors observed that the answer differed greatly depending on the order in which the sea heights were summed (e.g., row-first vs. column-first). In our experiment, we perform the summation one million times to minimize the effects of initialization (i.e., declaring the array of sea heights, reading the array entries in from a file, etc.).

In Table 3, we present the experimental results. The “Sea Height” column illustrates the accuracy issues for this kernel. The Native-64 results indeed vary wildly
based on the order of summation, and none of the summation orders lead to correct results. Some Native-64 results are incorrect by two orders of magnitude. REBits-64, however, obtains the same result in all orders of summation, and this result is the same as GMP-1000. REBits-64 is as accurate as Double-Double, but it is more than two orders of magnitude less costly.

Table 3. Results: Sea Heights

<table>
<thead>
<tr>
<th></th>
<th>Sea Height</th>
<th>normalized ED²</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Native-64</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row-first</td>
<td>34.418</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>32.3027</td>
<td>1.0</td>
</tr>
<tr>
<td>Col-first</td>
<td>0.48759</td>
<td>1.06</td>
</tr>
<tr>
<td>Reverse-col-first</td>
<td>0.16016</td>
<td>1.07</td>
</tr>
<tr>
<td><strong>REBits-64</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all directions</td>
<td>0.35799</td>
<td>4.53 row-first</td>
</tr>
<tr>
<td><strong>Double-Double</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all directions</td>
<td>0.35799</td>
<td>396.42 row-first</td>
</tr>
<tr>
<td><strong>GMP-1000</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.35799</td>
<td></td>
</tr>
</tbody>
</table>

3.5.2.2 Dot-Product

In this kernel, we compute the dot-product of two long vectors. The results in Figure 9 (left side is 32-bit inputs, right side is 64-bit inputs) show that REBits-32 is...
strictly preferable to MixedHW. *REBits*-32 is still one significant digit short of *Native*-64, but its cost is much less—in fact, its cost is almost the same as *Native*-32. This kernel is different from summation because the inner loop is more complicated and allows for the extra latency of FGEA, emulated by FMUL, to be hidden. The results also show that *REBits*-64 has as many significant digits of accuracy as *Double-Double* but at a small fraction of the cost.

The key takeaway point from these results is that *REBits* offers significant advantages for the experiments with 64-bit vectorized code for simple summation and *REBits*-32 can offer significant advantages for applications with more complicated loop bodies. The following section further motivates *REBits*-32.
3.5.3 Benchmarks

We identified several important, commonly-used scientific benchmarks that perform extensive summations.
3.5.3.1 LU Decomposition

LU matrix decomposition is a well-known, common numerical task, and we use LU software from NIST(http://math.nist.gov/tnt). We plot our results for LU in Figure 10. For this benchmark, \textit{REBits}-32 is more accurate than \textit{MixedHW} while offering significant performance and energy benefits. One might expect \textit{RE Bits}-32 and \textit{MixedHW} to always achieve the same accuracy, but there are situations in which they differ. In particular, because \textit{RE Bits}-32 uses two distinct floats to represent a sum, it achieves greater accuracy when representing a sum with a wide gap between its most significant and least significant bits (e.g., \textit{very long string of 0s}1001). Conversely, \textit{MixedHW} may achieve greater accuracy when there is no gap.
3.5.3.2 N-Body Simulation of Electrical Charges

Many scientific applications are examples of N-body simulations, where bodies can be molecules, planets, etc. An example well-known to computer architects is barnes-
hut, a benchmark in the Splash-2 benchmark suite [50]. We use a classic N-body simulation in which each body has a mass and an initial position and we calculate the next positions for all the bodies based on the interactions of the forces. All inputs are 32-bit in this experiment.

The results in Figure 11 show the trade-off between cost and accuracy. Because the result of an N-body simulation is the position of all N bodies, we present both the average-case (across all N) accuracy and worst-case accuracy. The results show that REBits-32 achieves the same accuracy as MixedHW but at lower cost. REBits-32 achieves the same average accuracy as Native-64 but one digit less of worst-case accuracy. The improvement in ED$^2$ for REBits-32 is caused by the improved scheduling due to the addition of the FMUL instruction that is used to emulate the FGEA instruction.

![Figure 11. Results: N-Body simulation](image.png)
3.5.3.3 Financial Pricing Simulation

Many financial applications rely on floating point arithmetic to predict prices for stocks, bonds, derivatives, options, etc. Accuracy is a paramount concern, and even small inaccuracies can get magnified when large sums of money are involved. Even a 0.01% error in pricing can be disastrous when speculating on billions of dollars worth of securities. The particular benchmark we study here is a proprietary Monte Carlo simulation for predicting European derivative pricing. All inputs are 32-bit.

As with the N-body simulation, the Monte Carlo simulation produces a vector of results, and we present both the average-case and worst-case accuracy results. The results, shown in Figure 12, are similar to those of the N-body simulation, but with one notable difference. REBits-32 has a worst-case accuracy that is two significant digits less than Native-64, as opposed to the one-bit difference for N-body simulation.
3.6 REBits vs. Software Numerical Methods

3.6.1 Comparison to Software-Only Schemes

There exists a large body of prior work that uses software to improve the accuracy (not precision) of summation in the presence of finite-precision hardware. Prior work—including notable contributions by Dekker [51], Knuth [52], Kahan [53], Bailey [7], Priest [54], and Shewchuck [55]—uses additional arithmetic operations to recover the rounding errors. Retrieving the exact error takes more operations, thus incurring more overhead, which is why some schemes choose to retrieve an approximate error at lower cost. REBits can retrieve the exact error automatically in hardware.
These software schemes also tend to be non-adaptive, treating all operations as potentially worst-case (i.e., with severe cancellations and loss of significant bits). Compared to these software techniques, \textit{REBits} uses a small amount of hardware to retrieve the exact error at a cost that is much lower than even the software schemes that retrieve only the approximate error. Because \textit{REBits} accelerates an operation, rounding error retrieval, that is integral to many of these schemes, \textit{REBits} can actually be used to accelerate some of these schemes. Table 4 summarizes the possible benefits of using \textit{REBits} to accelerate these schemes. In addition, we perform a detailed case study for \textit{Double-Double} in Section 3.6.2.

We looked at several BLAS (Basic Linear Algebra Subroutines) software implementations, including Intel®’s MKL, because BLAS is commonly used in scientific workloads and contains many running sums. However, we found that most BLAS routines are fine-tuned for performance and offer few accuracy benefits over accuracy-unaware software. \textit{REBits} can be used to improve the accuracy of BLAS without suffering significant performance degradation.
### Table 4. Improving software methods using **REBits**

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>Native Instructions</th>
<th>Native Instructions with <strong>REBits</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Knuth [52]</td>
<td>Addition</td>
<td>6 fpadd</td>
<td>1 fpadd, 1 fgea</td>
</tr>
<tr>
<td>Kahan [53]</td>
<td>Addition</td>
<td>4 fpadd,</td>
<td>2 fpadd, 1 fgea</td>
</tr>
<tr>
<td>Dekker [51]</td>
<td>Addition</td>
<td>3 fpadd</td>
<td>1 fpadd, 1 fgea</td>
</tr>
<tr>
<td>Priest [54]</td>
<td>Addition</td>
<td>7 fpadd, 2 fpcomp</td>
<td>1 fpadd, 1 fgea</td>
</tr>
<tr>
<td>Double-Double [7]</td>
<td>Addition/Subtraction</td>
<td>20 fpadd</td>
<td>6 fpadd, 4 fgea</td>
</tr>
<tr>
<td>(All operands are in Double-Double)</td>
<td>Multiplication</td>
<td>9 fpMult, 15 fpadd</td>
<td>9 fpMult, 13 fpadd, 1 fgea</td>
</tr>
<tr>
<td></td>
<td>Division</td>
<td>3 fpdiv, 16 fpMult, 81 fpadd</td>
<td>3 fpdiv, 16 fpMult, 40 fpadd, 13 fgea</td>
</tr>
</tbody>
</table>

48
3.6.2 REBits to Streamline Double-Double Arithmetic

We have previously (Section 2.3.1) discussed the Double-Double scheme [7] as a software-only approach for extending precision to achieve greater accuracy. Its accuracy is excellent, but its performance and energy consumption are many times greater than Native-64 or REBits-64. We now describe how we have used the REBits idea—recycling the error in floating point addition—to streamline Double-Double arithmetic.

Double-Double arithmetic is costly because it stitches together the results of multiple 64-bit computations on multiple 64-bit values. Consider the addition of two double-doubles, X and Y. X is two doubles (X\text{low} and X\text{high}), and Y is similarly two doubles (Y\text{low} and Y\text{high}). To add X and Y to produce a double-double sum, Z, the Double-Double library performs the operations in Figure 13.

We observe that much of the work performed in the Double-Double code could be simplified using REBits. Dieter et. al [45] made this same observation but did not show how to do so nor did they evaluate it. In Figure 14, we present functionally equivalent code that is accelerated using REBits. The code with REBits enhancement is clearly shorter and simpler. One of the key insights is that the original Double-Double code expends computational effort in inferring the error that REBits explicitly provides.

```c
struct dd{
    double lo, hi;
}
```
dd_add(dd X, dd Y) {
    double s1, s2, t1, t2;
    s1 = sum(X.hi, Y.hi, &s2);
    t1 = sum(X.lo, Y.lo, &t2);  
    s2 += t1;
    s1 = quick_sum(s1, s2, &s2);
    s2 += t2;
    s1 = quick_sum(s1, s2, &s2);
    return dd(s1, s2);
}

double quick_sum(double a, double b, double &err) {
    double s = a + b;
    *err = b - (s - a); // infer error
    return s;
}

double sum(double a, double b, double &err) {
    double s = a + b;
    double bb = s - a;
    *err = (a-(s-bb)) + (b-bb); // infer error
    return s;
}

Figure 13. Original Double-Double addition

struct dd{
    double lo, hi;
}

dd_add(dd X, dd Y) {
    double s1, s2, t1, t2;
    s1 = X.hi + Y.hi;
    s2 = _mm_getErr_sd(X.hi, Y.hi);
    // intrinsic to get rounding error
    // compiles down to fgea instruct.
    t1 = X.lo + Y.lo;
    t2 = _mm_getErr(sd(X.lo, Y.lo);
    s2 += t1;
    s3 = _mm_getErr(sd(s1, s2);
    s1 = s1 + s2;
    s2 = s3;
    s2 += t2;
    s3 = _mm_getErr_sd(s1, s2);
    s1 = s1 + s2;
    s2 = s3;
    return dd(s1, s2);
}
We performed similar transformations of Double-Double routines for multiplication and division. The results for all three operations are included in Table 4. We observe that using REBits can vastly reduce the computational effort—latency and energy—required for Double-Double math.

To illustrate the benefit of streamlining Double-Double math, we ran four of our previous benchmarks with both Double-Double and our streamlined Double-Double. The results in Figure 13 reveal that using REBits improves the ED² of Double-Double by a factor of 7.3 - 8.4.

Figure 14. Double-Double addition with REBits

Figure 15. Results: Accelerating Double-Double with REBits
3.7 Related Work

We are unaware of any prior work that provides lightweight, IEEE-754-compliant architectural support for extending precision, but there is work in related areas.

**Computing using Rounding Error:** There exists some prior work that shows how the rounding error can be used to improve the result of floating point applications. DeHon [44] demonstrates how the rounding error could be used to accelerate running sums. In addition, DeHon [44] presents an FPGA implementation of the hardware required to generate the error. Dieter et al. [45] present a framework to make the rounding error architecturally visible but does not evaluate the potential benefits of making the rounding error available to programmers.

**Co-processors for vector/matrix math:** Researchers have developed co-processors for vector and matrix math [56][57][58], particularly dot-product computations. These co-processors are similar in spirit to our work, but they are far more heavyweight and complicated.

**Hardware support for interval arithmetic:** One method for handling rounding error is to use interval arithmetic [14], in which each nominal value is represented by an interval (often represented with two floating point numbers, the infimum and supremum). Each value’s interval is kept wide enough to include any possible rounding errors. Researchers have developed hardware support for interval arithmetic (e.g., [17]).
and this support is related to our work, but computing with intervals is far more complicated and energy-intensive.

**Non-IEEE Compliant Extended Precision:** Intel processors support the legacy x87 floating point specification, in which floating point values within the core are 80-bit quantities and thus more precise than 64-bit doubles. The x87 standard does not adhere to the IEEE-754 standard, though. Also, because floating point values are 64-bit quantities in the memory system (i.e., not 80-bit), the result of a computation depends on register spills and fills and is thus not deterministic. Brunie et al. [59] propose a new floating point Fused Multiply and Add (FMA) instruction that will accept single precision inputs and produce a double precision result. This scheme allows the cast instructions to be eliminated allowing for better performance than current architectures. However, achieving higher precision than double precision will require significant modifications to the datapath because 128-bit values will be produced by the unit.

### 3.8 Conclusions

In this chapter, we have demonstrated the benefits of architectural support for energy-efficient numerical accuracy. Making the error in each floating point addition architecturally visible provides a simple-to-use “hook” for software to use to extend the precision beyond what is supported—or supported at high performance—by the hardware. Experimental results on real hardware show that REBits-32 can provide accuracy comparable to Native-64 at costs comparable to Native-32 for applications with
complicated loop bodies. Results also show that \textit{REBits-64} can both provide accuracy comparable to \textit{Double-Double} at a small fraction of the cost.


4. Profile Driven Mixed Precision Analysis

4.1 Introduction

In the previous chapter, we introduced REBits. REBits-32 offers accuracy between that of single and double precision and REBits-64 offers accuracy greater than double precision. Despite the compiler pass that we developed to allow programmers to automatically instrument their applications to use REBits, programmers need to know if their application requires the extra precision provided by REBits. Even in existing architectures, programmers need to determine the precision of their calculations. However, each of these different precisions have different accuracy and performance profiles, allowing programmers to make an accuracy performance tradeoff.

In this chapter, we propose a novel, profile driven, scheme to determine the precision of all operations in an application. We gather our profile at the floating point operation level by measuring undesirable behaviors that have the potential to negatively affect the accuracy of the final result. This profile can be used for both debugging, as well as automated mixed precision analysis. Programmers can use this profile to identify floating point operations with “bad” behavior. They can then alter their applications to reduce the incidence of “bad” behaviors. In addition, we demonstrate that our profile can be used to make decisions regarding the precisions of floating point operations.

In this chapter, we make the following contributions:
1. A novel profiling framework detects “bad” numerical behavior in floating point applications. Our profiling framework measures relative size of the rounding error, the range of results, as well as, other numerically interesting behaviors.

2. We develop a debugger that programmers can use to find variables with “bad” numerical behavior. These variables can be handled by either increasing the precision or changing the algorithm.

3. We design a heuristics based approach to automatically determine the precision that all computations are performed in an application based on its numerical profile.

In Section 4.2, we introduce our numerical profile. In Section 4.3, we show how it can be used to debug numerical applications by identifying operations with “bad” behaviors. In Section 4.4, we discuss our scheme to automatically set the precision of operations. In Section 4.5, we explain how programmers can set the thresholds required for our scheme. We present our methodology and evaluation in Sections 4.6 and 4.7 respectively. In Section 4.8, we discuss related works and conclude in Section 4.9.

4.2 Numerical Profiling

We profile applications at the floating point operation level, i.e. for each floating point operation, we detect and quantify “bad” numerical behaviors. The numerical profile consists of the Dynamic Data Flow Graph (DDFG) annotated with the numerical behaviors discussed in the next section. This profile is data-dependent and requires that the training datasets account for the worst-case scenarios.
Though this we quantify these behaviors using in software in this chapter, quantifying them in hardware, in the FPU, would speed up the profiling process significantly. As such, we design our quantifications to be easily, and cheaply, implementable in hardware.

4.2.1 Behaviors

We breakdown the behaviors that we detect into two main camps; one where information is lost due to rounding after each floating point operation and the second undesired numerical behavior based on prior work, i.e. good rules to live by. We do not claim that the behaviors discussed in this section are complete and/or optimal. However, these behaviors are interesting enough to be used to make the decisions shown in Sections 4.3 and 4.4. We believe adding more behaviors can only improve the quality of the numerical profile.

4.2.1.1 Size of Rounding Error

The first behavior that we consider in this work is the size of the rounding error. In any floating point operation, the rounding error captures all information lost due to limited precision. For any operation, $\circ$, with inputs $a$ and $b$ and result $c$, the observed rounding error, $\epsilon$, is given by the following equation:

$$a \circ b = c + \epsilon$$
The rounding error of any given operation is very small, relative to the result. If “enough” operations have a “lot” of rounding error, the accuracy of the final result. Obviously, if the error for any given computation is large, increasing the precision of that computation would reduce the observed rounding error and thereby improve accuracy.

We quantify the rounding error for any operation by comparing the observed rounding error to the theoretical maximum units in the last place (ulp) error. Broadly speaking, if the observed error is close to the ulp error, then the instruction is said to have a “lot” of error and if it is far smaller than the ulp error, then we say that the instruction does not have a “lot” of error. We measure closeness to the ulp error using the following ratio:

\[
\frac{OBSERVED\ ERROR}{ULP\ ERROR} = \frac{\epsilon}{2^{-(LENGTH\ OF\ MANTISSA)} \times c}
\]

The above ratio is less than equal to 1 for most operations, i.e. the observed error is almost always less than or equal to the ulp error. However, if cancellation occurs, then it is possible for the observed rounding error to be larger than the ulp error. We detect cancellation in Section 4.2.1.2.

**Hardware Acceleration**

Since we would like our scheme to be easily implemented in hardware and the above ratio is very expensive to compute in hardware, we approximate the ratio by
applying the division operator on just the exponents of the numbers involved and
dropping their mantissas. The above equation now becomes:

\[
\frac{2^{\exp(\varepsilon)}}{2^{-\text{(LENGTH OF MANTISSA)}}} \ast 2^{\exp(c)}
\]

Applying \(\log_2\) to the above equation results in the following expression:

\[
\exp(\varepsilon) + \text{LENGTH OF MANTISSA} - \exp(c)
\]

Thus, we approximate the original ratio by measuring \(\log_2\) of the ratio of exponents of
the observed rounding error with the ulp error. This new ratio can be easily
implemented in hardware with two adders that are as wide as the length of the
exponent of the floating point number. The hardware required to compute this ratio is
less than the floating point multiplier and divider required to compute the original ratio.

In general, to obtain the exponent of the observed error (\(\exp(\varepsilon)\)) for any
operation, we need the exponent of the result of the computation, which is easily
available in the hardware, as well as the position of the first leading one in the mantissa
bits discarded after rounding. We can then compute the exponent of the error which
would be the exponent of the result minus the length of the mantissa (23 for single
precision or 52 for double precision) minus the number of leading zeros before the first
one in the discarded part of the mantissa (to account for normalization of the error).
4.2.1.2 Cancellation

Cancellation occurs when two very similar floating point numbers are subtracted resulting in higher order bits being cancelled, causing the lower order bits to get promoted to become the higher order bits of the result. Cancellation can significantly affect the result of floating point applications by magnifying small rounding errors. In the example on the left of Figure 16, we present a calculation where cancellation occurs. In this example, the first four decimal digits are cancelled and the last digit of the calculation is promoted to be the leading digit. However, if there is a small error (caused by previous rounding) in the last digit of one of the inputs (the red digit, on the right of Figure 16), it can significantly affect the final result, which is now 0 instead of 100,000 due to a rounding error affecting the least significant bit on the input.

We quantify this behavior by measuring the percentage of the total mantissa bits that are cancelled. Prior work [60][61] has developed checkers to detect cancellation, as well as to quantify the amount of cancellation occurring.

Hardware Acceleration

Cancellation can easily checked in the floating point adder by measuring the number of places the leading one in the pre-rounded mantissa has to be shifted to the left in the normalization stage.
4.2.1.3 Adding Small Numbers to a Large Number

Adding a small number to a very large number will result in the small number being discarded. This behavior is particularly problematic for large summations where the small numbers maybe insignificant on their own but taken together, the sum of all the small numbers maybe be larger than the large number. As such, this behavior can significantly affect the results of running sums. Prior work has shown benefits of sorted summations over unsorted summations [62].

As this occurs for both negative and positive numbers, we consider the absolute value of the numbers rather than a signed comparison. For any addition (or subtraction) with inputs $a$ and $b$, we detect small numbers by measuring the ratio of the two inputs,

$$\frac{|a|}{|b|}, \text{ where } |b| \leq |a|$$
Hardware Acceleration

As in Section 4.2.1.1, the above ratio can be approximated by the following equation:

$$|\exp(a) - \exp(b)|$$

Again, we replace the expensive floating point division with an integer adder and an absolute value unit that are as wide as the exponent.

4.2.1.4 Closeness to Infinity

Increasing the precision from single to double precision provides both accuracy and range benefits. If the values in single precision for the test input datasets are close to the largest single precision value, we want to prevent small changes in the input from pushing those values, which were close to the largest single precision value, to be over infinity by increasing the precision of those operations. As this is true for both negative and positive numbers, we consider the absolute value of the largest positive single precision value instead of the largest negative and positive single precision numbers.

Hardware Acceleration

We measure closeness to the maximum single precision value by the ratio of the floating point number to the maximum single precision value. Again, this ratio can be approximated by looking at the absolute difference of the exponent of the number to the exponent of the absolute maximum single precision number for that precision.
4.2.2 Limitations

Our profile measures “bad” behavior at the floating point operation level. As such, we are implicitly treating each floating point operation as independent and equal. However, floating point operations are not independent and equal, i.e. some operations might have more of an effect on the final result than others. Also, the accumulated error grows across a chain of operations. We do not measure the growth of the error, instead we just measure the contributing factors to the error. As such, our profile is an incomplete, limited view of how rounding error and limited precision affect the accuracy of the final result.

Despite its shortcomings, the benefit of this scheme with respect to other, more complete schemes discussed in Section 4.8, is its cost. Measuring the interactions between floating point operation is far more complicated than just measuring behaviors within each floating point operation. As we shall show later on, though this profiling method is not complete, it is “good enough” for our purposes.

4.2.3 Implementation

Our numerical profiler described thus far works on each floating point operation. We implemented the numerical profiler using a compiler, specifically LLVM (Low Level Virtual Machine) [46]. We developed a compiler pass that instruments the application to emit its numerical profile.
Our compiler pass profiles each operation at the Intermediate Representation (IR) level, instead of at the assembly instruction level. For each IR instruction (henceforth referred to as instruction) that is a floating point operation, we insert calls to a software library that measure the instruction’s numerical behavior. We pass to the software library the input operations to the instruction, as well as a unique identifier for the instruction which consists of the parent function’s name, and the instruction’s destination register. Because LLVM’s IR is Single Static Assignment (SSA), each instruction writes to a unique register and we use this name, combined with the name of the function, as an identifier to tie the instruction to its behavior across all instances. We heavily modified SoftFloat\(^1\), a software implementation of an IEEE-754\([63]\) compliant floating point unit intended for embedded systems to quantify the numerical behavior of each instruction.

We can produce the DDFG annotated with the numerical behaviors as illustrated in Figure 17. The left part of the figure shows the high level code that we are profiling. The center part is the pseudo-LLVM SSA IR that the code on the left maps to. Note that even though we are writing to the variable “a” twice in the HLL code, the second instance of “a” gets renamed to “a.2” in the IR because of the SSA property. Thus, we can label the DDFG on the right with the instruction’s name (destination register)

\(^{1}\) http://www.jhauser.us/arithmetic/SoftFloat.html
thereby allowing for a reverse mapping between the DDFG annotated with the numerical profile and the LLVM IR.

One potential, and subtle, downside of implementing the numerical profiler at the IR level instead of the machine assembly language level, is that the IR may differ from the Instruction Set Architecture (ISA). For example, the IR might contain fused floating point instructions, such as Fused Multiply Add (FMA), not contained in the machine’s ISA or vice versa. This can cause the IR’s DDFG to differ from the assembly language DDFG. We can resolve this issue by ensuring that the Instruction Select phase of the compiler does not fuse, or break apart, IR instructions. If they are broken apart, the profiling compiler pass has to account for the break. We have not experienced this issue but note that it can occur and is something to be considered when designing the profiling compiler pass.

Figure 17: DDFG annotated with Numerical Behavior
4.2.4 Hardware Accelerated Profiling

Though this work profiles the numerical behaviors using a software library, we have designed the quantifiers for the behaviors to be easily and cheaply implementable in the floating point unit. The FPU can be modified to measure these behaviors (as shown in the “Hardware Acceleration” parts in Section 4.2.1) and then to store this information in a flag register. This flag register can then be read after each floating point instruction. This would necessitate the modification of the hardware and the addition of extra floating point flag register to the ISA.

However, in this case, the behaviors read from the FPU flag registers will not be annotated with the “name” of the IR instruction, instead it will be annotated with the Program Counter (PC). These behaviors can then be linked to actual machine instructions using the PC by a hardware-accelerated software library.

4.3 Profile Driven Numerical Debugging

One potential use of the profile generated in the previous section is to debug numerical applications. We believe the information within the profile will be helpful to numerical programmers. Programmers can use this information to identify locations in their applications that are numerically problematic.
4.3.1 Numerical Debugger

Unlike other debuggers like gdb and valgrind, our debugger does not operate in runtime. Instead, it reads the DDFG annotated with numerical behaviors described in the previous section and makes this information visible to the user. As the DDFG is at the IR level, our debugger provides information to the user about IR instructions, not HLL variables. While it is possible to automatically match debugging symbols embedded in the IR to high level constructs, we do not currently support this functionality. However, users can relatively easily perform this matching manually.²

After the numerical debugger reads the DDFG, the debugger breaks it apart into functions. It is important to note that these functions might not necessarily correspond to the functions in the source code especially if function inlining is enabled. Each function consists of one or more instances which corresponds to the number of times the function is called. A function instance contains one or more instruction instances which correspond to the number of times that instruction is executed. For example, an instruction not in a loop might only be executed once while instructions in a loop would be executed multiple times and as such have multiple instances. As we are only profiling floating point operations, a function instance only contains IR instructions that perform floating point arithmetic. In this work, we do not track memory operations.

² [http://llvm.lyngvig.org/Articles/Mapping-High-Level-Constructs-to-LLVM-IR](http://llvm.lyngvig.org/Articles/Mapping-High-Level-Constructs-to-LLVM-IR)
An instruction instance stores the numerical behavior of that instance. It contains a tuple of the values described in Section 4.2.1. A floating point IR instruction contains one or more instruction instances and stores the numerical behavior as a histogram. Thus an instruction contains 4 histograms, one for each behavior, detailing the value of the quantifier of the behavior on the x-axis and the frequency for that value on the y-axis. A function instance stores the behavior of all floating point instructions within that function, as well as that of any function called. It also stores its behavior as a histogram that is an amalgamation of all histograms of all instructions in that function as well histograms of all functions called. Thus, at the top level function, usually “main,” the programmer can view the cumulative behavior of the entire application including all function calls and all floating point variables as shown in Figure 18. The example shown in Figure 18 is purely illustrative and in the actual debugger, the variables are replaced with IR instructions as our debugger operates on the IR DDFG, and not on the HLL variable-level DDFG.

As the debugger operates on the numerical profile, we have to profile the application before we can debug it. We detail the costs of profiling in Section 4.7.5. The costs are both the time required for the profiler to run, as well as the size of the profile.
4.3.2 Example

In this section, we present an example of how one can use the debugger. In this example, we evaluate Rump’s polynomial with particular values for $x$ and $y$ as shown below:

$$y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + \frac{x}{2y}, x = 77617, y = 33096$$

In this case, none of the IEEE-754 precision provide the correct answer as shown in Table 5. Increasing the precision improves the accuracy significantly but does not get the right answer. However, if we use approximately 256 bits, we get the right answer.
Table 5: Evaluation of Rump’s Polynomial

<table>
<thead>
<tr>
<th>Precision Used</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>-6.33E+29</td>
</tr>
<tr>
<td>Double</td>
<td>-1.18E+21</td>
</tr>
<tr>
<td>Extended</td>
<td>1.172604</td>
</tr>
<tr>
<td>Correct Result (~256 bits)</td>
<td>-0.827396</td>
</tr>
</tbody>
</table>

We demonstrate how we can use our debugger to show that the single precision result is incorrect. In addition, we can also make a guess, using the information provided, that we would need more than extended precision to get the correct result. The operation with the worst cancellation in this equation has all twenty-three of its mantissa bits being cancelled. As such, we need more than just single precision; however, it is unclear how much precision we need. If we look at the “Adding Smaller Numbers to a Larger Number” behavior, described in Section 4.2.1.3, we see that the biggest difference in the exponent for addition is ninety-nine meaning that the ratio of the two inputs to the addition is ninety-nine. In floating point addition, we shift the smaller mantissa to the right by the difference in the exponents, 99, this case. In order for the least significant bit not to be lost in this addition, we need at least $99 + 23 = 122$ mantissa bits. As we can see in Table 1, this number is more than that provided in even
extended precision, thus for this application, we probably need more than extended precision.

4.3.3 Usage

For each of the measured behaviors, the debugger presents a histogram that plots of the frequency of behavior quantifiers. Some behaviors may be more tolerant of a single instance of an extreme point than others. For example, if we are adding a very small number to a large number such that the result of the addition is the large number, the actual magnitude of the small number might not really matter because the result is the same. However, our behavior quantifies this by measuring the difference in the exponents; any difference greater than the length of the mantissa is bad, regardless of the actual value of the difference. Also, an application might be able to tolerate rare occurrences of this behavior. For other behaviors, like cancellation, even a single bad operation might significantly affect the accuracy of the application. Thus, programmers can manually assess IR instructions to find the “problematic” ones. Our debugger has a text interface as shown in Figure 19. The green box in Figure 19 shows the function called by the current function (“FunctionName_1”), while the red box contains all the function called, as well as local variables in the application. The blue box shows part of the numerical behavior of the function’s local variables, as well as all called functions.

In this work, we do not explore the correlation between accuracy measured through other metrics, such as various measures for residue or other application specific
accuracy metrics, and the numerical behaviors presented in this chapter. This work can be used to understand some of the underlying reasons that lead to some algorithms suffering from lower accuracy than others. Programmers can use this tool to either manually increase the precision of HLL variables that correspond to the “problematic” IR instructions or to change the algorithm to get rid of troublesome behaviors. One example of the latter scenario could be single precision summation of an array of a few large numbers followed by much smaller numbers. However, by looking at the histogram of exponent differences, the programmer can see that going to double precision might not benefit this particular dataset. This could happen if the exponent difference was greater than the length of not only the single precision mantissa but also the double precision mantissa. Then the programmer has the option of using a software high precision library or some other algorithm to ensure accurate summation.

In addition, our debugger gives the programmer the ability to look at a particular instance of a function (or an instruction) independently from other instances. This allows programmers to make decisions not only across the instruction or function space, but also temporally across the application’s execution. For example, if a particular function “foo” is called \( n \) times but only the last \( m \) calls exhibit “bad” behavior, the user can replace the last \( m \) calls to another function, “foo2”, that exhibits better numerical behavior. “foo2” can either be “foo” in higher precision or a more numerically stable, but slower algorithm than “foo.” The idea of temporal changes can also be applied to
instructions. For example, if an instruction is well behaved in all but the last few
iterations of a loop, the programmer could choose to end the loop early and perform
those last few iterations in higher precision
Figure 19: Numerical Debugger Screenshot for Rump’s Polynomial
4.4 Profile Driven Automated Mixed Precision

In addition to using the numerical profile for debugging, we also show how it can be used for automating mixed precision analysis. We profile an application to collect its numerical behavior. Based on the application’s numerical behavior, we then decide the precisions of the performed computations. In this work, we limit ourselves to single and double precision because they are the precisions most commonly available in today’s hardware. However, this scheme can be easily applied for both 16-bit (half) and 128-bit (extended) precisions.

As the numerical profile and DDFG are gathered at the IR level, we select the precision of IR instructions as opposed to HLL variables. This gives us fine-grained control over the precisions of all of the floating point operations in the application.

4.4.1 Why Automated Mixed Precision?

Mixed precision attempts to achieve the accuracy close to that of double precision with the cost close to that of single precision. This is done by making the “important” variables in an application, which hopefully are few in number, double precision and all “unimportant” variables in single precision. One of the key benefits of mixed precision is that it is easy to use in HLL because it requires only casts. The main challenge with mixed precision is correctly identifying and classifying variables as “important” and “unimportant.” While currently mixed precision refers to only single
and double precisions, one can imagine choosing from a range of different precisions, including half, single, double, and extended.

Manually determining the precision of each variable in an application, for especially large applications, can be time consuming and even hard. Automating the process to decide the precision of variables allows even non-numerical experts to benefit from mixed precision.

Despite its benefits, mixed precision does not come for free. The costs of mixed precision for scalar case are the extra cast instructions added to the code to convert values from single precision to double precision and vice versa. Vectorized mixed precision code requires, in addition to the cast instructions, instructions to pack and unpack the vector registers before and after casting as the number of effective SIMD lanes used changes with vectorization.

Our profile driven automated mixed precision analysis consist of three steps. In the first step, we determine thresholds for “bad” behaviors. We then classify all IR instructions based on these thresholds. Instructions with “bad” behavior are promoted to higher precision while those with “good” behavior can be executed in single precision. After the all the instructions are classified, we re-write the application’s IR to reflect the new precisions of instructions.

As this is a profile-driven scheme, the profile gathered is data dependent. The behaviors we measure are a function of the inputs to the floating point operations. As
such, the mixed precision version of the application is based on the dataset that was used to generate the profile. Thus, for different datasets, we could conceivably end up with different sets of variables in higher precision.

### 4.4.2 Determining Bad Behaviors

We use thresholds, grouped in a Threshold Vector ($\vec{t}$), to detect undesirable behavior. In this section, we determine what to do with instructions with “bad” behavior. Our current implementation uses the five thresholds that, when working together, can detect when the floating point behaviors described in Section 4.2.1 are “bad.” When the quantification of numerical behavior a particular instruction exceeds a threshold, we say that for the particular value of the threshold, that instruction is “bad.” We summarize the thresholds and the behaviors detected by each counter in Figure 20.

![Figure 20: Threshold Behavior Coverage](image)

<table>
<thead>
<tr>
<th>BEHAVIORS</th>
<th>CANCELLATION</th>
<th>RANGE</th>
<th>ERROR RATIO</th>
<th>EXPONENT DIFFERENCE</th>
<th>PERCENTAGE THRESHOLD EXCEEDED</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCELLATION</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOSETNESS TO INFINITY</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIZE OF ROUNding ERROR</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ADDING SMALL NUMBERS TO LARGE</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 20: Threshold Behavior Coverage
Using $\tilde{T}$ is in essence a quantification and automation of the process a programmer would use to determine if a particular IR instruction is problematic using the numerical debugger described in Section 4.3. This automation allows us to make decisions regarding the precision of the IR instructions. However, we note that changing merely the precision is not always enough to improve accuracy; sometimes the underlying algorithm may also have to be changed. Changing the algorithm based on the numerical profile is beyond the scope of this chapter.

### 4.4.2.1 Cancellation Percentage

In Section 4.2.1.2, we described how we quantify cancellation as a fraction of total mantissa bits cancelled for addition or subtraction operations. Our threshold is a percentage that measures the percentage of mantissa bits that are cancelled. When an operation’s cancellation amount exceeds that percentage, that instruction is said to have “bad” cancellation. The cancellation threshold operates on an operation level, i.e. if even one instance an instruction exceeds the cancellation threshold, the instruction is said to have “bad” cancellation because a single cancellation has the potential to ruin the result of the application.

### 4.4.2.2 Range Percentage

The Range Percentage threshold looks at the range of result values of floating point operations to ensure that none of the results are close to infinity. Again, our
threshold is a percentage that measures how close the absolute value of any particular value is to the absolute maximum allowed in that precision. As with cancellation, this threshold also operates on a per instance basis because a single infinity can ruin the final result of the application.

4.4.2.3 Percentage Threshold Exceeded

While the previous two thresholds binned instructions if even a single instance of an instruction exceed the threshold, the next two thresholds bin instructions if the behaviors of “enough” instances of an instruction exceed the thresholds that detect that particular behavior. For each of the next two thresholds, this threshold measures instances whose behaviors exceed the threshold as percentage of total instances for that instruction.

4.4.2.4 Error Ratio

This threshold measures the size of the rounding error as defined in Section 4.2.1.1. If the quantification of the size of the rounding error exceeds the value of this threshold, then that instance of an instruction has a “lot” of rounding error. If “enough” instances of a particular instruction, as measured by the threshold in Section 4.4.2.3, have a “lot” of rounding error, then that instruction is said to have a “lot” of rounding error. This threshold operates on instructions instead of instances because the rounding
error at each operation is relatively small and can only affect the final result if it occurs often enough.

### 4.4.2.5 Exponent Difference

This threshold measures adding a small number to a large number as defined in Section 4.2.1.3. If a particular instance of an instruction’s operand exponent difference (for addition or subtraction) is larger than this threshold, then we can say that one of the operands of the addition was “too” small. If “enough” instances of an instruction, as measured by the threshold in Section 4.4.2.3, add “too” small numbers to larger ones, then we can say that this instruction adds “enough”, “too” small numbers.

### 4.4.3 Classifying Instructions

We use the heuristics grouped together in a threshold vector, \( \bar{T} \), to classify instructions based on their numerical behavior. These instructions are then binned into four distinct bins. While it is possible for some instructions to be placed in multiple bins, these conflicts are resolved using the rules described in Section 4.4.3.5.

#### 4.4.3.1 Single Instruction Promotion Bin

IR instructions that are clearly intended for higher precision are placed in the Single Instruction Promotion bin. Instructions that have “lot” of error as measured by the metric described in Section 4.2.1.1 are placed in this bin. For instructions with
multiple instances, i.e. those in a loop or in a function that is called multiple times, are placed in this bin only if they have a “lot” of error “most” of the time they are accessed.

Instructions that add a “very” small number to a large number as defined in Section 4.2.1.3 “most” of the time are also placed in this bin. Adding a single small number to a larger number is not necessarily problematic. However, if a lot of small numbers are added to a large number, then it is possible that the sum of the smaller numbers would not be small with respect to the larger number.

In addition, instructions that have even a single instance close to $\pm\infty$ as defined in Section 4.2.1.4 are placed in this bin. If even a single instance of an instruction’s value goes to $\pm\infty$, then this single instance can propagate the infinity along to the result of the application.

4.4.3.2 No-Error Bin

Instructions in the No Error bin produce an exact, rounding-free result every time they are executed with the training dataset. Instructions with no rounding error at the current level of precision will, by definition, not benefit from higher precision.

4.4.3.3 Cancellation Bin

Instructions with a “lot” of cancellation as measured by the metric described in Section 4.2.1.2 are placed in the Cancellation bin. If even a single instance of an IR
instruction has a “lot” of cancellation, we place the instruction in this bin because even a single “bad” cancellation can really affect the result of the application.

4.4.3.4 “Other” Bin

Instructions not placed in any of the bins described above are placed in the “Other” bin. Instructions in this bin may benefit from higher precision but were not explicitly chosen to be in higher precision by the heuristics.

4.4.3.5 Instructions in Multiple Bins

In some cases, it is possible for an instruction to be placed in multiple bins. In the worst case, an instruction may be placed in up to three bins (note that by definition, it cannot be in the “other” bin if it is already binned). While it is unlikely to occur in practice, we present a contrived example of how this could happen. If an add instruction with a single instance subtracts two, equal, very large (close to infinity) floating point numbers then this instruction will be placed in all three bins. It will be placed in the “Cancellation” bin because all the mantissa bits would be cancelled. It would be placed in the “No Error” bin, because this operation is free from rounding error. It would also be placed in the “Single Instruction Promotion” bin because the values being subtracted are very large and exceed the “Range Percentage” threshold. The instruction would not be placed in the “Other” bin because it has already been binned.
We ensure that the instruction appears in only one bin by applying the following rules. Our method for handling cancellation, presented in Section 4.4.4.1, is a superset of our method to handle instructions in the “Single Instruction Promotion” bin (Section 4.4.4.2). As such, we only allow instructions to be in one bin with the following priorities:

1) Cancellation Bin
2) Single Instruction Promotion Bin
3) No Error Bin.

By using these priority rules, we can ensure that each instruction in the IR appears in only one bin.

4.4.4 Rewriting the IR

Once we have binned all the instructions in the IR, we invoke the compiler’s mixed precision pass to promote the selected instructions. Our mixed precision pass has two components, the cancellation pass and the single instruction promotion pass. Each of these passes promotes certain instructions to higher precision. Because these passes change floating point arithmetic instruction types, we insert cast instructions before and after the promoted instructions as appropriate to ensure the IR type checker does not fail.
4.4.4.1 Cancellation Pass

The cancellation pass operates on instructions in the cancellation bin. For instructions with cancellation, increasing the precision of merely that instruction would not reduce the number of bits being cancelled as illustrated in Figure 21. This is because cancellation is a property of the input operands to the operation and not the precision.

Therefore, instead of just increasing the precision of operation with cancellation, we need to reduce the accumulated rounding error in the input operands to that operation. Reducing this rounding error will allow us to be more certain that the lower-order bits that get promoted due to the cancellation are more accurate. In Figure 22, we provide an example of this scenario.

The cancellation pass’s algorithm is presented in Figure 23. For each instruction in the cancellation bin, we increase its precision, as well as, recursively increase the precision of all instructions that produce its operands (except those in the “no error” bin). We do increase the precision of producer instructions to the cancellation that are in the No Error bin because, by definition, these instructions will not benefit from double precision and are a good stopping point for the backwards trace. We also do not change the precision of values in memory and do not increase the precision of loads.
**Figure 21:** Higher Precision Computation does not reduce Cancellation

<table>
<thead>
<tr>
<th>4 digits after the point</th>
<th>6 digits after the point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.9834</td>
<td>1.983400</td>
</tr>
<tr>
<td>− 1.9835</td>
<td>− 1.983500</td>
</tr>
<tr>
<td>− 0.0001</td>
<td>− 0.000100</td>
</tr>
</tbody>
</table>

- $1 \times 10^{-4}$

**Figure 22:** Cancellation Backwards Trace in Higher Precision

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation with 4 decimal digits</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>$%r1 = %r2 \times %r3$</td>
</tr>
<tr>
<td>$1.9352 = 1.1800 \times 1.6400$</td>
</tr>
<tr>
<td>$%r3 = 1.94 - %r1$</td>
</tr>
<tr>
<td>$0.0048 = 1.94 - 1.9352$</td>
</tr>
</tbody>
</table>
4.4.4.2 Single Instruction Promotion Pass

The single instruction promotion pass works on instructions in the “single instruction promotion” bin. Each single precision instruction in the “single instruction promotion” bin is promoted to be in double precision. The algorithm for this pass is presented in Figure 24.
4.4.5 Benefits of Profile Driven Mixed Precision Analysis

The key benefit of profile-based analysis is its extensibility from the training dataset to other datasets as long as the numerical behaviors are similar, or less “bad”, than the training dataset’s behavior. Ideally, we would train on a dataset with “bad” numerical behavior and based on this dataset’s numerical behavior, set the precision of all IR instructions. This application, trained on the “bad” dataset, will perform well on data that is “well” behaved. As such, training datasets must always be chosen to have “bad” numerical behavior. The choice of datasets to include “bad” numerical behavior is beyond the scope of this work.

4.5 Choosing Threshold Vectors

Ideally, the programmer would specify $\bar{T}$ and then based on the numerical profile and $\bar{T}$, we would produce a mixed precision application. However, we currently do not know how to translate the thresholds into something a programmer, even one
well-versed in numerical analysis, would be able to use easily. As such, instead of selecting “good” or “bad” threshold vectors, we sweep over the range of threshold values.

We generate a large number of $\overrightarrow{T}$'s by sweeping over the range of threshold values in Table 6. For each of the 5 threshold values, we look at 5 distinct values in their range leading to $3,125 (5^5)$ Sample Threshold Vectors ($\overrightarrow{T}_S$). We call the set of all Sample Threshold Vectors, $S$.

### Table 6: Threshold Values Range

<table>
<thead>
<tr>
<th></th>
<th>Minimum</th>
<th>Maximum</th>
<th>Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation Percentage</td>
<td>0%</td>
<td>100%</td>
<td>25%</td>
</tr>
<tr>
<td>Range Percentage</td>
<td>0%</td>
<td>100%</td>
<td>25%</td>
</tr>
<tr>
<td>Percentage Threshold Exceeded</td>
<td>0%</td>
<td>100%</td>
<td>25%</td>
</tr>
<tr>
<td>Error Ratio</td>
<td>-19</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Exponent Difference</td>
<td>0</td>
<td>23</td>
<td>5.75</td>
</tr>
</tbody>
</table>

### 4.5.1 Result Equivalence

One can imagine a scenario where multiple $\overrightarrow{T}$'s lead to the same set of results. We call this result equivalence. We call the set of all vectors that lead to one result the Result Equivalent Threshold Vector Set ($R$) while each element in $R$ is $\overrightarrow{T}_R$. The size of $R$ can vary
from 1, where every $\overline{T}$ leads to a different result, to the size of $S$. The latter case can happen if there is only one result which means the application is precision indifferent and, as such, does not benefit from higher precision.

4.5.2 Instruction Change Equivalence

There is another, more subtle case where multiple $\overline{T}$s can lead to the precision of the same set of IR instructions being increased, which we call the Instruction Change Set (ICS). We call the process where multiple $\overline{T}$ lead to the same ICS Instruction Change Equivalence (ICE). An example of ICE can be seen in Figure 25. In the top third of Figure 25, we present some pseudo-IR code that loads two elements from memory, multiplies them, and then subtracts the product from the first value loaded from memory. In the second third of Figure 25, we have two different $\overline{T}$s, $\overline{T}_0$, and $\overline{T}_1$, with different threshold values. $\overline{T}_0$ detects cancellation and places “sub.float” in the Cancellation Bin. On the other hand, $\overline{T}_1$ does not detect cancellation, but instead places “sub.float” and “mul.float” in to the Single Error Promotion Bin, because the relative size of the rounding error for those operations is large. Despite these $\overline{T}$s having different threshold values, they both lead to the same ICS. The final third of Figure 25 shows the mixed precision version of the pseudo-IR code for both $\overline{T}$s.
4.5.3 Prime Threshold Vectors

For any set of $T$'s, like $R$, we can find the Prime Threshold Vectors ($T_p$) that reduce the size of the $T$ set without losing information. $T_p$s are the $T$'s in the set that
minimize the size of the expected ICS. We organize $\vec{t}_p$'s in a set called Prime Threshold Vector Set ($P$).

The expected size of the ICS is obtained by insights into the counters. However, this size may be different from the actual size of the ICS as shown in the example in Figure 26. We present two $\vec{t}$'s, $\vec{t}_1$ and $\vec{t}_2$, that differ only in the cancellation amount. $\vec{t}_1$'s cancellation threshold is 40%, meaning that any instruction which has more than 40% of its mantissa bits cancelled will be placed in the cancellation bin, while $\vec{t}_2$'s cancellation threshold is only 30%. Obviously, any instruction that satisfies $\vec{t}_1$'s cancellation threshold ($\geq 40\%$) will also satisfy $\vec{t}_2$'s cancellation threshold ($\geq 30\%$), while the converse statement is not true. If $\vec{t}_1$ and $\vec{t}_2$ lead to the same result, we can safely say that the expected size of the ICS will be smaller for $\vec{t}_1$ than for $\vec{t}_2$. Hence, $\vec{t}_1$ is a Prime Threshold Vector.

On the other hand, in Figure 27, the two $\vec{t}$'s, $\vec{t}_1$ and $\vec{t}_2$, differ in both cancellation and range thresholds. As in Figure 27, when considering only cancellation, $\vec{t}_1$ is expected to have the smaller ICS. However, when we consider the range threshold, $\vec{t}_2$ is expected to have the smaller ICS. As such, and assuming both $\vec{t}_1$ and $\vec{t}_2$ lead to the same result, both $\vec{t}_1$ and $\vec{t}_2$ are Prime Threshold Vectors.

When the size of $P$ is greater than 1, then certain thresholds can be traded for others. In the example in Figure 27, 10% cancellation can be traded for 10% range without affecting the result (with all the other thresholds held constant). This suggests
that there are interactions between the thresholds. In this work, although we do not model these interactions, we show that they do in fact exist. We also note that the existence of these interactions is result and application dependent.

Figure 26: $\vec{T}_1$ is $\vec{T}_p$
4.6 Experimental Methodology

We evaluate the cost of profiling and the quality of the profile driven mixed precision analysis. We do not, however, evaluate the performance of the debugger as it is an interactive tool.

4.6.1 Methodology: Profiler

Profiling numerical behavior is a costly process as it is currently done entirely in software. We evaluate the time spent profiling as well as the size of the profile generated (which is piped to a file). As we measure the behavior of only floating point instructions, both the time to profile and the size of the profile are a function of the number of floating point instructions in the application.
4.6.2 Methodology: Automated Mixed Precision

We sweep over the range of threshold values shown in Table 6. Our goal is to obtain a range of mixed precision results using our method. In addition, we show that Result Equivalence and ICE do indeed occur in our experiments. We also demonstrate that the $T^*$s are very common, implying that the thresholds presented in Section 4.4.2 are not indeed independent and that there exist relationships between the various thresholds. We also evaluate the performance of our mixed precision results. Based on the performance and accuracy of the mixed precision results, we can get the “best” mixed precision result.

We also show that input datasets affect the profile and that the ICSs that lead to the “best” results are different for different datasets. In addition, we show that the choice of accuracy metrics affect the determination of the “best” mixed precision result. While our scheme can generate mixed precision results without the need for application level accuracy metrics, we cannot compare between the generated mixed precision results without at least one application level accuracy metric.

4.6.3 Evaluation Environment

We perform all our experiments on a machine running 64-bit Ubuntu with Linux 3.13.0-37-generic. Our machine has an Intel Core i7-3770 processor with 8 MB of cache and 16 GB of main memory. We use LLVM version 3.4 for our automated mixed
precision analysis. We repeat all experiments to measure the performance ten times in order to account for variations between the different runs.

4.7 Experimental Evaluation

We use four case studies to show the benefits of profile driven mixed precision analysis. The first three are intended to show that our scheme can obtain a range of mixed precision results, while the final case study is intended to show a performance difference between the different mixed precision results, the single precision result, and the double precision result.

In addition, because we are using a profile driven analysis, our scheme is highly data dependent. For different datasets, we expect the same $\tilde{T}$ to lead to different ICSs. In addition, we expect the best ICS, i.e. the ICS that leads to the best result, to be different for different datasets due to the difference in numerical behavior between the datasets. However, as our scheme is dataset behavior dependent, even a small training dataset that exhibits extreme behavior is preferable to a larger training dataset that is better behaved. In the third case study (Section 4.7.3), we empirically explore this behavior.

The first case study, numerical integration, is intended to demonstrate the viability of this scheme. We aim to show that, for a well-behaved application, our scheme can provide a range of mixed precision results. For the second case study, an n-body simulation, we show that increasing the size of the ICS does not, perhaps surprisingly, improve the result. In the third case study, QR decomposition, we look at
the choice of the input dataset and how it affects our scheme. In our final case study, LU decomposition, we show the potential performance benefits of our scheme. Our scheme shows the performance benefits for this application because it is the only one of our applications for which we present vectorised results.

4.7.1 Case Study 1: Numerical Integration

The first case study that we look at is numerical integration of $\int_{-10}^{10} \sin(x) e^x dx$ using the Gauss-Legendre method [64]. The accuracy metric that we use for this benchmark is closeness to double precision, therefore a result that is closer to double precision is said to be more accurate than one that is further.

4.7.1.1 Mixed Precision Results Range

In Figure 28, we present the range of mixed precision results, as well as the single and double precision results. For this benchmark, our scheme results in six unique mixed precision results, though only five are visible in Figure 28, as one of the mixed precision results overlaps with the single precision result. In this benchmark, none of the mixed precision results equals the double precision result. The reason for this difference is that this benchmark has a small storage array, the precision of which is not increased. The distance between the best mixed precision result and the double precision result is $5 \cdot 10^{-5}$. The difference between the single precision result and the double precision result is $\sim 3.5 \cdot 10^{-3}$. Thus, mixed precision improves the result of this application by two
orders of magnitude. Therefore, our profile driven analysis can get a range of results depending on the threshold value settings.

![Figure 28: Range of Mixed Precision Results for Numerical Integration](image)

4.7.1.2 Equivalences

Next, we look at how the sampled threshold vectors lead to the set of unique mixed precision results shown in Figure 28. Figure 29 presents the size of $R$ for each of the 6 unique results. The x-axis is mixed precision results sorted from worst to best. This shows that given $S$, the most common result is, with almost a 50% probability, the best result. In Figure 30, we present the Prime $\overline{T}_R$ for each of the results. The size of $R$ can be significantly reduced for all results. For some results, there exist multiple Prime $\overline{T}_R$s while for others there exists just one. The takeaway from this graph is that for the results, there exists interactions between the thresholds in the $\overline{T}_P$s.

For every prime $\overline{T}_R$s for each result, we look at Instruction Change Equivalence as illustrated in Figure 31. The red bars are the size of the Prime $R$ while the blue bars
are the number of unique ICSs for each result. For result 1, there are five Prime $\overrightarrow{T}_R$'s leading to 4 unique ICSs. These four ICSs lead to the same result. However, Result 2’s five Prime $\overrightarrow{T}_R$'s lead to only 2 unique ICS. For Results 3, 4, and 5, we have only one Prime $\overrightarrow{T}_R$ and obviously, only one ICS. Result 6 has three $\overrightarrow{T}_R$'s leading to only one ICS. While it might seem counter-intuitive that multiple ICSs can lead to the same result, this is possible when we consider that certain instructions in the ICS may not actually affect the result if they are in higher precision. The reason those instructions are allowed to be in higher precision is because we consider the instruction’s numerical behavior when making the decision to promote them, not the instruction’s impact on the final result.

Figure 32 gives the average percentage of original IR instructions that are converted to double precision to produce each unique mixed precision result for each $\overrightarrow{T}$ in $R$. The error bars indicate the range of instructions in double precision for the different ICSs for each result. The x-axis labels are the mixed precision results sorted from worst to best (again with respect to double precision). For this benchmark, the worst mixed precision result (labelled as “Result 1” in Figure 32) has approximately 13% of its IR instructions in double precision with no difference in results, with all the instructions being in single precision. This means that the precision of this 13% of the IR instructions does not affect the result in any way. In addition, the best mixed precision result (labelled as “Result 5”), has around 72% of its IR instructions in double precision.
Recall that our mixed precision framework does not achieve the double precision result for this benchmark.

![Figure 29: Result Equivalence for Numerical Integration](image)
Figure 30: Prime $T^P$ for Numerical Integration
Figure 31: Instruction Change Equivalence for Numerical Integration
4.7.1.3 Performance

The improved accuracy of mixed precision, over single precision, comes at an increase in runtime over single precision. However, as we discussed in Section 4.4.1, mixed precision is an energy/runtime win when compared to the double precision result.
in today’s hardware if the application is vectorized or streaming. However, this application is not vectorized nor is it streaming. As such, for this benchmark, we expect no benefits in runtime for mixed precision. Furthermore, the extra cast instructions added for mixed precision could negatively impact performance, and do, in fact, cause mixed precision to run slower than double precision.

In Figure 33, we present the runtime, measured in seconds, of each of the mixed precision results, as well as the single and double precision results. For this application, the runtime is very small with a lot of noise. To measure the runtime more accurately, we measure the runtime of 10 million numerical integrations. For this application, there is not a lot variation in performance as the accuracy is improves because this application has a very small memory footprint and is not vectorized. The variation that does exist falls well into the category of noise. As discussed earlier, we use this benchmark not to motivate the need for mixed precision, but instead, to gain a better understanding of our scheme.
4.7.2 Case Study 2: Gravitational N-Body Simulation

In the second case study, we advance a gravitational N-Body simulation by one time step. As in the previous case study, the accuracy metric for this workload is closeness to double precision, i.e. how far the position in the XYZ coordinates calculated
by either single precision or mixed precision is from the double precision result for that particular body. Since we have a large number of bodies, we consider both the magnitude of the distances for the worst body, as well as a measure to aggregate the distance across all bodies. For the former case, we use the largest Euclidean distance between the single/mixed precision body and the double precision body, and in the latter case, we use the Frobenius norm of the differences in distances between all bodies. This benchmark demonstrates that different accuracy metrics lead to slightly different behavior. In addition, this application contains significant amount of storage. The precision that values are stored in is outside the scope of this work.

4.7.2.1 Mixed Precision Results Range

The range of unique mixed precision results as we vary the number of bodies in the N-Body simulation is shown in the shaded region in Figure 34. The solid red line shows the single precision result while the dashed line indicates the training dataset size, i.e. the size at which the mixed precision decisions were made (64 bodies). The top graph in Figure 34 shows the Frobenius norm while the bottom graph shows the worst case error. For this benchmark, mixed precision provides 6 unique mixed precision results when we use the Frobenius Norm as the accuracy metric and only 5 unique mixed precision results when we use the worst case error as our metric. This figure illustrates that different metrics can change the analysis. For all subsequent analysis in this case study, we use the Frobenius Norm.
For small values of N, we notice that mixed precision sometimes does poorly with respect to single precision as seen in Figure 35. However, as we can see in Figure 36, as we increase the size of the N, mixed precision starts doing better. Our hypothesis for this behavior is that changes to the precision of instructions will affect applications with smaller datasets more than applications with larger datasets. This is because for small datasets, variations in computation’s precision can really affect the accuracy. As the datasets get bigger, these variations get masked. In addition, we can see in Figure 35 that the order in which results are ordered (based on the given accuracy metric) vary across N. This shows that the best result for one N might be slightly different for N’. However, as we can see in Figure 37, there is distinct clustering of the mixed precision results for this benchmark. As N gets larger, clear clusters develop, thereby masking a lot of the initial noise in the benchmark for small values of N.
Figure 34: Range of Mixed Precision Results for N-Body Simulation with Learning at 64 Bodies
Figure 35: All Mixed Precision Results for small N for N-Body Simulation with Learning at 64 Bodies
Figure 36: Number of times Mixed Precision Result is worse than Single Precision for N-Body Simulation with Learning at 64 Bodies
Figure 37: All Mixed Precision Results with Clustering for N-Body Simulation with Learning at 64 Bodies

4.7.2.2 Equivalences

In Figure 38, we present the size of $R$ against the unique Frobenius Norm result. Again, as in the previous example, a large number of the $\tilde{T}$s in $S$ leads to the best result for this benchmark. In Figure 39, we present the number of Prime $\tilde{T}_{\text{P}}$s for each result.
Results 1 and 2 contain only 1 $\overrightarrow{T_p}$ while the other results have multiple $\overrightarrow{T_p}$s. This again indicates that a relationship exists between some of the thresholds for certain results for this benchmark.

In Figure 40, we present Instruction Change Equivalence for N-Body simulation. Only the $\overrightarrow{T_R}$s associated with Results 4 and 5 lead to more than one ICS. In the previous case study, increasing the precision of more instructions improved the result. In this benchmark, that is not the case as demonstrated in Figure 41. Result 3 has fewer instructions in double precision and has better accuracy than Results 1 and 2. This shows that improving the precision of instructions does not necessarily lead to global improvements in accuracy. It also demonstrates that the limitations and weakness of our numerical profiling scheme, discussed in Section 4.2.2, can be exposed in real-life applications.
Figure 38: Result Equivalence for N-Body Simulation for N-Body Simulation with Learning at 64 Bodies
Figure 39: Prime Result Equivalent Threshold Vectors for N-Body Simulation with Learning at 64 Bodies
Figure 40: Instruction Change Equivalence for N-Body Simulation with Learning at 64 Bodies
4.7.2.3 Learning Dataset Size

Another issue to consider is the effect of the learning dataset size. We look at the ICS that produces the lowest error result at sizes 32, 64, 128, and 256. For this benchmark, the learning dataset does not matter, as illustrated by the overlapping lines.
in Figure 42. This indicates that all the numerical behavior, at the operation level, as measured by the numerical profiler, is the same regardless of the dataset. Increasing or reducing the dataset size (from 32 to 256) does not seem to affect the “best” result for both metrics for this application.

![Graphs showing Frobenius Norm Error and Worst Cast Error](image)

**Figure 42: Best Results at Different Learning Dataset Sizes for N-Body Simulation**
4.7.2.4 Performance

In Figure 43, we present the runtime, measured in seconds, of each of the mixed precision results as well as the single and double precision results. As discussed earlier, we use this benchmark not to motivate the need for mixed precision but instead to gain a better understanding of our scheme. Clearly, for this application, there is clearly no benefit in runtime because this application has a relatively small memory footprint and is not vectorized. In fact, some of the mixed precision results perform significantly worse, when compared to double precision, because of the overhead due to the cast instructions inserted.
4.7.3 Case Study 3: QR Factorization

The third case study in this work is QR decomposition. We decompose a matrix, $A$, into an orthogonal matrix, $Q$, and the residue matrix, $R$. Our accuracy metric measures the “goodness” of this decomposition by computing the following expression:
\[ \|I - Q^T Q\|_F \]

We are measuring how orthogonal the matrix Q is using the Frobenius norm. In addition to the Frobenius norm, we also considered the infinity norm, the 1-norm, and the max norm. In this work, we focus on results primarily from the Frobenius norm as the results using the other norms do not qualitatively affect the analysis presented in this section.

One benefit of this application is the ease with which we can varying the input dataset which is the matrix, A, to be decomposed. We populate A with values drawn from different random distributions while also varying the size of A.

### 4.7.3.1 Mixed Precision Results Range

The range of mixed precision results for this application is given in Figure 44. The input matrix, A, is of size 100x100 and is drawn from the following distribution, \( U[-1, 1] \). In Figure 44, we present results for all 4 accuracy metrics though in subsequent results, we use Frobenius norm (shown in the lower right graph) as our primary accuracy metric. The double precision result is shown by the solid green line which, for these results, is along the x-axis, i.e. has no error. The single precision result is the solid red line. The mixed precision results are given by the blue cloud. As with the previous case study, for small values for N, the worst mixed precision result performs worse than the single precision result. In addition, certain accuracy metrics show worse results than single precision while others do not.
Table 7, we present the mixed precision result counts. Clearly, different accuracy metrics lead to different numbers of unique mixed precision results. Thus, though our scheme can determine the ICS without need for any accuracy metrics, we need accuracy metrics to choose the “best” mixed precision result. As such, the choice of the accuracy metric is important.
Figure 44: Range of Mixed Precision Results for QR Decomposition with Learning Matrix Size 100 x 100 drawn from U[-1, 1]
Table 7: Unique Result Counts for each Accuracy Metric for QR Decomposition with Learning Matrix Size 100 x 100 drawn from U[-1, 1]

<table>
<thead>
<tr>
<th>Accuracy Metric</th>
<th>Number of Unique Mixed Precision Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Norm</td>
<td>8</td>
</tr>
<tr>
<td>Frobenius Norm</td>
<td>8</td>
</tr>
<tr>
<td>Infinity-Norm</td>
<td>8</td>
</tr>
<tr>
<td>Max Norm</td>
<td>5</td>
</tr>
</tbody>
</table>

4.7.3.2 Equivalences

In Figure 45, we present the size of $R$ vs. the unique Frobenius Norm result. While a large number of the $T$'s in the $S$ leads to the best result for this benchmark, the most common result based on $S$ is the second most accurate result. In Figure 46, we present the number of Prime $T^*_R$'s for each result. Only for Results 1 and 3 is the number of $T^*_P$'s one. For all other results, the number of $T^*_P$'s is greater than one. Again, this indicates a relationship exists between some of the thresholds for all but one result for this benchmark.

In Figure 47, we present ICE for QR Decomposition. For this benchmark, only the $T^*_R$'s associated with Results 1 and 4 lead to more than one ICS. In this application, as in the previous one, increasing the size of the ICS does not necessarily improve the result,
as illustrated by Results 3 and 4 in Figure 48. Although Result 4’s ICS is smaller than Result 3’s ICS, it produces a better result. Similarly, although some of the 7 ICS’s that lead to Result 2 are bigger than the Result 3’s ICS, they but lead to a worse result. This reinforces the previous example, showing that improving the precision of instructions does not necessarily lead to improvements in accuracy.
Figure 45: Result Equivalence for QR Decomposition with Learning

Matrix Size 100 x 100 drawn from U[-1, 1]
Figure 46: Prime Result Equivalent Threshold Vectors for QR Decomposition with Learning Matrix Size 100 x 100 drawn from U[-1, 1]
Instruction Change Equivalence vs. Result

Figure 47: Instruction Change Equivalence for QR Decomposition with Learning Matrix Size 100 x 100 drawn from U[-1, 1]
4.7.3.3 Datasets

For this application, we empirically evaluate the effect of different sizes and distribution choice (recall the input matrix $A$ is randomly generated). The purpose of this section is to show that the dataset affects the profile and thus the ICS and the final
result. We also only consider the “best” for each input dataset. We do note that the
other, not the best, mixed precision results generated by our scheme might also be
different for different inputs, but we do not evaluate this.

First, we evaluate the effect of the size of the learning dataset, the matrix $A$, on
the “best” mixed precision result. However, we do this by decoupling $A$’s size from its
distribution. We present the results for four different distributions and three different
learning sizes: 50, 75, and 100. As shown in Figure 49, when the matrix $A$ is drawn from
the uniform distribution, the size of the dataset at which learning is performed at does
not matter. This means that the numerical behavior at the smallest matrix size, 50 x 50 in
this case, is similar to the behavior at size 100. For this dataset, size 50 contains all the
information needed and there is no benefit in increasing the size to 100. In other cases,
like those in Figure 51, Figure 50, and Figure 52, we see that size does in fact matter. In
Figure 50, values are drawn from a uniform distribution from negative one million to
positive one million. We see that for this case, the size of the input matrix is very
important. For different sizes, different behaviors are observed leading to different
“best” ICSs. In Figure 51 and Figure 52, only two lines are seen. For these cases, learning
at size 50 is equivalent to learning at size 100.
Figure 49: Effect of Size for $U[0, 1]$
Figure 50: Effect of Size for U[-1e6, 1e6]
Figure 51: Effect of Size for $U[-1, 1]$
4.7.3.4 Performance

In Figure 53, we present the runtime, measured in seconds of each of the mixed precision results, as well as that of the single and double precision results. The mixed precision results are gathered using matrix input size 100 and distribution, $U[-1, 1]$. For the performance experiments, we use an input matrix size of $1000 \times 1000$, instead of 100
x 100, drawn from the same distribution. We only present results for these parameters as they are representative of results using other input.

Clearly, for this application, there is clearly no benefit in runtime because even though this application has a relatively large memory footprint, and is not cache-resident, the cost of the extra cast instructions inserted overwhelms the benefits from the memory arrays being in lower precision. As we can see, the mixed precision results perform significantly worse than double precision, both in terms of accuracy and performance, because of the number of the cast instructions.
Figure 53: Performance vs. Accuracy for QR Decomposition for Matrix Size 100 x 100 with Learning Matrix Size 100 x 100 drawn from U[-1, 1]

4.7.4 Case Study 4: LU Factorization

The final case study in this work is LU Decomposition. The input to this application is a matrix, $A$, which is drawn from different distributions. In this case study, we focus primarily on the performance impact of mixed precision. We do not discuss equivalences or the impact of accuracy metrics/datasets on our scheme as we have
already explored this in the previous case studies. For a given input matrix, \( A \), we decompose it into a lower triangular matrix, \( L \), and an upper triangular matrix, \( U \). For each of the output matrices, we compare its relative accuracy against the double precision case using the Frobenius Norm. We use the following equation to compute the relative error:

\[
\frac{\|U_{\text{DOUBLE}} - U_{\text{MIXED}}\|_2}{\|U_{\text{DOUBLE}}\|_2}
\]

Again, as in the previous case study, we considered other norms but only show results for the Frobenius Norm.

### 4.7.4.1 Mixed Precision Results Range

In Figure 54, we present the range of mixed precision results for the Frobenius Norm of \( U \) for an input matrix of size 100 by 100 drawn from the following distribution:

\( U[-1e6, 1e6] \)

For this application, the best mixed precision result does not lead to the double precision result (the x-axis in this figure) because LU has large storage arrays whose precision we do not modify as our scheme currently only operates on floating point operations. For all metrics, we see four unique mixed precision results for this application given the input dataset.
Figure 54: Range of Mixed Precision Answers for LU with Learning Matrix

Size 100 x 100 drawn from U[-1e6, 1e6]
4.7.4.2 Performance

For the performance experiment we focus only on one metric; the Frobenius Norm measures the error in the lower triangular matrix with respect to the double precision result. We get the mixed precision ICSs by using the training dataset drawn from $U[-1e6, 1e6]$, with input matrix size 100x100. For the performance experiments, we use a different size to ensure the dataset is either cache resident or not. In addition, for this benchmark, we consider both vectorised and scalar implementations.

Cached

For the cached case, we consider a matrix size of 500 x 500. The scalar case is Figure 55 and the vectorised case is presented in Figure 56. In Figure 55, we can see that all mixed precision results – with the exception of the one with the same accuracy as single precision – perform worse than double precision. This is because the overheads of the extra cast instructions cannot be hidden. In fact, the “best” mixed precision result in terms of accuracy performs significantly worse than double precision. However, for the vectorised case, every mixed precision result is faster than the double precision result. Thus, as discussed in Section 4.4.1, for vectorised applications, mixed precision is a win even for small datasets. We do not see a 2X performance difference between double precision and mixed precision due to the cast instructions, as well as, the bit-shuffle instructions required by mixed precision.
Figure 55: Cached Performance for Scalar LU with Matrix Size 500 x 500 with Learning Matrix Size 100 x 100 drawn from U[-1e6, 1e6]
Figure 56: Cached Performance for Vectorized LU with Matrix Size 500 x 500

with Learning Matrix Size 100 x 100 drawn from U[-1e6, 1e6]

Streaming
For the streaming case, we consider a matrix size of 4000 x 4000. The scalar implementation is shown in Figure 57 while the vector implementation is shown in Figure 58. In the scalar case, we see that none of the mixed precision results outperform the double precision result. However, in the vectorized case, all the mixed precision results outperform the double precision result and have better accuracy than the single precision result. Thus, for both cached and streaming sized datasets, the vectorised version allows mixed precision to outperform double precision and have better accuracy than single precision.
Figure 57: Scalar Streaming Performance for LU with Matrix Size 4000 x 4000 with Learning Matrix Size 100 x 100 drawn from U[-1e6, 1e6]
Figure 58: Vectorized Streaming Performance for LU with Matrix Size 4000 x 4000 with Learning Matrix Size 100 x 100 drawn from U[-1e6, 1e6]
4.7.5 Cost of Numerical Profiling

Profiling the accuracy of applications is not without cost. In this section, we experimentally quantify these costs. The costs are due to the calls to the heavily modified SoftFloat library described in Section 4.2.3. Recall that our software library implements each floating point operation in software which is the primary cause of the slowdown. In Table 8, we summarize the overheads for all the case studies for each of the dataset sizes discussed in this chapter. We see two to five orders to magnitudes increase in runtime with the numerical profiler. The size of the numerical profile, shown in the last column in Table 8, is dependent on the number of floating point operations; for larger applications, the size of the profile is large (e.g. QR Factorization with 100x100 input) and small for other applications (e.g. Numerical Integration).
Table 8: Profiling Overheads

<table>
<thead>
<tr>
<th></th>
<th>Input Dataset</th>
<th>Baseline Runtime (s)</th>
<th>Profiled Runtime (s)</th>
<th>Profile Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical Integration</td>
<td>N/A</td>
<td>4.06E-04</td>
<td>0.1164</td>
<td>32 kB</td>
</tr>
<tr>
<td>N-Body Simulation</td>
<td>32 bodies</td>
<td>4.21E-04</td>
<td>3.03</td>
<td>792 kB</td>
</tr>
<tr>
<td></td>
<td>64 bodies</td>
<td>5.18E-04</td>
<td>12.17</td>
<td>3.1 MB</td>
</tr>
<tr>
<td></td>
<td>128 bodies</td>
<td>7.46E-04</td>
<td>49.61</td>
<td>13 MB</td>
</tr>
<tr>
<td></td>
<td>256 bodies</td>
<td>1.61E-03</td>
<td>200.22</td>
<td>50 MB</td>
</tr>
<tr>
<td>QR Factorization</td>
<td>50 x 50 matrix</td>
<td>1.08E-03</td>
<td>78.85</td>
<td>22 MB</td>
</tr>
<tr>
<td></td>
<td>75 x 75 matrix</td>
<td>2.30E-03</td>
<td>261.69</td>
<td>73 MB</td>
</tr>
<tr>
<td></td>
<td>100 x 100 matrix</td>
<td>4.53E-03</td>
<td>608.57</td>
<td>171 MB</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>50 x 50 matrix</td>
<td>7.59E-04</td>
<td>15.42</td>
<td>4.2 MB</td>
</tr>
<tr>
<td></td>
<td>75 x 75 matrix</td>
<td>9.83E-04</td>
<td>50.96</td>
<td>14 MB</td>
</tr>
</tbody>
</table>

4.8 Related Work

4.8.1 Profiling Floating Point Applications

One scheme to profile floating point applications is FloatWatch [65]. FloatWatch is implanted in Valgrind and, as such, can operate on binaries, instead of only on source code. It works by finding the magnitudes of floating point numbers and checking to the see whether the dynamic range of values for a given computation can fit in single
precision instead of in double precision. However, it does not change the precision of the computations but this can be relatively easy to do manually if the locations of these computations in the HLL source code can be specified by Valgrind.

Benz at al. [60] use binary translation to perform each computation side-by-side in higher precision using shadow variables. They use their scheme to detect rounding errors and cancellation. Their scheme is also implemented in Valgrind. Similar to the Benz et. al[60], An et. al. [66] also compute shadow variables and link each floating point variable to its absolute error in double precision allowing programmers to find variables with the largest error.

Darulova and Kuncak[67] propose a dynamic range analysis scheme for Scala. Their approach uses affine arithmetic, combined with interval arithmetic, to examine how the errors are effected by each operation during execution.

Our work detects dynamic range as well as the relative error of each floating point operation. However, this error is an approximation of the true relative error which we do not present to the programmer. In addition, we do not track the flow of error between operations because we treat each operation independently.

4.8.2 Debugging Floating Point Applications

Fluctuat [68] is a tool that uses static analysis to compute the range of all possible values in the application for both real numbers and IEEE-FP numbers. This allows the programmer to use the tool to find the section(s) of the application that contribute the
most towards the error. Tang et al. [70] develop a statistical technique to ensure the
numerical stability and accuracy of their applications. Their tool dynamically replaces
the last few binary digits of floating point calculations, as well as replacing numerical
expressions with equivalent, real number (not floating point), expressions to introduce
numerical error. It then compares the results of the perturbed versions of the application
with the true version to detect potential instabilities.

Lam et. al[61] develop a tool that instruments code to detect catastrophic
cancellation. Once catastrophic cancellation is found, these calculations are re-run in
higher precision. Bao and Zhang [69] develop a technique to detect numerical instability
in floating point applications by tracking relative errors and detecting if they get too
large.

The debugging tool presented in Section 4.3 is in the same vein as [61] and [69] as
they are both input dataset driven. Again, [69] tracks dependencies between floating
point operations while our scheme treats the floating point operations as independent.
By contrast, [68] and [70] effectively test the application with more inputs than just the
provided input dataset and, thus, the debugging information they provide may be more
useful than the information provided by our scheme if the user supplies a well behaved
input dataset.
4.8.3 Automating Mixed Precision

Rubio-Gonzalez et al. [71] present a scheme to automatically set the precision of all variables in an application. Their scheme performs a search on the variable (IR instruction in this chapter’s terminology) space to determine set of variables that lead to best answer using delta-debugging. Lam et al. [72] propose a scheme that is similar to the previous scheme. Their approach searches the variable space to find ones that produce results that lead to the best answer using binary search. Schkufza et al. [73] develop a framework that uses stochastic search to optimize the precision of floating point applications. These previous approaches use different search algorithms to find the set of variables that need to be in single precision. Our scheme, by comparison, is profile driven an only variables that are said to exhibit “bad” behavior are promoted.

4.8.4 Analytical Floating Point Analysis

While the scheme described in this chapter is profile driven, there also exist analytical schemes to determine the precision of floating point variables. Linderman et al. [74] develop a scheme that uses affine and interval arithmetic to bound the numerical error for all computations in the application. This information can then be used by a compiler to change the precision of variables in the application based on the error of each operation.
Borland and Constantinides [75] develop an analytical framework for reconfigurable architectures that allows for the error of operations to be tracked. They then propose that this information be used to design hardware with non-IEEE compliant floating point units with reduced mantissa size while still meeting the accuracy needs of the application, thus saving silicon area.

### 4.8.5 Generating Input Dataset

This work depends heavily on the training dataset having “bad” numerical behavior, or, at least, worse behavior than the expected input datasets after deployment. If the training dataset has better behavior than expected input datasets, then quality of the mixed precision result will suffer. Chaing et al. [76] present a framework that generates inputs for floating point applications to produce the most error. Our scheme can be used in conjunction with [76] to ensure that the training dataset has “bad” numerical behavior.

While Chaing et al. [76] develop a scheme to generate inputs that produce the highest error, Barr et al. [77] develop Ariadne, a symbolic execution system to detect floating point exceptions. Ariadne executes an application symbolically using real numbers to detect inputs that can cause exceptions. Then, these real number inputs are converted to floating point and the original program is executed with these inputs to see if they also raise an exception.
4.9 Conclusion

In this chapter, we present a profiling framework that operates at the floating point instruction level. For each floating point computation, we quantify behaviors for “goodness”. In this work, we make the assumption that all floating point operations are independent and equal, allowing us to not have to track the interactions between operations. We develop a numerical debugger that programmers can use to find variables with “bad” behavior. However, the key contribution of this chapter is the use of the numerical profile to make heuristics based decisions regarding the precision of the variables in the applications being profiled. We evaluate our scheme to show that the choice of heuristics affects the set of variables in higher precision and, thus, the result of the application. We also show that we can use our mixed precision framework to get improved performance with respect to double precision and improved accuracy with respect to single precision.
5. Conclusion

In this chapter, we review our findings on REBits and Profile Driven Mixed Precision Analysis. In addition, I present some future avenues of research as well as some of my unsubstantiated thoughts for each of these chapters.

5.1 REBits

5.1.1 Review

In Chapter 3, we presented REBits. REBits is an architectural hook for accurate, low-cost floating point summation. REBits makes the rounding error of each floating point addition visible to the programmer. The programmer can then instrument his/her application to use this error to create compensated floating point summations that offer high accuracy. When used with double precision (64-bit) inputs, REBits-64 can be used to obtain higher accuracy than just double precision. However, when used with single precision (32-bit) inputs, REBits-32 can provide better accuracy than single precision and better performance than double precision. In addition, we show that REBits-32 can achieve even better performance than mixed precision due its architectural interface. REBits can be used to accelerate other, pre-existing, numerical schemes that compute the rounding error of floating point operations in software by providing this error architecturally.
5.1.2 Reflections and Future Work

While we only implement REBits for addition, it can also be extended to work for other operations such as floating point multiplication, division, etc. However, in this case the performance benefits of REBits that we see for addition might not translate to other operations. While long chains of additions are common in scientific code; we have not found similar long chains of multiplies or divides in applications. In addition, the error produced by REBits is additive. This additive error would complicate the code for multiplication.

Another potential use-case for REBits might be to track the error and present the programmer with the total error due to rounding for any operation. There are two ways to do this, the first would be similar to interval arithmetic except the growth of the interval is conditioned on the size of the rounding error instead of on the assumption of worst case error [16]. Thus, we can limit the growth of the interval and hopefully achieve tighter bounds on the final result. Producing the error in hardware would accelerate the interval arithmetic library. The second option would be to “shadow” the computations with the rounding error. This would be beneficial to applications that require error bounds as they could be calculated on the fly. However, the downside is that the precision of the “shadow” calculations would be the same as that of the result with the potentially small benefits.
Thus, while REBits-like schemes may be implemented for other operations and use cases, in my opinion, there would be a substantial performance impact. In addition, the benefits of REBits for such use cases might be small and not worth the overhead of modifying the architecture.

5.2 Profile Driven Mixed Precision

5.2.1 Review

In Chapter 4, we present our scheme for profiling the accuracy of numerical applications at the floating point operation level. We develop a debugger that programmers can use to detect “bad” behavior. We also present an automated tool that uses the compiler to increase the precision of variables with “bad” behavior. We use heuristics to determine whether a variable is “bad” or not. We show that in cases mixed precision can offer better performance than double precision.

5.2.2 Reflections and Future Work

There are several avenues for future work. We divide the future work into three broad sections; improving the profiling process, improving the automation process, and finally making this scheme easier to use by programmers who may not be able to understand the thresholds.
5.2.3 Improving the Profiling Process

There are two possible improvements that can be made to the profiling process. The first is to ensure that the behaviors measured are a complete set of behaviors. Also, it is important, and useful to programmers, to relate these behaviors to existing techniques to measure the error of floating point applications.

The profiling process, as it currently stands, is time consuming and expensive as shown in Section 4.7.5. We can use the hardware to quantify the behaviors and thereby significantly accelerate the profiling process.

5.2.4 Improving the Automation Process

Based on Section 4.7.4, we can clearly see that vectorization is almost necessary to achieve a significant performance gain for mixed precision. However, the current implementation does not support vectorised code. One obvious first step would be to add this support. As vectorization affects the accuracy of an application, the numerical profile will have to be gathered for the vectorized version of the code. In addition, the compiler pass described in Section 4.4.4 will have to be modified to work for vectorised code. Vectorized mixed precision will require extra cast instructions and shuffle instructions over those required by the scalar version.

We can also add a performance model to the compiler passes described in Section 4.4.4. The performance model could be used to minimize the number of cast
instructions required. This could occur, for example, if there is a cast instruction from
double precision to single precision, followed by a single precision operation, and then
another cast to double precision. In this case, replacing the single precision operation
with a double precision operation would allow us to save two cast instructions and
thereby reduce the performance overhead of that ICS. However, this change would
affect the accuracy of the final result.

The current implementation does not change the precision of storage; we only
change the precision of floating point operations. One potential solution would be a two
stage process. In the first stage, we profile the behaviors of each of the floating point
operations and change their precisions. In the second case, we profile the cast
instructions that we have added to the code. If the cast instructions from double
precision to single precision before a store to a storage array has “bad” behavior
(however “bad” behavior is defined for a cast instruction), then we can remove the cast
instruction by making the storage array in double precision. Thus, with a two-stage
profiling and instrumentation process, we can allow for the precision of the storage
arrays to be changed as well.

5.2.5 Exposing the Behaviors to Programmers

Programmer who make heavy use of floating point, many of whom may not be
numerical analysts, do not consider the behavior of individual floating point operations.
Instead, they consider the accuracy of the entire application. Such accuracy metrics
might be application specific. As such, we need a model to relate the behaviors observed at the operation level to higher level, pre-existing, accuracy metrics. Then, the programmers would be able to specify the desired accuracy in metrics familiar to them and the model would translate those metrics to threshold values. This model would also allow us to trade off behaviors and gain a better understanding of Instruction Change Equivalence. With this, programmers can easily generate $\tilde{T}$s without having to consider floating point behaviors at the instruction level. Such a system would remove the need for generating a lot of $\tilde{T}$s to find the optimal one.

5.3 Increasing Relevance of Low-Cost Accuracy

Floating point arithmetic is increasingly being used in applications that are accuracy sensitive. In addition, these applications, which can be found at both the high-end (high performance computing) and at the low-end (embedded computing), have performance and energy constraints. For these applications it is important to obtain the right answer at the lowest possible cost, which can have an effect on the bottom line.

We expect the size of applications’ datasets to keep increasing, especially given the push towards exascale computing. This increase in dataset size may reduce the accuracy.

In my opinion, I expect specialized cores to be more prolific in the future. These specialized cores will be application specific and will have their dedicated compute units. These compute units may be some combination of integer, fixed-point, and
floating point units, both IEEE-754 compliant and non-IEEE 754 compliant, with different precisions. Each of these compute units are expected to have different accuracy and performance characteristics. Already, we can see an example of this scenario in today’s heterogeneous systems with both CPUs and GPUs; we see accuracy differences for the same code run on CPUs and GPUs. This difference is mainly due to multi-threading on the GPU.

Thus, in the future, since it will be more complicated for programmers to make accuracy/performance tradeoffs, automated tools will have to be developed to allow programmers to easily make these tradeoffs.
References


**Biography**

Ralph Nathan was born on November 17\textsuperscript{th}, 1989 in New Delhi, India. In 2011, He received a Bachelor’s of Science in Engineering from Duke University in Electrical and Computer Engineering. He received a Master of Science from Duke University in Electrical and Computer Engineering in 2014. In 2015, he received a Doctorate of Philosophy in Electrical and Computer Engineering from Duke University. His PhD research interests were Low-Cost, Resilient Hardware Designs as well as the intersection between Computer Architecture and Numerical Analysis with a view towards improving the accuracy and energy efficiency of numerical applications.