

# AUTOMATED FLOATING-POINT PRECISION ANALYSIS

*A prospectus submitted in partial fulfillment of the degree of Doctor of Philosophy*

*Preliminary Oral Examination for*  
MICHAEL LAM

*Advisor:*  
DR. JEFFREY K. HOLLINGSWORTH

*Committee Members:*  
DR. ATIF MEMON  
DR. ALAN SUSSMAN

Department of Computer Science  
University of Maryland, College Park, MD 20742  
October 26, 2011



## Abstract

As scientific computation continues to scale, it is crucial to use floating-point arithmetic processors as efficiently as possible. Lower precision allows streaming architectures to perform more operations per second and can reduce memory bandwidth pressure on all architectures. However, using a precision that is too low for a given algorithm and data set will result in inaccurate results. Thus, developers must balance speed and accuracy when choosing the floating-point precision of their subroutines and data structures. I am investigating techniques to help developers learn about the runtime floating-point behavior of their programs, and to help them make decisions concerning the choice of precision in implementation.

I propose to develop methods that will generate floating-point precision configurations, automatically testing and validating them using binary instrumentation. The goal is ultimately to make a recommendation to the developer regarding which parts of the program can be reduced to single-precision. The central thesis is that automated analysis techniques can make recommendations regarding the precision levels that each part of a computer program must use to maintain overall accuracy, with the goal of improving performance on scientific codes.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Related Work</b>	<b>8</b>
2.1	Manual Analysis . . . . .	8
2.1.1	Traditional Error Analysis . . . . .	8
2.1.2	Mixed Precision . . . . .	9
2.1.3	Semi-Automated Analyses . . . . .	10
2.2	Automated Analysis . . . . .	11
2.2.1	Interval Arithmetic . . . . .	11
2.2.2	Affine Arithmetic . . . . .	12
2.2.3	Static Affine Analysis . . . . .	13
2.2.4	Dynamic Approaches . . . . .	13
2.3	Floating-Point Alternatives . . . . .	14
2.4	Current Practices . . . . .	14
<b>3</b>	<b>Preliminary Work</b>	<b>15</b>
3.1	Cancellation Detection . . . . .	15
3.1.1	Tool description . . . . .	16
3.1.2	Case study: Instability and pivoting . . . . .	18
3.1.3	Case study: Ill-conditioned behavior . . . . .	20
3.1.4	Conclusions . . . . .	22
3.2	Overflow Detector . . . . .	23
3.3	Framework for Floating-Point Replacement . . . . .	26
3.4	Whole-program Replacement . . . . .	26
3.4.1	Shadow values with table-based look-ups . . . . .	26
3.4.2	Shadow values with pointer-based replacement . . . . .	27
3.4.3	In-place down-cast truncation with bit flag . . . . .	28

3.4.4	Development challenges . . . . .	31
<b>4</b>	<b>Proposed Work</b>	<b>35</b>
4.1	Expanded Overflow Detector . . . . .	35
4.2	Dynamic Range Analysis . . . . .	36
4.3	Mixed-precision Replacement Framework . . . . .	36
4.4	Automated Search . . . . .	38
4.5	Validation and Evaluation . . . . .	41
4.6	Optimization . . . . .	42
<b>5</b>	<b>Future Work</b>	<b>42</b>
5.1	Environmental Support . . . . .	43
5.2	Dynamic Configurations . . . . .	43
5.3	Expansion Replacement . . . . .	44
5.4	Other Precisions . . . . .	44
5.5	Tool Integration . . . . .	45
5.6	Source Modification . . . . .	45
<b>6</b>	<b>Timeline</b>	<b>46</b>
<b>7</b>	<b>Conclusion</b>	<b>46</b>
<b>A</b>	<b>Reading List</b>	<b>47</b>
A.1	Binary Instrumentation . . . . .	47
A.2	Floating-Point Error Analysis . . . . .	47
A.3	Mixed-Precision Algorithms . . . . .	48
<b>B</b>	<b>Fortran Precision Conversion Script</b>	<b>49</b>

## List of Figures

1	IEEE standard formats . . . . .	5
2	Roundoff error demonstration code . . . . .	7
3	Roundoff error demonstration results in varying precisions (correct answer is 1000) . . . . .	7
4	Backward and forward error (dashed line indicates floating-point computation) . . . . .	8
5	Mixed precision algorithm (stars/red indicate double-precision steps) . . . . .	9
6	Sample log viewer results . . . . .	17
7	Classical Gaussian elimination with partial pivoting (Matlab code) . . . . .	18
8	Bordered algorithm for Gaussian elimination (Matlab code) . . . . .	18
9	Example of cancellation in Gaussian elimination . . . . .	19
10	Cancellation for unpivoted Gaussian elimination . . . . .	21
11	Diagonal elements for classical (left) and bordered (right) Gaussian elimination . . . . .	21
12	Cancellation counts for classical (C) and bordered (B) Gaussian elimination . . . . .	22
13	Integer overflow results from NAS DC, aggregated by line number and function; the actual source line is included in the line number aggregation . . . . .	24
14	Purposeful integer overflow in AMG2006 . . . . .	25
15	In-place down-cast replacement . . . . .	29
16	Replacement code snippet template . . . . .	30
17	Replacement code generated for “addsd %xmm0,%xmm2” . . . . .	32
18	Sequoia microkernel overhead for whole-program single-precision replacement . . . . .	32
19	Packed XMM register replacement . . . . .	34
20	Sample configuration file for controlling the replacement analysis; the first column indicates which parts of the program should be run in single-precision (s) and which should be run in double-precision (d) . . . . .	37
21	System overview: the system will generate a number of candidate single-precision replacement configurations from the original program, comparing the final results with the original to find the candidate with the most replacement under the given error threshold . . . . .	39

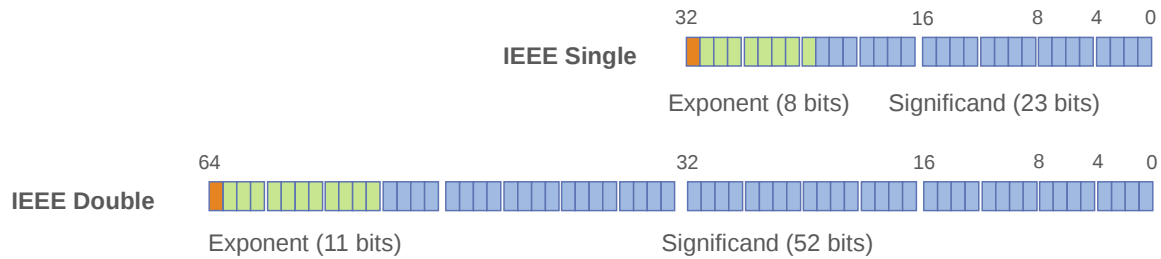


Figure 1: IEEE standard formats

## 1 Introduction

“Floating-point” is a method of representing real numbers in a finite binary format. It stores a number in a fixed-width field with three segments: 1) a sign bit, 2) an exponent field ( $e$ ), and 3) a significand ( $s$ ). The significand is also sometimes called the “mantissa.” The actual value of the number stored is  $s \cdot 2^e$ . Floating-point was first used in computers in the early 1940’s, and was standardized by IEEE in 1985, with the latest revision approved in 2008 [35]. The IEEE standard provides for different levels of precision by varying the field width, with the most common widths being 32 bits (“single” precision) and 64 bits (“double” precision). See Figure 1 for a graphical representation of these formats.

Double-precision arithmetic will generally result in more accurate computations, but with several costs. The main cost is the higher memory bandwidth and storage requirement, both of which are at least twice the footprint of single-precision. Another cost is the drastically reduced opportunity for parallelization. One example of this is on the x86 architecture, where packed 128-bit XMM registers can only hold and operate on two double-precision numbers simultaneously instead of the four numbers that can be stored with single-precision. Finally, some architectures even impose a higher cycle count (and thus energy cost) for each arithmetic operation in double-precision. In practice, it has been reported that single-precision calculations can be 2.5 times faster than corresponding double-precision calculations, because of the various factors described above [27].

As high-performance computing continues to scale to petascale, exascale, and beyond, these concerns regarding precision, memory bandwidth, and energy usage will become increasingly important [22]. Because

of this, application developers have powerful incentives to use a lower precision wherever possible, as long as it does not compromise the overall accuracy. In addition, long-running computations may exhibit numerical accuracy issues not seen at shorter scales [32], providing even more impetus for an analysis solution that accounts for these runtime effects.

The pitfalls of floating-point representations are numerous and have been extensively studied in the decades since its adoption. Most issues stem from the fact that few real numbers can be represented exactly in floating-point; most numbers must be rounded to the nearest real number that is representable. This results in “rounding error” that accumulates and transforms in various ways that can severely compromise the overall calculation. Figures 2 and 3 demonstrate this effect with a sample program that adds a rounded number (0.001) to itself many times, resulting in a number that is incorrect to varying degrees depending on the precision level used. There have also been several real-world incidents involving rounding error, such as the Patriot missile failure in 1995 and the Vancouver stock index slump in the 1970s [38, 33].

Over the years, numerical analysts have developed many techniques for analyzing the error of floating-point computation: backwards and forwards error analysis, interval arithmetic, static analysis. Unfortunately, most of the current analysis techniques are difficult for application developers to apply and/or interpret. Other analyses are limited in scope or are not dataset-sensitive.

My goal is to develop and build analysis techniques that incorporate runtime effects, provide reasonable assurance that code is accurate when run at a lower precision, and provide recommendations to the developer regarding precision levels that will maximize CPU and memory efficiency while minimizing overall rounding error. In this context, “reasonable assurance” means “to within epsilon error with a representative data set.” A recent roadmap describing research challenges in the area of exascale computation highlights the importance of such a system for automatic mixed-precision computation based on a user-specified error margin [22].

This leads to the following thesis statement: *Automated analysis techniques can make recommendations regarding the precision level that each part of a computer program must use in order to maintain overall accuracy.*

```

/**
 * sum1.c
 *
 * Roundoff error example. Accumulates 100,000 additions of the value 0.001,
 * which should result in the value 1000, using three different levels of
 * precision. Roundoff error causes the results to be inaccurate to varying
 * degrees depending on the precision.
 */

#include <stdio.h>

float      sumf  = 0.0;
double     sumd  = 0.0;
long double sumld = 0.0;

void dosums()
{
    int i;
    for (i=0; i<1000000; i++) {
        sumf += 0.001;
        sumd += 0.001;
        sumld += 0.001;
    }
}

int main(int argc, char* argv[])
{
    dosums();
    printf("sumf:   %.20g\nsumd:   %.20g\nsumld: %.20Lg\n", sumf, sumd, sumld);
    return 0;
}

```

Figure 2: Roundoff error demonstration code

```

sumf:   991.14154052734375          (single-precision: 32 bits)
sumd:   999.9999998326507011       (double-precision: 64 bits)
sumld: 1000.00000000000008743      (extended-precision: 80 bits)

```

Figure 3: Roundoff error demonstration results in varying precisions (correct answer is 1000)

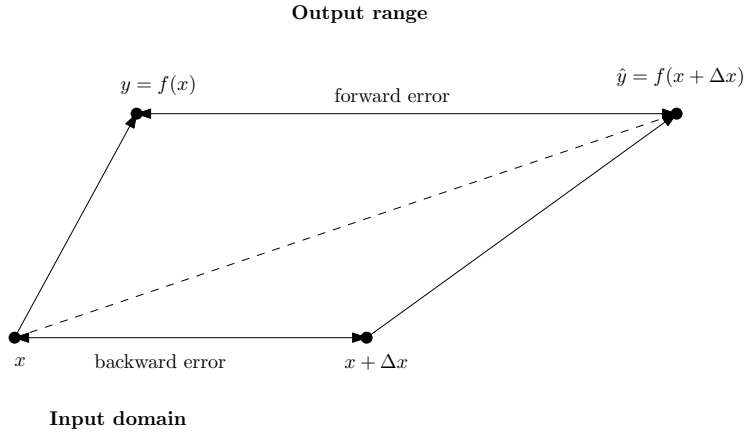


Figure 4: Backward and forward error (dashed line indicates floating-point computation)

## 2 Related Work

### 2.1 Manual Analysis

First we review the large body of work in the area of rigorous manual error analysis. These methods provide mathematically sound analyses of computer algorithms, but are generally difficult or impossible to apply in practice for non-experts.

#### 2.1.1 Traditional Error Analysis

Significant research has explored manual backward and forward floating-point error analysis work, as early as 1959 [20], with Wilkinson’s seminal work in the area being published in 1964 [70]. This research was continued by others throughout the following decades up to the present time [50, 39, 43, 41] and recently summarized by Goldberg and Higham [29, 33].

Forward error analysis begins at the input and examines how errors are magnified by each operation. In general, the result of a floating-point operation  $fl(x_1 \diamond x_2) = (x_1 \diamond x_2)(1 + \epsilon)$ , where  $\epsilon \leq 2^{-p}$  and  $p$  is the number of bits of precision used. Thus, the result of the  $fl(x_1 \diamond x_2)$  operations will gradually begin to diverge from the true answers  $x_1 \diamond x_2$ .

Backward error analysis is a complementary approach that starts with the computed answer  $\hat{y}$  and determines the exact floating-point input  $\hat{x}$  that would produce it (i.e.,  $fl(\hat{x}) = \hat{y}$ ); this “fake” input  $\hat{x}$  can then be

```

1:  $LU \leftarrow PA$ 
2: solve  $Ly = Pb$ 
3: solve  $Ux_0 = y$ 
4: for  $k = 1, 2, \dots$  do
5:    $r_k \leftarrow b - Ax_{k-1}$  (*)
6:   solve  $Ly = Pr_k$ 
7:   solve  $Uz_k = y$ 
8:    $x_k \leftarrow x_{k-1} + z_k$  (*)
9:   check for convergence
10: end for

```

Figure 5: Mixed precision algorithm (stars/red indicate double-precision steps)

compared to the real input  $x$  to see how different they are. This comparison provides an indication of how sensitive the computation is, and how incorrect the computed answer  $\hat{y}$  might be. Computations that are highly sensitive are called *ill-conditioned*.

Figure 4 demonstrates these analyses graphically. Higham [33] describes examples of these analyses for a variety of different numerical analysis problems. Unfortunately, it is difficult for a programmer to understand or apply the results of these analyses without extensive training or error analysis background.

### 2.1.2 Mixed Precision

More recently, many researchers [49, 63, 64, 19, 18, 9, 34] have demonstrated that mixed precision (using double-precision in some parts of a program and single-precision in others) can achieve the same results as using purely double-precision arithmetic, while being much faster and memory-efficient. They usually present linear solvers (particularly sparse solvers) as examples, showing how the majority of operations can be performed in single-precision. These solvers have been applied to a wide range of problems, including fluid dynamics [8], lattice quantum chromodynamics [21], finite element methods [28], and Stokes flow problems [26]. Often, GPUs are cited as the target of these optimizations because of their streaming capabilities [8, 21, 27].

In the iterative algorithm shown in Figure 5, for example, only the steps in red (lines 5 and 8) must be executed in double-precision. The authors note that all  $O(n^3)$  steps can be performed in single-precision, while the double-precision steps are only  $O(n^2)$ . Thus, using mixed precision can yield significant performance and memory bandwidth savings. On the streaming Cell processor, for instance, the mixed-precision version performed up to eleven times faster than the original double-precision version. Even on non-streaming

processors, they obtained a performance improvement between 50% and 80%.

Researchers in the field of computer graphics have also found that mixed-precision algorithms can improve performance [31]. By varying the number of bits used for graphics computations, they report speedups of up to 4X or 5X, with little or no apparent image degradation. They use fixed-point arithmetic, but the mixed-precision concepts are similar to floating-point.

Unfortunately, these techniques are not automatically generalizable to other problems and algorithms. However, this work provides an impetus to develop automatic mixed-precision recommendation techniques.

### 2.1.3 Semi-Automated Analyses

One approach to semi-automatic error analysis is to insert error-tracking statements in computer code manually [71, 37, 33]. This augmented code calculates the error associated with the result at any given point in the calculation, and either 1) adding it back as part of the next calculation or 2) maintaining it through all calculations and producing it as output along with the final result. This approach works, but is tedious and error-prone for several reasons. First, developers still have to work with a numerical analyst to determine the correct error formulas. Second, if the developers ever decide to change any part of the computation, they have to ensure that they also update every corresponding error calculation. Finally, running the code without the overhead requires manually removing the tracking code.

Some programmers manually change their code to use a lower precision in a trial-and-error attempt to get faster code with similar accuracy. Doing this manually, however, can be tedious and troublesome, especially on a large codebase. The programmer must ensure that data is converted where necessary and that library functions receive the size requested. They must also take care to fix any numeric constants and conversion functions to use the new precision levels.

Specialized code like random number generators and optimized graphics routines can create additional problems since it may not be possible to change the precision without drastically affecting the semantics of the program. Random number generators, for instance, frequently perform bitwise integer operations on floating-point data to generate pseudo-random numbers. Changing the precision would change the number of bits available for the integer operations, which would affect the statistical distribution of the generated numbers.

Several other early “one-off” attempts were not widely used [33]. These include methods such as simulating significant digits using subnormal arithmetic [60] and approximate bounding with both absolute [56, 57] and relative error [45, 46].

## 2.2 Automated Analysis

### 2.2.1 Interval Arithmetic

Researchers have attempted to model the behavior of a program using a technique called “interval arithmetic,” [61, 41, 42] which represents every number  $x$  in a program using a range  $\bar{x} = [x.lo, x.hi]$  instead of a fixed value. Arithmetic operations operate on these intervals, usually resulting in a wider interval in the result:

$$\bar{x} + \bar{y} = [x.lo + y.lo, x.hi + y.hi]$$

$$\bar{x} - \bar{y} = [x.lo - y.hi, x.hi - y.lo]$$

Unfortunately, regular interval arithmetic is not always useful due to the quick compounding of errors [7], and the difficulty of handling intervals containing zero [33]. To see why this is, consider a sequence of functions  $f_i$  where the output of each function is the input to the next:  $x_{i+1} = f_i(x_i)$ . Even if the initial value  $x_0$  has an interval width of zero, the interval for  $\hat{x}_1 = f_1(x_0)$  will be  $[x_1 - \delta_2, x_2 + \epsilon_2]$ , where  $\delta$  and  $\epsilon$  are error constants depending on the given precision. The width of this error interval will never decrease; it will only increase proportionally to the condition number of each  $f_i$  in the sequence. In the worst case, division by zero will produce an invalid interval, or the interval will eventually expand to  $(-\text{inf}, +\text{inf})$ , a result that is trivially correct but practically useless. Even in less extreme circumstances, however, the average-case error is rarely as bad as the worst-case, and so interval analysis by itself is usually of little value to programmers who are merely interested in the practical behavior.

Other researchers have proposed extensions or variations on interval arithmetic, but few have proven long-lived. Richman [61] described a rather complex way to use a trial low-precision interval arithmetic calculation to determine what level of precision is necessary for a given calculation. Aberth [6] briefly described a variation on interval analysis that stores the interval midpoint in a high precision, effectively combining

interval analysis with extended precision arithmetic.

Other researchers have tried using stochastic arithmetic [36], applying Monte Carlo methods by representing a number as a set of several numbers obtained by small random perturbations from the original number. By overriding arithmetic operations to operate on all of these values, they approximate interval arithmetic with less overhead. However, this technique has drawn criticism for being ad-hoc and imprecise [38].

### 2.2.2 Affine Arithmetic

Interval arithmetic was later improved by Andrade and others [7] with the concept of “affine arithmetic,” replacing the ranges of interval arithmetic with a linear combination of error factors. In this scheme, a number  $x$  is represented as a first-degree polynomial  $\hat{x}$ :

$$\hat{x} = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \cdots + x_n\epsilon_n$$

Affine representation preserves information about error independence, and allows some errors to cancel out others. In the following example, for instance, the error term  $\epsilon_4$  is shared between the two numbers. This sharing indicates that the error came from the same input and will cancel out in the sum. Thus, the bounds for the result are tighter than those that would be obtained in standard interval arithmetic.

$$\begin{array}{rcccc} \hat{x} = & 10 & +2\epsilon_1 & +1\epsilon_2 & & -1\epsilon_4 \\ \hat{y} = & 20 & -3\epsilon_1 & & +1\epsilon_3 & +1\epsilon_4 \\ \hline \hat{x} + \hat{y} = & 30 & -1\epsilon_1 & +1\epsilon_2 & +1\epsilon_3 & \end{array}$$

The authors relate simple formulas for handling operations on numbers that are composed of simple linear functions involving affine numbers  $\hat{x}$  and  $\hat{y}$  and scalars  $\alpha \in \mathbb{R}$ :

$$\begin{aligned} \hat{x} \pm \hat{y} &= (x_0 \pm y_0) + (x_1 \pm y_1)\epsilon_1 + \cdots + (x_n \pm y_n)\epsilon_n \\ \alpha\hat{x} &= (\alpha x_0) + (\alpha x_1)\epsilon_1 + \cdots + (\alpha x_n)\epsilon_n \end{aligned}$$

$$\hat{x} \pm \alpha = (x_0 \pm \alpha) + (x_1 \pm \alpha)\epsilon_1 + \dots + (x_n \pm \alpha)\epsilon_n$$

For non-affine operations, the authors choose an approximation function and add an extra error term. This extra error term is considered independent from the numbers, even though it is a function of them. This causes the analysis still to be less precise than an optimal analysis.

Affine analysis is more expensive than regular interval analysis, but the authors maintain that the extra expense is worth it when normal interval analysis fails. The actual implementation relies on a sparse representation of the error coefficients, with a stack that allows the reclamation of indices after they are no longer needed. The authors present two applications in the field of computer graphics and vision: 1) computation of octrees and quadrees, and 2) ray-tracing. These applications demonstrate how affine arithmetic can save computation time in domains where errors are highly dependent. The authors do not present results for other domains.

### 2.2.3 Static Affine Analysis

More recently, Goubault and Martel and others [30, 23, 52, 53, 25] have built abstract semantics and static analyses using affine arithmetic. These techniques, like any static analysis, are entirely *a priori* and give conservative estimates. In addition, the most recent work [54] describes a system that can actually perform program transformations to increase accuracy. These transformations involve rearranging operations according to well-known rules of floating-point arithmetic, rather than by adjusting the precision.

Unfortunately, the fact that they are static analyses also means that they are not dataset-sensitive, and may give conservative estimates that may not be useful. In addition, they require tuning by the programmer, particularly with regards to the extent that loops are unrolled: more unrolling produces better answers but requires more lengthy analyses. These techniques also only work for a subset of language features (often excluding HPC-specific interests like MPI communication), and are usually limited to C programs.

### 2.2.4 Dynamic Approaches

There are a variety of generic dynamic binary instrumentation frameworks, including EEL [47], DyninstAPI [17], LLVM [48], Pin [51], and Valgrind [58, 59]. None of these tools provide native floating-point replacement

analysis, but provide a framework for building one. I chose to use DyninstAPI because it does not use an intermediate representation and provides the ability to add and remove instrumentation dynamically at runtime, a feature that may prove useful during the automated search phase of my analysis.

FloatWatch [15, 16] is a dynamic instrumentation approach that uses the Valgrind framework to monitor the minimum and maximum values that each memory location holds during the course of execution. This type of range information could be used to adjust the precision. For instance, if a value has a small dynamic range, it can probably be stored in reduced precision. Although this type of analysis is not conclusive, I do plan to use it as I investigate automated precision analysis. Unfortunately, FloatWatch no longer seems to be in active development.

There has also been some recent work on fault-tolerant computing with probabilistic accuracy bounds [62]. This effort attempts to quantify in a probabilistic model the failure rate of particular portions of a program, called “task blocks.” Once the failure rates are known, their system can pre-emptively abort task blocks to short-circuit failures and reduce the overall runtime while maintaining an acceptable level of accuracy on the final results. This approach is designed for hardware and software errors, however, and relies on the failures being relatively easy to detect. It’s not clear how this technique would be extended to floating-point roundoff analysis, where error detection is the core issue.

## 2.3 Floating-Point Alternatives

Finally, there are alternatives to using floating-point numbers. There are multi-precision libraries that allow large or even variable precisions [2, 3, 13, 11]. Some of these libraries also provide a rational representation, storing real numbers using a ratio of integers. Both of these approaches decrease the chance of numerical issues at the cost of a much higher computation time. There can also be a high cost in developer time to the converting of a legacy codebase, although some researchers have developed automated tools for this purpose [13, 14, 66].

## 2.4 Current Practices

Unfortunately, the practical reality is that many developers still do not utilize any of these methods, either because of the difficulty of use or the permanent performance penalties. Often, a programmer will simply

recompile at a lower precision and re-run the program to see if the answer is “close enough” to be acceptable. Other programmers choose to ignore the possibility of catastrophic rounding error, using the higher double-arithmetic and accepting the performance hit. For many engineering applications, however, the measurement error far exceeds any rounding error and single-precision would be sufficient. My goal in this proposed research is to build an automated system that does not require specific numerical analysis knowledge to use, but that still provides useful recommendations about the choice of precision in a computer program.

### 3 Preliminary Work

This section describes work that I have already done in this research area, and prototypes that I have already built. My initial preliminary work was to develop techniques that would detect numerical cancellation at runtime [44]. This initial method was chosen because the level of analysis required to detect cancellation is much simpler than that required for alternate precision analysis. The techniques worked and the tool was successful, allowing me to venture into other directions towards the ultimate research goal of automated precision analysis. I have since developed several other prototypes of various methods for running a program in an alternate precision.

#### 3.1 Cancellation Detection

Numerical cancellation occurs when an instruction subtracts two numbers that are identical in many of their digits. The identical digits are “canceled,” and the resulting number has fewer significant digits than either of the operands. For example, consider the following operations:

$$\begin{array}{r}
 1.613647 \quad (7) \\
 - 1.613635 \quad (7) \\
 \hline
 0.000012 \quad (2) \\
 \text{(a)}
 \end{array}
 \qquad
 \begin{array}{r}
 1.613647 \quad (7) \\
 - 1.613647 \quad (7) \\
 \hline
 0.000000 \quad (0) \\
 \text{(b)}
 \end{array}$$

In the operation on the left (a), the operands all have seven significant digits, while the result only has two. In the operation on the right (b), the problem is even worse; all digits cancel and the result has no significant digits. This cancellation may seem innocuous; after all, the answer is correct! However, consider what may

happen if the two numbers were not actually identical, but were rounded by previous operations. If the difference between the numbers is ever used as a scalar in a multiplication operation, for example, the result will be dramatically different (zero!) than expected.

Cancellation is something like a compiler warning message in this respect. It can tell that there may be a problem, but not necessarily exactly where the problem is. In the case of the above examples, the actual problem might be in the routine that rounds the numbers directly before this calculation.

### 3.1.1 Tool description

I used the DyninstAPI library [17] to insert instrumentation using an approach that replaces a section of executable code with a call to a *trampoline*, a newly-allocated area of code that contains the original (now relocated) instructions as well as the desired instrumentation code. I also used the XED instruction decoder from the Intel Pin toolkit to parse floating-point instructions [5].

The tool instruments every floating-point addition and subtraction operation, augmenting it with code that retrieves the operand values at runtime. The algorithm compares the binary exponents of the operands ( $exp_1$  and  $exp_2$ ) as well as the result ( $exp_r$ ). If the exponent of the result is smaller than the maximum of those of the two operands (i.e.,  $exp_r < \max(exp_1, exp_2)$ ), cancellation has occurred. The *priority* is defined as  $\max(exp_1, exp_2) - exp_r$ , a measure of the severity of a cancellation. The analysis will ignore any cancellations under a user-defined minimum threshold, with a default setting of ten binary digits (roughly three decimal digits). Cancellations of one or two decimal digits occur on most subtraction operations and can safely be ignored in most cases. If the analysis determines that the cancellation should be reported, it saves an entry to a log file. This entry contains information about the instruction, the operands, and the current execution stack. The analysis also maintains basic instruction execution counters for the instrumented instructions.

Since many programs produce thousands or millions of cancellations, it is impractical (and unhelpful) to report the details of every single one. Instead, we use a sample-based approach. Unfortunately, there is a large discrepancy between the number of cancellations at various instructions. In the same run, some instructions may produce fewer than ten cancellations while others produce millions. Thus, a uniform sampling strategy will not work, and I instead implemented a logarithmic sampling strategy. In my tool, the first ten cancellations for each instruction are reported, then every tenth cancellation of the next thousand, then every hundred thousandth cancellation thereafter. I found that this strategy produces an amount of

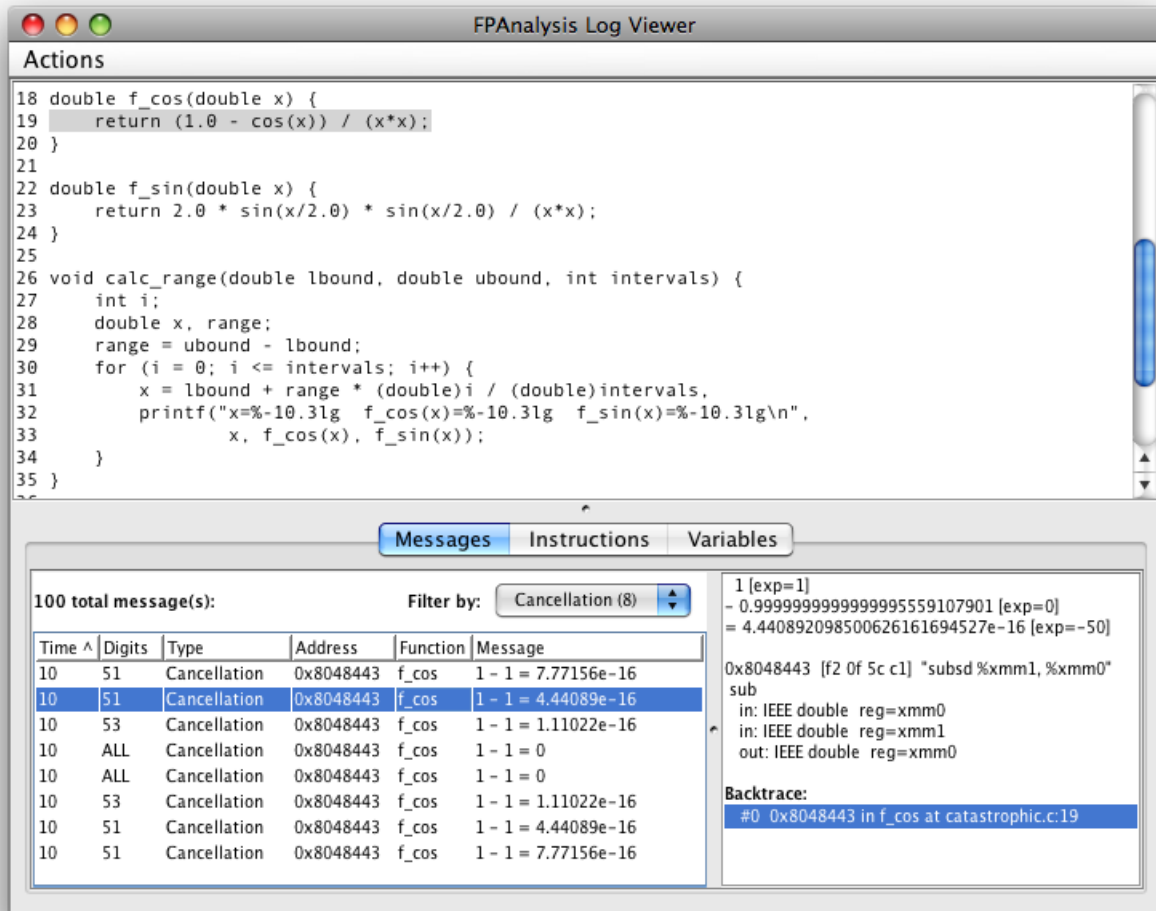


Figure 6: Sample log viewer results

output that is both useful and manageable. A point to emphasize is that all cancellations are counted and that the sampling applies only to the logging of detailed information such as operand values and stack traces.

I also created a log viewer (see Figure 6) that provides an easy-to-use interface for exploring the results of an analysis run. This viewer shows all events detected during program execution with their associated messages and stack traces. It also aggregates count and cancellation results by instruction into a single table.

The viewer also synthesizes various results to produce new statistics. Along with the raw execution and cancellation information, it also calculates the *cancellation ratio* for each instruction, which is defined as the number of cancellations divided by the number of executions. This ratio gives an indication of how cancellation-prone a particular instruction is. The viewer also calculates the average priority (number of

```

1. perm = 1:n
2. for k=1:n
3.     [maxak, kpvt] = max(abs(A(k:n,k)));
4.     A([k,pvt],:) = A([pvt,k],:);
5.     perm([k,pvt]) = perm([pvt,k]);
6.     A(k+1:n,k) = A(k+1:n,k)/A(k,k)
7.     A(k+1:n,k+1:n) = A(k+1:n,k+1:n)
                       - A(k+1:n,k)*A(k,k+1:n);
8. end

```

Figure 7: Classical Gaussian elimination with partial pivoting (Matlab code)

```

1. for k = 2:n
2.     A(k,1:k-1) = A(k,1:k-1)/triu(A(1:k-1,1:k-1));
3.     A(1:k-1,k) = (tril(A(1:k-1,1:k-1),-1)
                    + diag(ones(1,k-1)))*A(1:k-1,k);
4.     dot = A(k,1:k-1)*A(1:k-1,k);
5.     A(k,k) = A(k,k) - dot;
6. end

```

Figure 8: Bordered algorithm for Gaussian elimination (Matlab code)

canceled bits) across all cancellations for each instruction, indicating how severe the cancellations induced by that instruction were.

I verified the tool using some small examples, and in a previous paper [44] with other researchers presented several case studies demonstrating the usefulness of the technique. These case studies revealed two observations about the ability of cancellation detection to shed light on a particular algorithm. First, almost all algorithms contain a background of trivial cancellations that can mask more significant ones. Second, some algorithms may conceal a significant cancellation under a sequence of small, harmless looking cancellations. We examined these limits by looking at two issues in Gaussian elimination: 1) the instability of classical Gaussian elimination without pivoting and 2) the ability of Gaussian elimination to detect ill conditioning in a positive definite matrix.

### 3.1.2 Case study: Instability and pivoting

Matlab code for Gaussian elimination with partial pivoting is given in Figure 7. The result of this code is a unit lower triangular matrix

$$L = \text{tril}(A, -1) + \text{diag}(\text{ones}(1, n))$$

$$\begin{bmatrix} 1.00000 \cdot 10^{-03} & 1.00000 \cdot 10^{+00} & 1.00000 \cdot 10^{+00} & 1.00000 \cdot 10^{+00} \\ 1.00000 \cdot 10^{+00} & -7.92207 \cdot 10^{-01} & -3.57117 \cdot 10^{-02} & -6.78735 \cdot 10^{-01} \\ 1.00000 \cdot 10^{+00} & -9.59492 \cdot 10^{-01} & -8.49129 \cdot 10^{-01} & -7.57740 \cdot 10^{-01} \\ 1.00000 \cdot 10^{+00} & -6.55741 \cdot 10^{-01} & -9.33993 \cdot 10^{-01} & -7.43132 \cdot 10^{-01} \end{bmatrix}$$

(a)

$$\begin{bmatrix} -1.00079 \cdot 10^{+03} & -1.00004 \cdot 10^{+03} & -1.00068 \cdot 10^{+03} \\ -1.00096 \cdot 10^{+03} & -1.00085 \cdot 10^{+03} & -1.00076 \cdot 10^{+03} \\ -1.00066 \cdot 10^{+03} & -1.00093 \cdot 10^{+03} & -1.00074 \cdot 10^{+03} \end{bmatrix}$$

(b)

$$\begin{bmatrix} -6.40000 \cdot 10^{-01} & 9.00000 \cdot 10^{-02} \\ -1.02000 \cdot 10^{+00} & -1.90000 \cdot 10^{-01} \end{bmatrix}$$

(c)

Figure 9: Example of cancellation in Gaussian elimination

and an upper triangular matrix  $U = \text{triu}(A)$  such that

$$A(\text{perm}, :) = L * U.$$

The purpose of the partial pivoting in lines 3–5 of Figure 7 avoids division by zero in line 6, but it also avoids inaccurate results if  $A(k,k)$  is small. To see why, consider what happens when we omit lines 3–5 in Figure 7 and apply it to the matrix shown in Figure 9(a). After line 6 the elements of  $A(2:4,1)/A(1,1)$  are all  $10^3$ , so that we can expect a large matrix when we compute the Schur complement  $A(2:4,2:4)$ . Indeed, we get the matrix shown in Figure 9(b). Since all numbers in the Schur complement are approximately  $-10^3$ , we can expect cancellation when we compute the next Schur complement, as shown in Figure 9(c). The numbers in this matrix are back to the original magnitude, but as the trailing zeros indicate, they now have at most two digits of accuracy.

The cancellation detector can point out that is happening and show the developer where the loss of digits occurs. Although the cancellation itself introduces no significant errors, the rounding of data that occurred between Figure 9(a) and 9(b) set up a situation where the subtraction of  $10^3$  from the elements of  $A(2:4,2:4)$  caused about four digits to be lost in each of the elements. Thus, cancellation is a lot like a null pointer dereference error, where the null pointer exception is not itself the problem, but rather the notification of an earlier error.

To see how well cancellation due to lack of pivoting was detected by our system, we performed the following experiment. We first generated a matrix  $A$  of order  $n$  that had a pivot of size  $10^{-s}$  at stage  $p$  of the

elimination. In the example above,  $n = 4$ ,  $s = 3$ , and  $p = 1$ . We then ran the elimination and counted cancellations. We set the threshold (the number of bits required for a cancellation to register) at  $\log_2 10^{s-2}$  rounded to the nearest integer greater than zero, meaning that we regard cancellations of greater than  $s - 2$  decimal digits as significant. As the threshold is increased over this value we increasingly risk missing cancellations due to the bad pivot. As it is decreased we increase the risk of including cancellations not due to the pivot (i.e., background cancellations).

We can compute the number of cancellations we can expect due to the bad pivot by determining the dimensions of the array where the cancellation will occur. It is of order  $n - p - 2$ , and hence the expected number of cancellations  $(n - p - 2)^2$ . We can also estimate the background cancellation. The matrix  $A$  was generated in such a way as to dampen cancellation before  $k = p$ . If we then stop the process after the cancellation (at  $k = p + 1$ ) and if  $p$  is not large, the cancellation count will be a good estimate of the cancellation due to the bad pivot.

The results are summarized in Figure 10. The rows labeled “Count” give the cancellation counts for the entire elimination while the rows labeled “Trunc” give the count for the truncated elimination (stopping after the bad pivot). The rows labeled “Est” contain the cancellation count as estimated by the formula  $(n - p - 2)^2$ .

In the first column, the counts considerably overestimate the amount of cancellation due to the bad pivot, because of the small value of the threshold. In the remaining three columns, all counts are in reasonable agreement. This suggests that if care is taken to keep the threshold high enough, one can detect the effects of a reasonably small pivot. A potential application for this method is to sparse elimination, where the ability to pivot can be limited.

### 3.1.3 Case study: Ill-conditioned behavior

Our second example concerns the ability of Gaussian elimination to detect ill-conditioning. To avoid the complications of pivoting, we worked with positive definite matrices, where pivoting is not required to guarantee stability. Let us suppose that we have a positive definite matrix  $A$  for which the eigenvalues descend in geometric progression from one to  $10^{(-\log \kappa)}$ . Thus,  $A$  has the condition number  $\kappa = \|A\| \|A^{-1}\| = 10^{\log \kappa}$ . The diagonals of  $U$  become progressively smaller, and result in higher levels of cancellation as the algorithm proceeds. We wanted to see how well these cancellations would be detected by our techniques.

log(size)	-2	-4	-6	-8
Threshold	1	7	13	17
$n = 10$				
Count	66	37	37	34
Trunc	55	37	37	34
Est	25	25	25	25
$n = 15$				
Count	225	123	122	122
Trunc	154	122	122	122
Est	100	100	100	100
$n = 20$				
Count	663	247	252	257
Trunc	298	245	252	257
Est	225	225	225	225
$n = 25$				
Count	1227	394	423	441
Trunc	447	381	423	441
Est	400	400	400	400

Figure 10: Cancellation for unpivoted Gaussian elimination

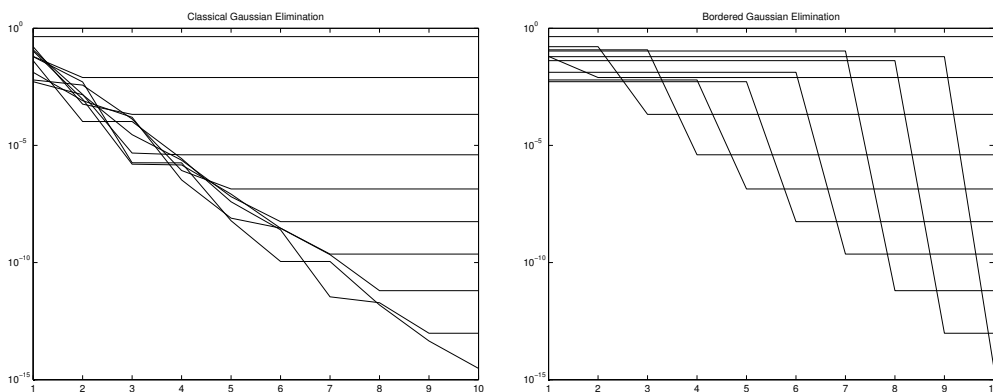


Figure 11: Diagonal elements for classical (left) and bordered (right) Gaussian elimination

A difficulty here is that Gaussian elimination has many variants. Consider, for example, the code in Figure 8. It performs Gaussian elimination by bordering; after step  $k$ ,  $A(1:k, 1:k)$  contains the LU factorization of the original submatrix  $A(1:k, 1:k)$ . Numerically the algorithms are almost identical, even to the effects of rounding error. However, they exhibit the cancellation in different ways. The plots in Figure 11 contain histories of the diagonal elements of the reduction of the matrix  $A$  described above with  $n = 10$  and  $\log\text{kap} = 15$ . Naturally, the initial and final values for both methods are identical. The x-axis is the step in the elimination and the y-axis is the value of the diagonal element in question.

The difference in the behaviors of the two methods is remarkable. For classical Gaussian elimination, each diagonal remains constant after the iteration in which it is calculated but decreases in every iteration before

threshold	1		2		3		4		5	
logkap	C	B	C	B	C	B	C	B	C	B
5	14	8	8	7	1	6	0	5	0	4
10	29	8	23	8	16	7	11	7	3	6
15	39	9	33	9	27	9	21	8	17	8

Figure 12: Cancellation counts for classical (C) and bordered (B) Gaussian elimination

that. In the border variant, all diagonals remain constant during each iteration except the value that is being calculated, which drops to its final value and remains constant thereafter. To summarize, the classical method has many small cancellations while the bordered method has fewer and larger cancellations even though they end up at the same values.

All this suggests that cancellation detection will work better for the bordered variant. Figure 12 contains counts for both for various values of the cancellation detection threshold and logkap. It is easy to see by counting the drops in the graph for the border method that it should register nine cancellations; this is indeed the case unless the threshold is too high or logkap is too small. Ideally, classical Gaussian elimination should register 45 counts: nine in the first step, eight in the second, seven in the third, etc. However, a look at the plot shows that the sizes of the cancellations varies irregularly, so that there are small ones that may fall by the wayside due to being under our priority threshold. It comes near 45 only with logkap equal to 15 and a threshold of one bit.

### 3.1.4 Conclusions

From these experiments, we observed the following:

1. It is important to vary the threshold. Most computations have a background of small cancellations, which will overwhelm more important cancellations if the threshold is set too low. Trying different thresholds may give a better view of what is happening.
2. As a corollary to the first, cancellations near the background cannot be made to stand out. In particular if a large cancellation is obtained by a sequence of smaller cancellations, it may go undetected. Classical Gaussian elimination in the second experiment is an example.
3. Cancellation detection is not a panacea. It requires interpretation by someone who is familiar with the algorithm in question. Nonetheless, the experiments also suggest that cancellation detection, properly

employed, can find trouble spots in an algorithm or program.

Additionally, we note that not all cancellations are bad. A good example is the computation of a residual to determine the convergence of an iterative method. Since a small residual means convergence, high cancellation in its computation means that the answer is close to the correct value. There are also instances where cancellation indicates particular situations in the problem domain that may be of interest. For instance, many cancellations in a nearest-neighbor algorithm may indicate that the point field is dense. Another example occurs in the program POV-Ray, part of the SPEC benchmark [4]. In some color-calculating routines, color components are calculated by subtracting a floating-point number from the value 1.0, and thus high cancellation indicates a color close to black.

All of these observations indicate that this tool provides interesting information that can lead to the identification of trouble spots in a program, or to a verification that the program behaves as expected. Unfortunately, this type of analysis falls prey to many of the same criticisms as the techniques described in the related work section; i.e., that it can be difficult for programmers to use and may frequently give non-useful results.

One potential future use of this tool is to inform the precision level automatic search (see later sections) about deviations before the end of an execution. If a calculation can be performed in single-precision, the number and severity of cancellations should be similar or identical to those obtained using double-precision. This was verified on the Gaussian elimination application using a large matrix. Running cancellation detection alongside replacement analysis could allow the search mechanism to short-circuit failed configurations if the cancellation reaches a given threshold beyond that of the original program.

## 3.2 Overflow Detector

I have also implemented a simple technique that detects runtime overflow in integer programs, and is useful for finding math errors and bugs in large codes. I wrote a Dyninst mutator that instruments every instruction that modifies the OF (overflow) flag in the `%rflags` register, and checks that flag after the instruction's execution. Currently the tool only logs the instruction address for output at the end of execution, although this could be extended to report stack traces or operand values. The overhead is proportional to the number of integer instructions that must be checked at runtime. Thus the overhead varies from program to program but appears to be 10-15X for programs with predominantly integer arithmetic.

```

% OVERFLOW    LINE NUMBER
2.3    jobcntl.c:20
      ob >>= pos;

9.1    jobcntl.c:30
      om >>= 1;

9.1    jobcntl.c:344
      oc >>= 1;

9.1    jobcntl.c:311
      oc >>= 1;

9.1    jobcntl.c:295
      ob >>= 1;

9.2    jobcntl.c:17
void SetOneBit(uint64 *s, int32 pos){ uint64 ob = MLB; ob >>= pos; *s |= ob;}

52.0    jobcntl.c:129
      if ( (avp->groupby & tgrpb) == avp->groupby ) {

% OVERFLOW    FUNCTION
2.3    SetOneBit32
9.1    GetRegTupleFromParent
9.1    Mlo32
9.1    NumberOfOnes
9.1    GetRegTupleFromBin64
9.2    SetOneBit
52.0    GetParent

```

TOTAL OVERFLOWS: 11185

Figure 13: Integer overflow results from NAS DC, aggregated by line number and function; the actual source line is included in the line number aggregation

```

double  hypre_Rand()
{
    int  low, high, test;

    high = Seed / q;
    low = Seed % q;
    test = a * low - r * high;          // <-- 1601 overflows here
    if(test > 0)
    {
        Seed = test;
    }
    else
    {
        Seed = test + m;
    }

    return ((double)(Seed) / m);
}

```

Figure 14: Purposeful integer overflow in AMG2006

I ran the tool on the two integer benchmarks from NAS [12]: DC (data cube) and IS (integer sort). For IS, the overhead was 5X and no overflows were detected. For DC, the overhead was 12X, and the tool detected 11,185 individual overflows. Figure 13 shows the output for the DC run. All of the overflows came from one source code file that contains routines for supervising the benchmark thread pool. Examining the actual lines indicated by the analysis, it was obvious that all of these instructions were bitwise operations and were not intended to perform actual arithmetic. In these cases, the overflow is either intentional or harmless.

I also ran the tool on the AMG2006 benchmark [1], which is composed nearly entirely of floating-point operations and only had an instrumentation overhead of around 2%. Running the tool on AMG2006 (solver #1 on the Laplacian data set with a size of 20x20x20), it detected 1601 overflows, all of which occurred on the same line of code in a random number generation routine that purposefully overflows integer variables. See Figure 14 for a listing of this function with the offending line indicated.

Although this implementation is not immediately applicable to floating-point instructions, it is possible to build a similar sort of analysis for detecting floating-point overflow and underflow, as well as occurrences of NaN and Inf values. This sort of tool would aid in debugging the other analyses in my research, as well as be a useful standalone tool.

### 3.3 Framework for Floating-Point Replacement

I have developed a fairly large (24K+ LOC) framework for runtime binary instrumentation of floating-point instructions. This framework is based on the DyninstAPI library [17] for binary code patching and uses XED from the Intel Pin library [51] for instruction parsing. I created my own structure for storing floating-point instruction semantics, and built a framework for analyses based on those semantics. The semantics allow for multiple operations per instruction, with multiple operands per operation. I created an instruction decoder interface and implemented two decoders, one using the InstructionAPI library that comes with DyninstAPI, and the other using the XED library from Pin. The XED-based decoder tends to work better, and I have been using it nearly exclusively since an early stage of my research. I also created an analysis interface that contains basic routines for handling instrumentation insertion and runtime handlers. This analysis interface allows me to develop new types of floating-point analyses fairly quickly with only a minimal amount of duplicated code. This framework provides a powerful and flexible foundation for current and future research in the area of floating-point program analysis.

### 3.4 Whole-program Replacement

There are multiple approaches to instrumenting a program to use single-precision instead of double-precision. This section describes three of these approaches and my initial implementations of them: 1) shadow values with address-based look-ups, 2) shadow values with pointer-based replacement, and 3) in-place down-cast truncation with a signal bit flag. A *shadow value* is a piece of data that corresponds to a location somewhere in memory or on disk, storing some kind of related information about that location. For my research, I use shadow values to maintain alternate-precision copies of the original values. This technique would allow me to compare the values and determine which values needed extra precision and which values would retain accuracy with lower precision. At this point in my research, I focused on whole-program replacement; in other words, instrumenting a program such that the modified program executes entirely in single-precision.

#### 3.4.1 Shadow values with table-based look-ups

One method of implementing shadow values builds a hash table, where the address of each operand serves as a key into the table. The table stores a pointer to a heap-allocated shadow structure containing additional

precisions or information about that variable. This type of analysis is general and allows many types of information to be associated with each floating-point number in a program.

Unfortunately, upon implementation I found the scheme to have two major problems:

1. The hash table proved to be fairly inefficient as the number of floating-point values quickly grew to millions as larger programs ran. The table became unreasonably large and doing look-ups was impractically time-consuming.
2. It is also necessary to trace the data movement of floating-point numbers, since the hash table needs to be updated with the new location. Not all data movement uses specialized floating-point movement instructions, however, and so to ensure correctness this type of analysis would need to instrument *any* potential program construct that could move memory, and check each movement for floating-point data. This would be prohibitively expensive. Static analysis could be used to reduce the amount of work done, but alias problems would probably still prevent it from being practically useful.

### 3.4.2 Shadow values with pointer-based replacement

I also implemented an analysis that replaces the actual floating-point data with a pointer to the shadow structure. This tactic eliminates the overhead of the hash table as well as the need to track all data movement (since the movement does not care whether the bits being moved are actual floating-point values or pointers). The shadow-value structure is generic and extensible, making it relatively simple to create analyses of varying types: single-precision, double-precision, extended-precision, rational, and various combinations.

The major issue with this type of analysis is that it fundamentally changes the semantics of the original program: the machine instructions such as `mov` that used to have “copy-value” semantics now had “copy-reference” semantics. This opened the door to many new hard-to-find bug possibilities, generally involving a sequence of instructions similar to the following:

```
# assume that %rax has previously been initialized
# as a pointer to a shadow entry with the value zero

mov %rbx, %rax          # copy from %rax to %rbx
```

```
add %rax, 4          # add 4 to the value of %rax
add %rbx, 8          # add 8 to the value of %rbx
```

At the end of the sequence in the original program, the registers `%rax` and `%rbx` would contain the values 4 and 8, respectively. After the pointer-replacement instrumentation, however, the first instruction would make both registers alias to the same shadow-value location. Thus, after the sequence, both registers would point to the same value, which would equal 12.

To solve this problem, the analysis allocated a new shadow structure for every new result of an arithmetic operation. Thus, in the above example, the additions in the second and third instructions would create new shadow values to store the results. At the end of the sequence, there would be three unique shadow entries: 1) the original entry for `%rax` containing the value zero and no longer referenced by either register, 2) a new location for `%rax` containing the value 4, and 3) a new location for `%rbx` containing the value 8. This partially restores the original semantics by essentially making shadow entries into constants, allowing us to continue to ignore data movement.

However, this scheme resulted in many expensive calls to the memory allocation routine, and quickly used up all memory for anything but the smallest examples. To alleviate the memory usage, I added an automatic garbage collector to clean up unreferenced shadow values. This worked because the floating-point values themselves were pointers to these values, so when a value was discarded, the pointer was no longer present to the garbage collector during its marking pass. Unfortunately, all of these workarounds created an environment where it was difficult to debug errors and where the overhead was extremely high (200-500X or higher). Development was slow and the actual analysis was too inefficient for practical use.

### 3.4.3 In-place down-cast truncation with bit flag

The third (and most successful thus far) type of analysis was originally motivated by the desire to simplify and speed up the analysis. This was done by conceding a major feature: the ability to up-cast replacements. By only allowing downward-casting, narrowing replacements (e.g., double-precision to single-precision), the analysis can store the new value in the same location as the old value. The remaining bits are set to a specific bit pattern to indicate to the analysis that this value has been replaced. For 64-bit double-precision values, the tool stores the 32-bit down-cast single-precision value in the lower 32 bits of the original location, and

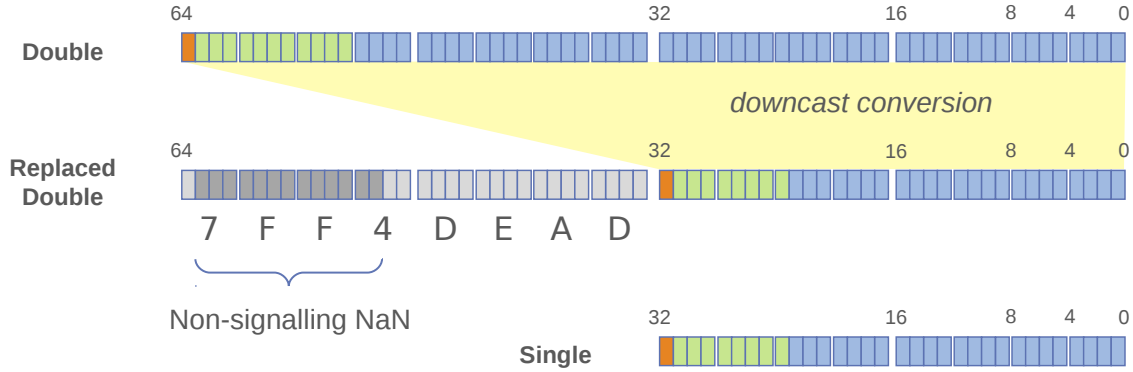


Figure 15: In-place down-cast replacement

then stores the flag `0x7FF4DEAD`<sup>1</sup> in the higher 32-bits. Figure 15 illustrates this process.

Although this method does not allow up-casts, the most common use-case for a system like mine is to test for down-cast possibilities to improve performance, as recommended by the high-performance computing community roadmap for exascale computing [22] and the various mixed-precision research efforts [9, 19, 18]. If in the future it becomes clear that there are advantages to allowing up-casts as well, I plan to build a sort of hybrid analysis that does in-place replacement when down-casting, and pointer-based replacement when up-casting.

This approach to the analysis has proved to be much easier to debug, and the implementation is much more amenable to optimization. In fact, with the simplified handling and the lack of a call to memory allocation, the tool can insert a streamlined “binary blob” of machine code instructions, greatly reducing the instrumentation overhead. These machine code instructions are generated by a simple mini-compiler, which generates “snippets” of code for every double-precision floating-point instruction. This generator has routines for building flag tests and for re-writing instruction opcodes in lower precisions. Figure 16 shows the template for these snippets.

Since most of the operations are integer, the snippets impose a minimal overhead, and the down-cast operation is performed only when the input has not already been replaced. In order to avoid hard-to-find synchronization bugs or writing to unwritable memory, the analysis actually copies any memory operands into a temporary register, and modifies the replaced instruction to use only register operands. Figure 17

<sup>1</sup>This value was chosen for two reasons. The first four hex digits (`0x7FF4`) encode a non-signaling NaN, ensuring that the program will never silently propagate incorrect values. The second four hex digits (`0xDEAD`) form a common human-readable value that is easy to spot in a hex dump.

```

push %rax
push %rbx

<for each input operand>
  <copy input into %rax>

  mov %rbx, 0xffffffff00000000
  and %rax, %rbx                                # extract high 32 bits
  mov %rbx, 0x7ff4dead00000000
  test %rax, %rbx                               # check for flag
  je next                                       # skip if already replaced

  <copy input into %rax>
  cvtsd2ss %rax, %rax                          # down-cast value
  or %rax, %rbx                                # set flag

  <copy %rax back into input>
next:
  <next operand>

pop %rbx
pop %rax

<replaced operand>                                # i.e., addss instead of addsd, etc.

```

Figure 16: Replacement code snippet template

shows the code generated by the analysis for a sample instruction.

I have verified the correctness of the replacement on several NAS benchmarks [12] by manually converting the codes to use single-precision, and comparing the outputs to that of the instrumented version. As a part of this process, I developed a small script (see Appendix B) that attempted an automatic translation of Fortran source code to use single-precision instead of double-precision. However, there were still files I had to tweak by hand, since some source files were generated by auto-configuration scripts. In some cases, while examining the results of the comparison, I found errors in my manual conversion, which needed to be corrected before the results matched. This process demonstrates that the whole-program replacement system is already valuable as a complete automation of a error-prone manual (or semi-automated) process.

The final results were identical, bit-for-bit, indicating that the instrumented versions were performing the exact same operations as the manually-converted versions of the original programs. The manually-converted (single-precision) versions did differ from the original (double-precision) versions, and the difference was greater than the epsilon value that defined correctness for the benchmarks. This result indicates that either the epsilon value is set too low or the benchmarks really cannot be run in entirely single-precision while still passing the verification. It remains to be seen whether particular parts of these benchmarks might be converted to single-precision without causing the benchmark to fail its verification.

In its current state, my implementation of the techniques described in this section can replace all double-precision floating-point instructions with their single-precision equivalents, and the overhead is only 3-10X. I have tested this system on many small examples and on several larger microkernels from the ASC Sequoia benchmark suite [1]. The overhead results for the microkernels are shown in Figure 18.

#### 3.4.4 Development challenges

There are many minor challenges that arose during the implementation of the methods described in the previous sections, but I have either already addressed them or have plans to do so in the future. Here is a list of challenges that I have already addressed:

- Instructions that use `%rip`-relative (static/global variables) or `%rsp`-relative (stack variables) addressing modes will no longer point to the correct locations since these values will be different. I address this by automatically fixing the offsets in the replaced instruction.

```

push    %rax                                # preserve %rax and %rbx
push    %rbx

movq    %xmm2,%rax
mov     $0x7fffffff00000000,%rbx
and     %rbx,%rax
mov     $0x7ff4dead00000000,%rbx
cmp     %rax,%rbx                          # check for flag in %xmm2
je      skip_xmm2_init
cvtss2ss %xmm2,%xmm2                       # down-cast conversion
mov     $0x7ff4dead,%eax                    # set flag and copy back to %xmm2
pinsrd $0x1,%eax,%xmm2

skip_xmm2_init:

movq    %xmm0,%rax
mov     $0x7fffffff00000000,%rbx
and     %rbx,%rax
mov     $0x7ff4dead00000000,%rbx
cmp     %rax,%rbx                          # check for flag in %xmm0
je      skip_xmm0_init
cvtss2ss %xmm0,%xmm0                       # down-cast conversion
mov     $0x7ff4dead,%eax                    # set flag and copy back to %xmm0
pinsrd $0x1,%eax,%xmm0

skip_xmm0_init:

pop     %rbx                                # restore %rax and %rbx
pop     %rax

addss  %xmm0,%xmm2                          # replaced instruction

push   %rax                                # ensure flag is present (can be omitted)
mov    $0x7ff4dead,%eax
pinsrd $0x1,%eax,%xmm2
pop    %rax

```

Figure 17: Replacement code generated for “addsd %xmm0,%xmm2”

AMGmk	4x
CrystalMk	4x
IRSmk	7x
UMTmk	3x
LULESH	4x

Figure 18: Sequoia microkernel overhead for whole-program single-precision replacement

- The operand-initialization routine clobbers `%rflags` in addition to `%rax` and `%rbx`, so I have to save it as well. Unfortunately, the normal `fxsave` and `fxrstor` instructions are extremely slow. Thus, I used a workaround consisting of several pairs of instructions that only save the flags important for program execution.
- The math functions in `libm` cannot handle replaced operands, so I had to create wrappers that would intercept calls to these functions with double-precision numbers, passing any replaced values along to the single-precision version of the function.
- There are a variety of conversion instructions in the SSE instruction set, including conversions between double- and single-precision numbers, as well as conversions between floating-point values and integer numbers. I had to build specialized handlers for some of these conversions to ensure that replacement flags did not propagate beyond double-precision floating-point numbers.
- Compilers tend to use bitwise operations on floating-point numbers whenever possible, usually involving some tweak to the sign bit. For instance, XORing with `0x80000000` will invert the sign bit, while ANDing a number with `0x7fffffff` will take the absolute value. Less obviously, the `BTC` instruction (Bit Test and Complement) can be used to flip the sign bit as well. These integer operations are faster than the corresponding dedicated floating-point instructions. Thus, the analysis must intercept all of these instructions and handle them appropriately. Thankfully, all of the cases I've seen so far involved specialized SSE integer instructions (`PXOR`, `PAND`, etc.) rather than the generic x86 integer operands, so I do not need to instrument every single x86 integer bitwise operation in the program.
- Compilers also generate different code depending on the optimization level. Since my analysis works at the binary level, I take these variations into account, but I have to test programs at the different levels to make sure that the higher levels have not introduced some strange construct that the system cannot currently handle.
- One of the primary motivations for this system is the use of packed instructions, and I had to make sure that the analysis handled packed instructions properly. Mainly, this meant making sure to “patch up” double-precision packed numbers after a replaced instruction, since the single-precision version would destroy the flags in the higher 32 bits of each double-precision location. Figure 19 illustrates this.

Here is a list of challenges that I may need to address in the future:

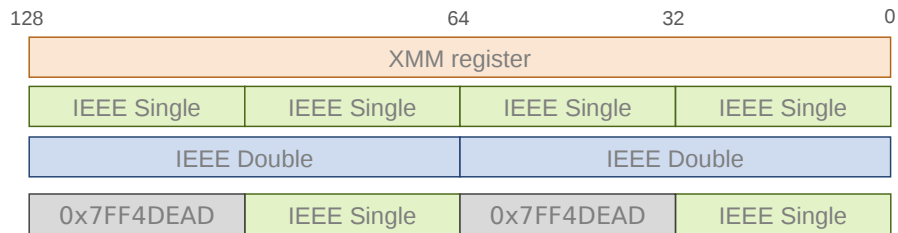


Figure 19: Packed XMM register replacement

- Some codes make use of the non-IEEE 80-bit “long double” format that was introduced in the x87 architecture and standardized in C99. The original motivation for this was to provide a larger format for arithmetic on the CPU, which would provide greater accuracy for intermediate computations, with the end result being rounded to double-precision. Unfortunately, these numbers are hard to work with and pose many difficulties to my analysis. They are currently being deprecated in favor of the IEEE standards and the SSE instruction set. However, it is possible that some legacy codes may require them.
- There are also some IEEE standards that are not so widely-used, such as the 16-bit “half-precision” and the 128-bit “quad-precision.” Half-precision is occasionally implemented in hardware (ex. some ARM chips) and software (ex. POV-Ray) to store values with small dynamic ranges. This provides even better memory bandwidth reductions, although it’s not always faster. Quad-precision is implemented by many software libraries, and may in the coming years be widely supported in hardware.
- Many random-number generation algorithms are width-dependent, and break if my system tries to instrument them. I suspect that there may be heuristics that will allow the analysis to detect this automatically, but for now I provide a mechanism that allows developers to configure the analysis such that these routines are “off-limits” to the mutator.
- Most HPC applications do not use GCC, preferring the highly-optimized output of the Intel or Portland Group compilers. There may also be complications arising from the MPI compiler wrappers. My system will need to support these tool sets at some point.
- Finally, some floating-point codes use various “arcane tricks” to perform operations quickly. For instance, the following operation will approximate the first iteration of a Newton’s method calculation of the square root:

```
val = (1<<29) + (val >> 1) - (1<<22)
```

Likewise, the following operation will approximate the first iteration of a Newton's method calculation of the inverse square root:

```
val = 0x5f3759df - (val >> 1)
```

These tricks, while amusing, are opaque to an analysis such as I propose, and unfortunately there seems to be no good way of working around this. Fortunately, numerical analysis codes do not use such tricks, since the use of hard-coded bitwise approximations belies an inherent disregard for full accuracy. This kind of trick is mostly used by graphics programmers who are willing to compromise on accuracy to extract the maximum performance in rendering code.

## 4 Proposed Work

This section describes the work that I propose to explore my thesis statement. In this work, I will assume that the code being analyzed does not expose pathological floating-point behavior. In other words, if a routine fails to give a sufficiently-accurate answer at a given precision for a given data set, it will also fail at the same precision on a similar data set. This assumption is reasonable for most industrial uses of floating-point numbers.

### 4.1 Expanded Overflow Detector

I will build a simple utility to check for floating-point overflow and underflow, as well as NaN and Inf occurrences. I will do this with an extension to my current floating-point instrumentation framework that will instrument all floating-point operations, checking the resulting values for these conditions. Any such condition will be logged to a file. This utility should be a relatively straightforward extension to the current framework. The overhead will be fairly high, but I such a tool will be helpful in debugging the other work that I plan to do.

## 4.2 Dynamic Range Analysis

I will build analysis mode for my system that will analyze the dynamic range for various instructions at runtime. This analysis will track and report the lowest and highest values that each instruction stores during an execution. It will work by instrumenting each instruction with a small snippet that extracts the result of the operation and checks it against the current min/max values for that instruction. This comparison will inform the search framework discussed in later sections by pointing out operations with low dynamic ranges, since such operations are good candidates for lower-precision replacement.

This mode will be similar to the FloatWatch analysis discussed in a previous section [15], but it will not require Valgrind.

## 4.3 Mixed-precision Replacement Framework

I will extend the current whole-program replacement analysis by adding the ability to up-cast from replaced double-precision as well as down-cast. At every instruction that is instrumented, the analysis will check an oracle (described below) to determine whether the instruction should be executed in double- or single-precision. The inserted instrumentation will up-cast or down-cast the operands if necessary and then replace the opcode and fix the resulting value as needed. This analysis is a relatively simple extension to the current framework, but it will provide the ability to simulate mixed-precision replacement analyses.

The core configuration model will be a series of mappings  $p \rightarrow \{single, double, ignore\}$  involving all  $p \in P_d$ , where  $P_d$  is the set of all double-precision instructions in program  $P$ . Since there are natural parent-child containment aggregations in program structure (ex. instructions are contained within basic blocks, which are contained within functions), the configuration will allow decisions to be made about these aggregate structures, overriding any decisions about child members.

I plan to use a simple, human-readable exchange file format (ex. Figure 20) to store these configurations. The file will be plain text and will list the program’s functions (“FUNC”), basic blocks (“BBLK”), and instructions (“INSN”), using indentation to aid human readers in interpreting this list. The initial list of these structures is easily generated using a simple static analysis that traverses the program’s control flow graph. The first column will contain flags such as “d” (double-precision), “s” (single-precision), or “i” (ignore) that will control the precision of that code structure during instrumentation. The format will allow

```

FUNC01: main()
  BBLK01
s    INSN01: 0x6f45ce "addsd %xmm1, %xmm0"
d    INSN02: 0x6f45d7 "mulsd %xmm2, %xmm1"
s    INSN03: 0x6f45da "subsd %xmm1, %xmm0"
d    INSN04: 0x6f45e8 "mulsd %xmm2, %xmm1"

FUNC02: solve()
  BBLK02
s    INSN05: 0x6f7abe "addsd %xmm1, %xmm0"
s    INSN06: 0x6f7ac6 "addsd %xmm1, %xmm0"
s    INSN07: 0x6f7aca "addsd %xmm1, %xmm0"
d    INSN08: 0x6f7ad3 "mulsd %xmm1, %xmm2"
s    INSN09: 0x6f7ada "addsd %xmm1, %xmm0"
  BBLK03
d    INSN10: 0x6f7aee "mulsd %xmm1, %xmm2"
d    INSN11: 0x6f7af4 "subsd %xmm1, %xmm0"
d    INSN12: 0x6f7af9 "mulsd %xmm1, %xmm2"

s    FUNC03: split()
  BBLK04
s    INSN13: 0x6f8248 "subsd %xmm1, %xmm0"
d    INSN14: 0x6f824c "divsd %xmm2, %xmm1"
d    INSN15: 0x6f824f "divsd %xmm2, %xmm1"

```

Figure 20: Sample configuration file for controlling the replacement analysis; the first column indicates which parts of the program should be run in single-precision (s) and which should be run in double-precision (d)

for easy toggling of the larger aggregate structures like functions. If an aggregate entry has a flag in the first column, it will override any flags specified for its children; if the aggregate entry has no flag, each child will require a flag individually.

Figure 20 shows an example configuration file. In this configuration, certain instructions from each function have been selected for replacement with single precision. In addition, the function `split()` has a single-precision replacement flag, overriding the individual flags of all instructions in that function.

The configuration mapping will serve as the oracle mentioned earlier, and will control the analysis engine as follows:

- If the mapping for  $p_i$  is *single*, then the opcode of  $p_i$  will be replaced with the corresponding single-precision opcode, the inputs will be cast to single-precision before the operation, and the result will be stored as a replaced double-precision number with a flag.
- If the mapping for  $p_i$  is *double*, then opcode of  $p_i$  will not be replaced, the inputs will be cast to double-

precision before the operation, and the result will be stored as a regular double-precision number.

- If the mapping for  $p_i$  is *ignore*, then instruction  $p_i$  will be ignored entirely; this is useful for flagging unusual constructs like random number generation routines.

I will also build a GUI that will display a tree corresponding to the program structure, allowing a developer to tweak a configuration manually without having to edit a lengthy text file. The GUI may allow support for user-friendly aggregate levels that are not explicitly included in the configuration file, such as modules and code regions (by source line).

## 4.4 Automated Search

I will investigate and implement techniques for automatically running an application multiple times, changing the replacement configuration between each run. In this section I describe a proposed system for doing this type of analysis. Figure 21 shows an overview of this system. The inputs to the system will be  $P$  (the target program) and a set  $E$  of error mappings  $v_i \rightarrow \epsilon_i$  (a desired error threshold for each output variable of interest).

The system will perform an initial evaluation run at full double-precision (i.e.,  $p \rightarrow \textit{ignore} \ \forall p \in P_d$ ) to determine baseline values  $v_i \rightarrow b_i$  for all output variables specified by the user. This run will also gather performance data, such as time spent in each function and the number of executions of each floating-point instruction. The system will then build multiple replacement configurations, each of which will consist of a set of precision mappings as described in the previous section. The system will choose these strategies based on the initial evaluation run, and will focus its replacement attempts on parts of the code that contribute the most to the overall runtime, as well as the parts with the lowest dynamic range. I will perform research and experiments to determine ways to build these strategies and how to run them efficiently.

The search process will be constrained by the values from the error mapping  $E$ ; if any output variable  $v_i$  exceeds its corresponding  $b_i$  value by more than  $\epsilon_i$  for a particular replacement configuration, then that configuration is not considered valid. The system will attempt to find a valid replacement configuration that maximizes the number of instructions replaced (i.e., maximize  $|P_r|$  s.t.  $(p_i \in P_d \wedge p_i \rightarrow \textit{single}) \ \forall p_i \in P_r$ ), in order to replace as much of the program as possible with single-precision arithmetic. If possible, the system will use performance data to maximize the reduction in dynamic (runtime) instruction executions, rather

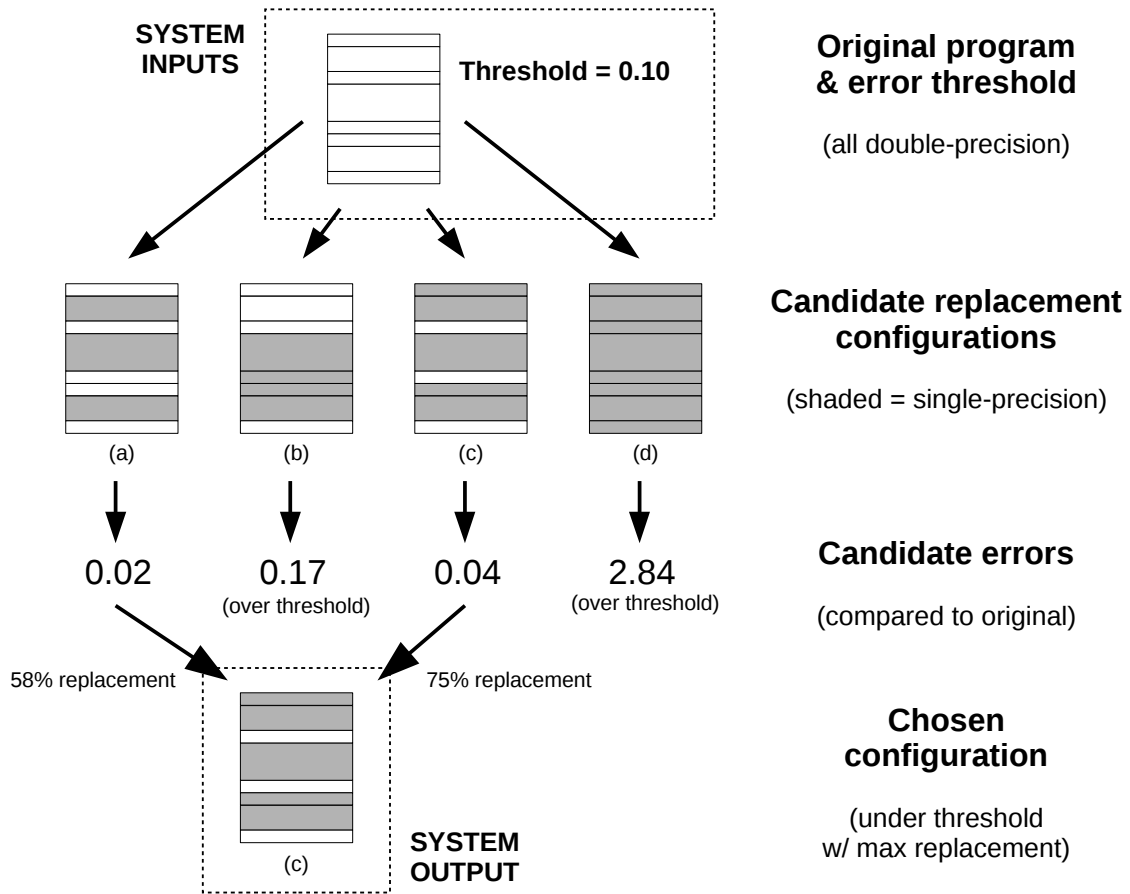


Figure 21: System overview: the system will generate a number of candidate single-precision replacement configurations from the original program, comparing the final results with the original to find the candidate with the most replacement under the given error threshold

than the static instruction count from the binary. The system should also attempt to factor in the cost of data conversion.

Researchers in the field of performance tuning have used variations on simplex methods for automated searches [67, 10] through optimization spaces. Unfortunately, these techniques are intended mainly for integer and real number (quantitative) domains, and the domain here is largely qualitative, with a few categorical choices for each variable (single, double, ignore). Thus, the traditional simplex-based methods may not work well in this context. Since there is no definition of a first or second derivative, other numerical analysis optimization techniques that rely on gradients also will be unhelpful, unless I use the difference in error values between runs as a gradient of those runs. This difference is a rough approximation, however, and makes assumptions about the smoothness of the search space that may not apply in all cases.

Thus, I will investigate and implement alternate search methods, possibly including techniques such as simulated annealing and other stochastic methods [40, 69]. These methods do not require quantitative domains, and may prove to be excellent matches for this problem. This problem also bears resemblance to SAT solvers and optimizers, in the sense that the variables are boolean or pseudo-boolean. It is possible that techniques for optimizing in these contexts [24] could be applied to the precision-level program search space as well. Of course, runtime performance data will also inform the search, and so any algorithm that I use must be able to incorporate performance feedback.

A simpler approach is to perform binary searches on the functions in a program (or instructions in a function). The hypothesis is that there is a relatively small number of program points that must run in double-precision. If some criteria could be used to order program points by their chance of sensitivity, a binary search could narrow down on the desired points quickly. I will explore various criteria (range data, instruction type, execution counts, etc.) for this ordering, to determine which criteria yield the best replacement with the lowest number of executions.

Another field of interest is that of GUI testing. Since exhaustively testing a graphical interface is prohibitively expensive, the software engineering community has developed methods of searching the event space for interesting and effective test cases [55]. This search is done by building an event flow graph to represent the possible GUI events and their interactions, and then traversing that graph to generate test cases. It is possible that a similar type of analysis on the data flow graph of a program could yield insight into replacement strategies for floating-point programs.

I will also explore ways to modify the search to maximize the actual performance gain instead of raw instruction count, since it is quite possible that a configuration that replaces slightly less of the program will exhibit a larger performance gain. For instance, this situation may happen if the configuration replaces a short but frequently-used and time-consuming routine (matrix multiplication or linear system solving, for example) and the other configurations do not. If the replacements of one valid candidate are a strict subset of the replacements of another candidate, then the latter candidate is clearly preferable. However, if a candidate is not a strict subset of another, I will use heuristics or actual performance benchmark runs to determine which of the configurations is preferable.

## 4.5 Validation and Evaluation

I will perform two types of experiments:

1. During development, I will perform tests against synthetic examples and benchmarks to verify that the analysis is working correctly. I have already developed a small suite of tests that I plan to continue using, and I will add more tests as I expand the system. I will also test on well-known benchmarks to verify on a larger scale that the analysis does not perturb the program beyond what is expected with reduced precision. I will do this by manually converting various parts of the benchmarks to single-precision and comparing the results of the modified benchmark to the results of my analysis. I will also implement one or two of the mixed-precision algorithms mentioned in the literature [9], and compare the configuration selected by the automatic search algorithm to the manually-developed one, in terms of error margin and performance gain.
2. After development, I will perform case studies on large benchmarks and numeric codes to evaluate the usefulness and ease of use of the system. I expect to find a variety of results, from programs that cannot be replaced at all with single-precision to those that can be entirely replaced with single-precision, including varying degrees in between. Based on the many past successes with mixed precision [19, 18, 9, 8, 21, 27], I predict that in many cases a large portion of the most time-consuming sections of a program may be performed in single-precision, resulting in a significant (2X+) speedup. The verification process will test this hypothesis.

## 4.6 Optimization

There are several ways that the analysis could be made more efficient. I plan to pursue the following research directions:

- More efficient code generation. There are many opportunities for low-level optimization of the generated instrumentation code. These opportunities range from localized optimizations involving more efficient machine code instructions and idioms to global optimizations such as skipping flag checks that are verifiably unnecessary, or inlining the instrumentation for several consecutive instructions that operate on the same data.
- Optimized search algorithms. As described in previous sections, there is a large body of research and literature about automated search algorithms that could be tapped for optimizing the search step. It is likely that the discrete, qualitative nature of the search space may yield itself well to a specialized form of an existing search algorithm. I plan to explore a wide range of options once I have built the replacement framework, to compare various search algorithms regarding their convergence time as well as the accuracy of the final result.
- Convert data at boundary points. When control entered a replaced section, the analysis could automatically convert all data used by that section to single-precision, and then convert back when the section is exited. This technique would impose a higher overhead at the boundaries between code segments, but would eliminate the need to do flag or replacement testing inside the segment. However, this may require some rather sophisticated static analysis and may only be applicable to certain basic blocks and functions.

## 5 Future Work

This section briefly relates some ideas for future work beyond that proposed in the previous section, showing the potential extensions of this research beyond the specific proposals in the this document.

## 5.1 Environmental Support

Currently, the implementation is limited to SSE x86-64 on Linux-based machines, with a GCC-based compiler. Many current HPC applications, however, run on BlueGene or ARM architectures and/or are compiled using ICC or PGCC (the LLVM compiler infrastructure is also growing in popularity). Floating-point support is currently limited to SSE 3, and could be expanded to include SSE 4 & 5, as well as legacy support for x87 instructions. There may also be unique concerns associated with MPI or OpenMP libraries, and these concerns would need to be addressed in order to perform this kind of analysis on large-scale distributed systems.

## 5.2 Dynamic Configurations

DyninstAPI provides the capability to add and remove instrumentation at runtime, and this could be used to dynamically change the configuration at timestep or node boundaries. This capability could be used in various ways.

One application of dynamic instrumentation would be to use it to test several candidate configurations during a single execution. The program could run in one configuration, with its associated instrumentation, for a given number of cycles or timesteps. Then, the mutator could pause execution, make observations about the error introduced by the current configuration, remove the old instrumentation, add new instrumentation corresponding to a different configuration, and resume execution.

One potential issue would be that any data currently in the system at the time of the reconfiguration might need to be examined and possibly cast to a different precision for the new configuration. This conversion may be difficult to coordinate, especially in a distributed setting, and it would introduce side effect correctness issues. Care would have to be taken to avoid making repeated downward casts. Another option would be to utilize knowledge about the program (perhaps from a previous execution) to perform the reconfigurations at particular “checkpoint” locations to avoid having to down-cast data.

If done carefully, however, dynamic instrumentation may allow some level of online tuning, decreasing the convergence time of the search algorithm. Researchers in the field of performance tuning have had success with this kind of automatic online adaptation [10, 68]. Since I am more concerned with correctness, however, it may be difficult to develop autotuning techniques that do not unacceptably compromise the validity and

precision of the analysis. Once a portion of a program has been executed in single-precision, it is difficult to say with any kind of certainty that the accuracy of later calculations have not been adversely affected, regardless of whether the later instructions were replaced.

### 5.3 Expansion Replacement

The analysis could be made more flexible if it could expand (up-cast) floating-point numbers instead of only truncating them. With a scheme like this, it could replace single-precision numbers with double-precision, for instance. This ability would increase the search space, opening up new possibilities for replacement. It may be that large portions of the program could be converted to single-precision if a single routine that was previously in single-precision could be switched to double-precision or quad-precision.

Unfortunately, this kind of replacement cannot be done in-place, and would require some sort of shadow-value or pointer-replacement analysis. As discussed in the preliminary work section, these types of analysis are difficult to implement and impose a high overhead. If it could be done, however, it would provide far more options for the recommendation engine.

As mentioned earlier, there is a possibility that a sort of hybrid analysis could provide the best of both worlds: in-place replacement when down-casting and index-based or pointer-based replacement when up-casting. This analysis would retain the speed and small memory footprint of the current in-place analysis, while expanding the search space to allow greater precisions if desired.

### 5.4 Other Precisions

Some benchmarks and applications (like POV-Ray [12]) use precisions like IEEE half-precision (16 bits). These are implemented in software instead of hardware. Also, there is a standard for quad-precision (128-bit) IEEE arithmetic, and it is possible that we will soon see hardware support for it. Adding the ability to use these precisions would provide more options for the analysis to use in building candidate configurations.

## 5.5 Tool Integration

The techniques and analyses proposed in this document could be integrated with other existing tools, such as performance measurement and data flow analyses, allowing for a more detailed analysis by providing the search engine with more information about potential information loss.

Recent work in static/dynamic variable blame analysis [65] could also provide insight into the data structures most responsible for rounding error. This analysis would allow replacement recommendations based on data structures instead of individual instructions. These kinds of recommendations would be more easily actionable for programmers since it is easier to simply change the data type of a variable than to analyze an instruction (or source line) manually to find the variables that are affected.

## 5.6 Source Modification

After my system makes recommendations about which parts of the program to convert to single-precision, the programmer must still modify the source code by hand to make this conversion permanent for future compilation. It is only after a recompilation that the full effects of the conversion will be seen in performance gains, since the compiler will be able to pack data structures more efficiently and pipeline more arithmetic operations. In fact, there may need to be multiple analysis and performance testing iterations to converge on a global optimum configuration, since the changes that the compiler makes after conversion may impact the numerical accuracy.

A useful extension of my system would be to build a static analysis and modification engine that could take recommendations and automatically modify the source code to implement the optimum configuration. This process is not trivial since my analysis currently operates at the instruction level, while programming alterations must take place at the source code line or data member level. A robust source modification tool would need to combine sophisticated dataflow analysis, compiler knowledge, and runtime information to map machine code instructions to static, stack, and heap data members for type conversion. However, this would enable the final integration of an automatic, end-to-end, precision and performance tuning platform for scientific codes.

## 6 Timeline

Overflow/range detector	1 month
Mixed-precision framework	1 month
Automated search	4-5 months
Case studies	2-3 months
Optimization (interleaved with case studies)	2 months
Writing dissertation	1 month
<b>TOTAL</b>	11-13 months

Paper deadline goals:

- SC'12, April 2012: Mixed precision replacement
- IPDPS'12, October 2012: Automated search
- SC'13, April 2013: Optimization and case studies

## 7 Conclusion

Automated analysis techniques can make recommendations regarding the level of precision that each part of a computer program must use in order to maintain overall accuracy. In this document, I have provided the motivation for such techniques, and explained why previous solutions are not adequate. I have described my previous work in this area, and proposed the development of new analysis techniques using novel methods of instrumentation and analysis, as well as the verification of the usefulness of these methods by means of case studies. This work will produce a significant, novel contribution to the fields of high-performance computing and floating-point error analysis.

## A Reading List

### A.1 Binary Instrumentation

- 1) James R Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In Proceedings of the SIGPLAN 95 Conference on Programming Language Design and Implementation, volume 30 of PLDI 95, pages 291-300. ACM, 1995.
- 2) Bryan Buck and Jeffrey K Hollingsworth. An API for runtime code patching. The International Journal of High Performance Computing Applications, 14:317-329, 2000.
- 3) Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO04), March 2004.
- 4) Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In PLDI 05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 190-200, New York, NY, USA, 2005. ACM.
- 5) Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In SIGPLAN Not., volume 42, pages 89-100. PLDI, ACM, 2007.

### A.2 Floating-Point Error Analysis

- 1) John W. Carr. Error analysis in floating point arithmetic. Communications of the ACM, 2(5):1016, May 1959.
- 2) Paul L. Richman. Automatic error analysis for determining precision. Communications of the ACM, 15(9):813-820, September 1972.
- 3) David Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys, 23(1):5-48, March 1991.

4) Walter Kraemer. A Priori Worst Case Error Bounds for Floating-Point Computations. *IEEE transactions on computers*, 47(7):750-756, 1998.

5) Eric Goubault. Static Analyses of the Precision of Floating-Point Operations. *Static Analysis*, pages 234-259, 2001.

### **A.3 Mixed-Precision Algorithms**

1) Xuejun Hao and Amitabh Varshney. Variable-precision rendering. *Proceedings of the 2001 symposium on Interactive 3D graphics SI3D 01*, pp. 149-158, 2001.

2) Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimire Tomov. Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy. 2007.

3) H. Anzt, B. Rucker, and V. Heuveline. Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms. *Computer Science Research and Development*, vol. 25, no. 3-4, pp. 141-148, 2010.

4) M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi. Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications*, vol. 181, no. 9, p. 30, 2010.

5) D. Goddeke and R. Strzodka. Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid. *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 22-32, 2011.

## B Fortran Precision Conversion Script

```
# Converts Fortran programs from double precision to single precision ("real").

for f in $@; do

    # create backup
    cp $f $f.old

    # convert data types
    sed -i -e 's/double precision/real/g' $f

    # convert function calls
    sed -i -e 's/dble\ *(/real(/g' $f
    sed -i -e 's/dsqrt\ *(/sqrt(/g' $f
    sed -i -e 's/dabs\ *(/abs(/g' $f
    sed -i -e 's/dlog\ *(/log(/g' $f
    sed -i -e 's/dexp\ *(/exp(/g' $f
    sed -i -e 's/dabs\ *(/abs(/g' $f
    sed -i -e 's/dsin\ *(/sin(/g' $f
    sed -i -e 's/dcos\ *(/cos(/g' $f
    sed -i -e 's/dtan\ *(/tan(/g' $f
    <other functions omitted>

    # convert constants
    sed -i -e 's/\([0-9]\{1,\}\.[0-9]*\)d/\1e/g' $f
    sed -i -e 's/\([0-9]\{1,\}\.[0-9]*\)D/\1E/g' $f

done
```

## References

- [1] ASC Sequoia Benchmark Codes. <https://asc.llnl.gov/sequoia/benchmarks/>. Accessed 21 September 2011.
- [2] GNU Multiple Precision Arithmetic Library. <http://www.gmpilib.org/>. Accessed 23 February 2010.
- [3] MPFR Library. <http://www.mpfr.org/>. Accessed 23 February 2010.
- [4] SPEC CPU2006 Benchmark. <http://www.spec.org/cpu2006/>. Accessed 23 February 2010.
- [5] X86 Encoder Decoder. <http://rogue.colorado.edu/Pin/docs/20751/Xed/html/main.html>. Accessed 5 December 2008.
- [6] Oliver Aberth. A precise numerical analysis program. *Communications of the ACM*, 17(9):509–513, September 1974.
- [7] Marcus Vinicius Alvim Andrade, João Luiz Dihl Comba, and Jorge Stolfi. *Affine Arithmetic*, 1994.
- [8] Hartwig Anzt, Björn Rucker, and Vincent Heuveline. Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms. *Computer Science Research and Development*, 25(3-4):141–148, 2010.
- [9] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating Scientific Computations with Mixed Precision Algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2008.
- [10] D H Bailey, J Chame, C Chen, J Dongarra, M Hall, J K Hollingsworth, P Hovland, S Moore, K Seymour, Jaewook Shin, Ananta Tiwari, S Williams, and H You. PERI auto-tuning. *Journal of Physics Conference Series*, 125(012089):6pp, 2008.
- [11] D H Bailey, Y Hida, X S Li, and B Thompson. ARPREC: An arbitrary precision computation package. *Manuscript*, pages 1–8, 2002.
- [12] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. *The NAS Parallel Benchmarks 2.0*, 1995.
- [13] David H. Bailey. Algorithm 719: Multiprecision Translation and Execution of Fortran Programs. *ACM Transactions on Mathematical Software*, 19(3):288–319, 1993.

- [14] D.M. Beazley. SWIG Users Manual. Technical report, University of Utah Technical Report UUCS-98-012 (1998), 1998.
- [15] A.W. Brown, P.H.J. Kelly, and W Luk. Profiling floating point value ranges for reconfigurable implementation. In *Workshop on Reconfigurable Computing, HiPEAC 2007*, 2007.
- [16] A.W. Brown, P.H.J. Kelly, and Wayne Luk. Profile-directed speculative optimization of reconfigurable floating point data paths. In *informal proceedings of the Workshop on Reconfigurable Computing, HiPEAC*, volume 2008. Citeseer, 2008.
- [17] Bryan Buck and Jeffrey K Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000.
- [18] A Buttari, J Dongarra, J Kurzak, J Langou, P Luszczek, and S Tomov. Exploiting Mixed Precision Floating Point Hardware for Scientific Computation. In *High Performance Computing and Grids in Action*. IOS Press, 2008.
- [19] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimire Tomov. Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy. 2007.
- [20] John W. Carr. Error analysis in floating point arithmetic. *Communications of the ACM*, 2(5):10–16, May 1959.
- [21] M A Clark, R Babich, K Barros, R C Brower, and C Rebbi. Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications*, 181(9):30, 2010.
- [22] J Dongarra, P Beckman, T Moore, P Aerts, G Aloisio, J C Andre, D Barkai, J Y Berthou, T Boku, B Braunschweig, F Cappello, B Chapman, A Choudhary, S Dosanjh, T Dunning, S Fiore, A Geist, B Gropp, R Harrison, M Hereld, M Heroux, A Hoisie, K Hotta, Y Ishikawa, F Johnson, S Kale, R Kenway, D Keyes, B Kramer, J Labarta, A Lichnewsky, T Lippert, B Lucas, B Maccabe, S Matsuoka, P Messina, P Michielse, B Mohr, M S Mueller, W E Nagel, H Nakashima, M E Papka, D Reed, M Sato, E Seidel, J Shalf, D Skinner, M Snir, T Sterling, R Stevens, F Streitz, B Sugar, S Sumimoto, W Tang, J Taylor, R Thakur, A Trefethen, M Valero, A Van Der Steen, J Vetter, P Williams, R Wisniewski, and K Yelick. The International Exascale Software Project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.
- [23] Sylvie Putot Eric Goubault Matthieu Martel. Asserting the Precision of Floating-Point Computations: A Simple Abstract Interpreter. *Programming Languages and Systems*, pages 287–306, 2002.

- [24] Fadi A. Aloul. Search techniques for SAT-based Boolean optimization. *Journal of the Franklin Institute*, 4(5):436—447, 2006.
- [25] Claire Fang Fang, Rob A Rutenbar, Markus Püschel, and Tsuhan Chen. Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling. *Proceedings of the 40th conference on Design automation DAC 03*, page 496, 2003.
- [26] Mikito Furuichi, Dave a. May, and Paul J. Tackley. Development of a Stokes flow solver robust to large viscosity jumps using a Schur complement approach with mixed precision arithmetic. *Journal of Computational Physics*, 230(24):8835–8851, October 2011.
- [27] Dominik Göddeke and Robert Strzodka. Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):22–32, 2011.
- [28] Dominik Göddeke, Robert Strzodka, and Stefan Turek. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):221–256, August 2007.
- [29] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.
- [30] Eric Goubault. Static Analyses of the Precision of Floating-Point Operations. *Static Analysis*, pages 234–259, 2001.
- [31] Xuejun Hao and Amitabh Varshney. Variable-precision rendering. *Proceedings of the 2001 symposium on Interactive 3D graphics SI3D 01*, http:149–158, 2001.
- [32] Y He and C H Q Ding. Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. *The Journal of Supercomputing*, 18(3):259–277, 2001.
- [33] Nicholas J Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM Philadelphia, 2002.
- [34] J D Hogg and J A Scott. A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 37(2):1–24, 2010.
- [35] IEEE. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, New York, August 2008.
- [36] Fabienne Jézéquel and Jean-Marie Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933–955, June 2008.

- [37] W Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965.
- [38] William Kahan. The Improbability of Probabilistic Error Analyses for Numerical Computations. Technical Report July 1995, University of California, Berkeley, 1996.
- [39] Toyohisa Kaneko and Bede Liu. On Local Roundoff Errors in Floating-Point Arithmetic. *J. ACM*, 20(3):391–398, 1973.
- [40] S Kirkpatrick, C D Gelatt, and M P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [41] Walter Kraemer. A Priori Worst Case Error Bounds for Floating-Point Computations. *IEEE transactions on computers*, 47(7):750–756, 1998.
- [42] Walter Krämer and Armin Bantle. Automatic Forward Error Analysis for Floating Point Algorithms. *Reliable Computing*, 7(4):321–340, August 2001.
- [43] Timo I Laakso and Leland B Jackson. Bounds for floating-point roundoff noise. *IEEE transactions on circuits and systems*, 41(6):424–426, 1994.
- [44] Michael O Lam, Jeffrey K Hollingsworth, and G W Stewart. Dynamic Floating-Point Cancellation Detection. In *WHIST '11*, 2011.
- [45] John Larson and Ahmed Sameh. Efficient Calculation of the Effects of Roundoff Errors. *ACM Transactions on Mathematical Software*, 4(3):228–236, September 1978.
- [46] John L. Larson, Mary E. Pasternak, and John A. Wisniewski. Algorithm 594: Software for Relative Error Analysis. *ACM Transactions on Mathematical Software*, 9(1):125–130, March 1983.
- [47] James R Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN 95 Conference on Programming Language Design and Implementation*, volume 30 of *PLDI '95*, pages 291–300. ACM, 1995.
- [48] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, March 2004.
- [49] Xiaoye S. Li, Michael C. Martin, Brandon J. Thompson, Teresa Tung, Daniel J. Yoo, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, and Anil Kapur. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, June 2002.

- [50] B Liu and T Kaneko. Error analysis of digital filters realized with floating-point arithmetic. *Proceedings of the IEEE*, 57(10):1735–1747, 1969.
- [51] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.
- [52] Matthieu Martel. Propagation of Roundoff Errors in Finite Precision Computations: A Semantics Approach. *Programming Languages and Systems*, pages 159–186, 2002.
- [53] Matthieu Martel. Semantics-Based Transformation of Arithmetic Expressions. *Static Analysis*, pages 298–314, 2007.
- [54] Matthieu Martel. Program transformation for numerical precision. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 101–110, New York, NY, USA, January 2009. ACM Press.
- [55] Atif M Memon. An event-flow model of GUI-based applications for testing. *Software Testing Verification and Reliability*, 17(3):137–157, 2007.
- [56] Webb Miller. Software for roundoff analysis. *ACM Transactions on Mathematical Software*, 1(2):108–128, 1975.
- [57] Webb Miller and David Spooner. Software for roundoff analysis, II. *ACM Transactions on Mathematical Software*, 4(4):369–387, December 1978.
- [58] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004.
- [59] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *SIGPLAN Not.*, volume 42, pages 89–100. PLDI, ACM, 2007.
- [60] R. L. Ashenurst and N. Metropolis. Error Estimation in Computer Calculation. *The American Mathematical Monthly, Part 2: Computers and Computing*, 72(2):47—58, 1965.
- [61] Paul L. Richman. Automatic error analysis for determining precision. *Communications of the ACM*, 15(9):813–820, September 1972.
- [62] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. *Proceedings of the 20th annual international conference on Supercomputing ICS 06*, pages 324–334, 2006.

- [63] Robert Strzodka and Dominik Goddeke. Mixed Precision Methods for Convergent Iterative Schemes. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures, May 2006*, 2006.
- [64] Robert Strzodka and Dominik Goddeke. Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. *IEEE Proceedings on Field-Programmable Custom Computing*, 2006.
- [65] Nick Rutar and Jeffrey K Hollingsworth. Assigning Blame: Mapping Performance to High Level Parallel Programming Abstractions. In *The 15th International Euro-Par Conference (Euro-Par '09)*, pages 21–32, 2009.
- [66] Walter Schreppers and Annie Cuyt. Algorithm 871: A C/C++ Precompiler for Autogeneration of Multiprecision Programs. *ACM Transactions on Mathematical Software*, 34(1):1–20, January 2008.
- [67] Vahid Tabatabaee, Ananta Tiwari, and Jeffrey K Hollingsworth. Parallel Parameter Tuning for Applications with Performance Variability. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05*, pages 57—. IEEE Computer Society, 2005.
- [68] Ananta Tiwari, Vahid Tabatabaee, and Jeffrey K Hollingsworth. Tuning parallel applications in parallel. *Parallel Computing*, 35(8-9):475–492, 2009.
- [69] V Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.
- [70] J H Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Inc., 1964.
- [71] J H Wilkinson. Error analysis revisited. *IMA Bulletin*, 22(11/12):192–200, 1986.