ABSTRACT

Title of dissertation: Automated Floating-Point Precision Analysis

Michael O. Lam, Doctor of Philosophy, 2014

Dissertation directed by: Professor Jeffrey K. Hollingsworth
Department of Computer Science

As scientific computation continues to scale upward, correct and efficient use of floating-point arithmetic is crucially important. Users of floating-point arithmetic encounter many problems, including rounding error, cancellation, and a tradeoff between performance and accuracy. This dissertation addresses these issues by introducing techniques for automated floating-point precision analysis. The contributions include a software framework that enables floating-point program analysis at the binary level, as well as specific techniques for cancellation detection, mixed-precision configuration, and reduced-precision sensitivity analysis. This work demonstrates that automated, practical techniques can provide insights regarding floating-point behavior as well as guidance towards acceptable precision level reduction. The tools and techniques in this dissertation represent novel contributions to the fields of high performance computing and program analysis, and serve as the first major step towards the larger vision of automated floating-point precision and performance tuning.

Automated Floating-Point Precision Analysis

by

Michael O. Lam

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Advisory Committee:
Professor Jeffrey K. Hollingsworth, Chair and Advisor
Dr. Bronis R. de Supinski (Lawrence Livermore National Laboratory)
Professor Atif M. Memon
Professor Martin C. Rabenhorst, Dean's Representative
Professor Alan L. Sussman

To my wife Lindsay,

and to my parents Neil and Alice.

Soli Deo gloria.

Acknowledgments

First and foremost, I honor God my Father and my Lord. I live to glorify him and enjoy all of his creations, especially the field of computer science. This work is a testimony to the life and abilities he has given to me.

I gratefully recognize my dissertation advisor, Jeff Hollingsworth. Thank you for your invaluable guidance and advice over the years; I have learned much from you. I look forward to collaborating as a colleague for many more years to come. I also want to thank the members of my dissertation committee (especially my LLNL project lead Bronis de Supinski) as well as my undergraduate academic and research advisors: David Bernstein, Daniela Raicu, Elizabeth Adams, and Charles Abzug. You all provided mentoring and wise advice that ultimately enabled me to complete this dissertation.

I am also very thankful for the support and camaraderie of all my fellow lab members at UMD, especially Ray Chen, Ananta Tiwari, Nick Rutar, Tugrul Ince, and Geoff Stoker. I will sorely miss the weekly lunch outings, the times spent together outside the lab, and the mutual encouragement that helped me persevere over years of course work and research. I hope that we will meet again often in the future.

I thankfully acknowledge my church family, both at Wallace Presbyterian Church in College Park, Maryland, as well as at the Evangelical Presbyterian Church

in Elkton, Virginia. Many of you have prayed for me and ministered to me in various ways as I have worked and studied these past years. You are too numerous to mention here by name, but I pray that God will bless you all immensely for your faithfulness.

I cannot thank my family enough for their love and support. I thank my dad Neil for being a quiet inspiration to me in all areas of life, and I thank my mom Alice for challenging me always to aspire to the highest level of achievement. I also thank my father- and mother-in-law Barry and Susan for their love and support over the past few years. Further, I acknowledge my siblings: Sarah, Aaron, Rachel, Nathan, and Hannah, as well as my brother-in-law Corey; thank you all for the love and care you've shown me.

Finally, I thank my amazing wife Lindsay. You supported me through the most challenging years of my time in graduate school; you were a constant source of encouragement and support. As I close this chapter of my life and prepare to move on to the next, I am profoundly grateful to have you by my side. Together we will see what kind of life God has in store for us. Thank you for choosing to explore it with me.

Table of Contents

List of Figures

Chapter 1

Introduction

Floating-point arithmetic is a technology used in a wide range of computer applications, from high-end scientific computation to consumer video games. As scientific computation continues to scale upward to larger systems and as computing becomes further integrated into everyday life, correct and efficient use of floating-point arithmetic is crucially important. However, the approximate nature of floating-point arithmetic ensures that rounding error will be an issue in every computer program that uses such arithmetic. This rounding error manifests with many nuances and caveats that programmers find difficult to address. This often prompts programmers to use a numerical precision that is unnecessarily high, which decreases performance. This unnecessary precision is particularly concerning in high-performance computing (HPC), where the consequences of such decisions are more severe both for correctness and for performance. This dissertation addresses these issues by proposing and implementing automated techniques for floating-point analysis, with the goal of aiding developers in the creation of efficient and accurate floating-point programs.

There has been extensive research in the area of general numerical analysis of floating-point error. However, much of this work is difficult to apply for programmers who do not have a numerical analysis background, and many static analysis techniques are overly conservative because they overestimate the effect of round-

ing errors. In addition, many studies have shown that mixed-precision algorithms (using both single- and double-precision arithmetic) have great potential to speed up computation without sacrificing accuracy. Automated runtime techniques can inform developers regarding various floating-point behaviors; these insights guide further development. Unfortunately, there are few existing tools and techniques for performing such analysis. This dissertation improves the state of the field by developing techniques for dynamic floating-point analysis and by laying the groundwork for further research and tool development.

Our approach is a practical one based on runtime analysis; i.e., running an analysis simultaneously with the target application. Runtime-based analysis allows our analysis to examine floating-point behaviors that only manifest with particular data sets or particular computation sequences. For example, floating-point accumulation is highly sensitive to the relative ordering of the data values being summed; adding the smallest numbers in a sequence first leads to a more accurate result than adding the larger numbers first. This non-associativity is a runtime behavior that is usually opaque to static or compile-time analyses. Additionally, our techniques work at the binary level; i.e., they analyze the machine code instructions that are directly executed on the target hardware. Working at the binary level allows us to incorporate compiler effects; e.g., a compiler may reorder operations during its optimization phase, which may affect floating-point behavior. Binary-level analysis also allows us to analyze highly optimized binaries as well as closed-source shared libraries. Our

approach provides the first runtime-based, floating-point-centric program analysis framework.

We analyze a target binary executable program, parsing the floating-point semantics of all instructions before inserting instrumentation or performing a full modification of instruction semantics. These instrumentation routines and modifications are rewritten into a new binary executable, which can be run in a manner identical to the original one. After running, the system produces various log files for which we have developed graphical user interface (GUI) viewers. These results provide insight and direction for floating-point development. No other contemporary system provides the analysis capabilities and insight generation as the system described in this dissertation.

In particular, this dissertation makes four major contributions:

1. We propose and implement a generic framework: the Configurable Runtime Analysis for Floating-point Tuning (CRAFT). This framework allows a wide variety of runtime analysis techniques specifically focused on floating-point arithmetic. The framework is based on binary parsing, instrumentation, and modification; it also provides generic configuration and logging capabilities. We show the effectiveness of this framework by building simple floating-point analyses for instruction counting and Not-a-Number (NaN) detection. We also replicate a previous effort to track the range of floating-point values. The remainder of the dissertation builds on this framework for more specialized

analyses.

2. We propose and implement techniques for cancellation detection. Cancellation is a loss of precision that can compromise future calculations. This dissertation describes techniques for detecting, aggregating, and reporting cancellation events during a program's execution. This information can inform a developer's decisions regarding algorithm choice. We evaluate these techniques by applying them to several example programs and showing how they can provide insights that were previously unavailable.

3. We propose and implement techniques for mixed-precision configuration implementation. A mixed-precision configuration allows some portions of a double-precision program's instructions to be replaced with the corresponding single-precision instructions, while other portions are still executed in double-precision arithmetic. These techniques allow developers to prototype mixed-precision configurations quickly. Additionally, we propose and implement a search routine to identify automatically the portions of a program must be run in double-precision arithmetic, leaving the rest to be run in single-precision arithmetic. This search technique allows developers to target their mixed-precision development efforts more effectively. We evaluate these techniques by applying them to various benchmarks and applications, demonstrating the insights and recommendations that they provide. In one case, we were able to achieve a 2x speedup based on the results of our analysis.

4. We propose and implement techniques for generalized floating-point sensitivity analysis. These techniques are based on truncating existing double-precision numbers in memory during execution to simulate varying levels of reduced precision. The level of precision can be independently adjusted for every instruction in a program. We also propose an automated search routine for these techniques, allowing the system to determine automatically the minimum level of precision necessary for each part of a program independently. The results of this search inform the developer regarding nuances in floating-point sensitivity, and allows them to focus their efforts to reduce the precision requirements of their application. We evaluate these techniques by benchmarking them to show viability in actual development, and by applying them to various benchmarks and applications.

The general thesis statement for this dissertation is as follows:

Automated runtime analysis techniques can inform application developers regarding floating-point behavior, and can provide insights to guide developers towards reducing precision with minimal impact on accuracy.

Chapter 2

Background

2.1 Overview

This chapter provides background information on floating-point representation, rounding error, and cancellation. These concepts serve as the basis for understanding the context of our work, including the relevant technical background on floating-point arithmetic as well as the problems associated with it. This chapter concludes with a few real-life examples of rounding error and its sometimes tragic consequences.

2.2 IEEE floating-point representation

"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 (b), 2) an exponent field (e), and 3) a fractional significand (s). The significand is also sometimes called the "mantissa." The stored value is $(-1)^b \cdot s \cdot 2^e$. "Machine epsilon" (ϵ or MEps) is a measure of the maximum relative error for a given floating-point representation, and is dependent on the value of the base and the number of bits allocated for the significand. Floating-point arithmetic was first used in computers in the early 1940s, and was standardized by IEEE in 1985, with the latest revision approved in 2008 [42]. The IEEE standard provides for different levels of precision

Name	Bits	Exp	Sig	Dec	MEps
IEEE Half	16	5	10+1	3.311	9.77e-04
IEEE Single	32	8	23+1	7.225	1.19e-07
IEEE Double	64	11	52+1	15.955	2.22e-16
C99 Long Double	80	15	64+0	19.266	1.08e-19
IEEE Quad	128	15	112+1	34.016	1.93e-34

```
Columns:
  Bits: total field width
  Exp:  # of bits for exponent
  Sig:  # of bits for significand = <explicit>+<implicit>
  Dec:  decimal digits  = log10(2^Sig)
  MEps: machine epsilon = b^(-(p-1))   = b^(1-p)
```

Figure 2.1: IEEE standard formats

by varying the field width, with the most common widths being 32 bits ("single" precision) and 64 bits ("double" precision). The C99 and IEEE standards also provide for an extended 80-bit format. See Figure 2.1 for a table of information regarding all of these formats. See Figure 2.2 for a graphical representation of the single- and double-precision formats.

Double-precision arithmetic will generally result in more accurate computations than single-precision arithmetic, 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 dramatically reduced opportunity for parallelization. One example is the x86/SSE 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 in single-precision. Fi-

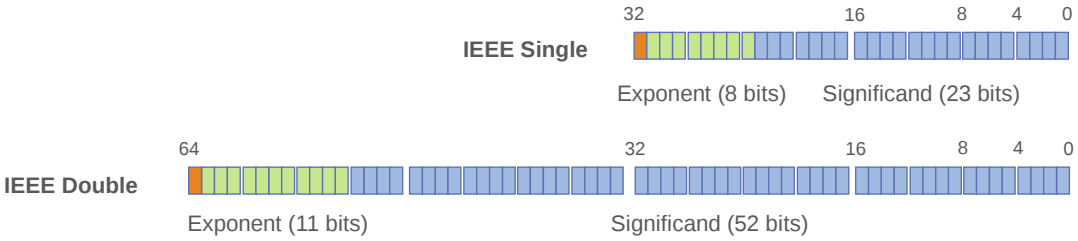


Figure 2.2: IEEE single- and double-precision formats

nally, some architectures even impose a higher cycle count (and thus energy cost) for each arithmetic operation in double-precision. In practice, researchers have reported [33] that single-precision calculations can be 2.5 times faster than corresponding double-precision calculations, because of the various factors described above.

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 [30]. Thus, application developers have powerful incentives to use a lower precision wherever possible, as long as it does not compromise overall accuracy. In addition, long-running computations may encounter numerical accuracy issues not seen at shorter scales [39], providing even more impetus for an analysis solution that accounts for these runtime effects.

Opcode	Operands
ADDSS	Add two scalar single-precision values
ADDPS	Add four sets of packed single-precision values
ADDSD	Add two scalar double-precision values
ADDPD	Add two sets of packed double-precision values

Figure 2.3: SSE opcodes for addition

2.3 SSE floating-point arithmetic

Streaming SIMD Extensions (SSE) is the primary instruction set for IEEE floating-point arithmetic on the x86 and x86_64 architectures [8]. Intel designed the SSE instruction set in the 1990s as an alternative to AMD's 3DNow! instruction set, which has since been discontinued. Before the design of SSE, the x87 instruction set used 80-bit floating-point registers to implement stack-based arithmetic; that instruction set is now rarely used.

Contemporary implementations of SSE floating-point arithmetic use specialized 128-bit registers: eight registers on 32-bit architectures and sixteen registers on 64-bit architectures. These registers can each hold four single-precision values or two double-precision values. The registers themselves can also be used for integer arithmetic, although this dissertation does not address such operations.

The instruction set includes a variety of unary (e.g., square root, reciprocal) and binary (e.g., add, multiply, min, max) mathematical operations. Most instructions are provided in both single- and double-precision versions, as well as scalar and packed variants. Thus, each mathematical operation generally has four possible opcodes. Figure 2.3 shows the SSE opcodes for the addition operation. Most of

these instructions take two operands, and the result is stored in one of the inputs (e.g., $a = a + b$ or $b = a + b$). Our techniques handle all four types of opcodes.

To compare values, the SSE instruction set provides various comparison opcodes that compare two input operands and encode the results in an output operand using all 1s (true) or all 0s (false). The SSE instruction set also provides a variety of data movement instructions, providing the ability to rearrange values in a packed register as well as between two registers or memory operands. Most of these movement instructions do not modify the values as they are moved, although a few of them also do upcast or downcast operations. Our instrumentation and replacement techniques handle all of these comparison, movement, and conversion instructions appropriately.

The SSE instruction set also includes a set of bitwise operations; these come in variants similar to regular operations, such as ANDSS/ANDPS, as well as in variants that operate on entire registers, such as PAND. Some of our techniques must analyze all of the bitwise operations, even if they are the latter variant, because these are sometimes used in conjunction with special binary values to manipulate floating-point numbers. In particular, floating-point numbers can be negated by XOR'ing the entire 32- or 64-bit field with a constant that has a 1 in the highest bit (the sign bit) and zeroes in the rest of the field. Alternatively, the absolute value of a floating-point number can be taken by AND'ing the field with a constant that has a 0 in the highest bit and ones in the rest of the field.

In recent years, some groups have proposed extensions or additions to the SSE instruction set. One of these is the Fused Multiply-Add (FMA) extension, which adds a new opcode that does a multiplication and an addition in a single instruction cycle (e.g., $d = a + b \cdot c$). Other proposed additions include the Advanced Vector eXtensions (AVX), which widen the register width to 256 or 512 bits, providing more SIMD parallelism. The AVX proposal also adds 3-operand instructions, allowing for non-destructive encodings (e.g., $c = a + b$ instead of $a = a + b$). This dissertation does not analyze such additions to the SSE instruction set, although it could easily be extended to include them as long as the underlying tools (Dyninst and XED2) supported them.

2.4 Rounding error

The pitfalls of floating-point representations are numerous and have been extensively studied in the decades since its adoption [35, 40, 46, 73, 75]. The fundamental problem is 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 propagates in ways that can severely compromise the overall calculation. Figures 2.4 and 2.5 show 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. Much of the historical research in floating-point error analysis is discussed in

```
/**
 * sum1.c
 *
 * Roundoff error example. Accumulates 1,000,000 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 2.4: Roundoff error demonstration code

Chapter 3.

2.5 Cancellation

Numerical cancellation occurs when an instruction subtracts two numbers that are of similar magnitude, or when an instruction adds two such numbers with opposite signs. The identical digits are "canceled," and the resulting number has fewer

```
sumf:   991.14154052734375              (single-precision: 32 bits)
sumd:   999.99999998326507011           (double-precision: 64 bits)
sumld: 1000.0000000000008743            (extended-precision: 80 bits)
```

Figure 2.5: Roundoff error demonstration results in varying precisions; the correct answer is exactly 1,000

```
  2.491264  (7)            1.613647  (7)
- 2.491252  (7)          - 1.613647  (7)
  0.000012  (2)            0.000000  (0)
      (a)                      (b)
```

Figure 2.6: Examples of cancellation

significant digits than either of the operands. Figure 2.6 shows several examples of numerical operations that illustrate cancellation. 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 phenomenon is called "complete" or "catastrophic" cancellation.

Cancellation may seem innocuous; after all, the answer is correct. However, one must consider what may happen if the two numbers were not truly 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 than expected. In the worst case (a zero factor from complete cancellation), the result will be 100% incorrect. In this way, cancellation serves as a warning that a loss of significant digits has occurred, which may be a symptom of

undesired rounding error.

2.6 Real-life examples

There have been several real-world incidents involving rounding error, such as the Patriot missile failure in the early 1990s [26] and the Vancouver stock index slump in the 1970s [40].

In the case of the Patriot missile failure, a radar tracking system encountered an issue calculating the position of incoming Scud missiles. The system stored velocity as a real number and the current system time as an integer number; however, the system also had a 24-bit field limit, meaning that any calculation involving both quantities had to round the time value to convert it to a real number. This loss of precision became more of an issue the longer the unit was operational. After 100 hours of operation, the inaccuracy in time was approximately 0.34 seconds, which was enough to throw off the positional calculation by nearly 700 meters. The tragic result was that an errant missile failed to stop an incoming Scud, which hit an Army barracks and killed 28 Americans.

In the case of the Vancouver stock index slump, a long-running financial calculation was compromised by an issue with floating-point rounding. The index value itself was stored using three decimal digits of precision. After nearly a year, the value was hitting suspiciously low values even though the exchange was apparently doing well otherwise. The issue was tracked down to an errant rounding mode,

which was truncating rather than rounding. After recalculating the value using the proper rounding mode, the final value of the index nearly doubled.

While the motivation for the work described in this dissertation comes from high-performance computing and thus does not directly involve the deadly or high-stakes situations involved in these examples, rounding error can have a severe impact on mission-critical computation taking place at supercomputing centers. Many of these codes deal with mission-critical endeavors, such as nuclear weaponry simulations and climate change forecasting. While a mistake due to rounding error may not directly cause a death while running these codes, an incorrect result could be catastrophic if it leads to a faulty real-life policy decisions. In addition, current high-performance computation requires the investment of millions of machine cores for many hours—an expensive proposition. A failed long-term run could mean the loss of thousands or millions of dollars of computing and personnel resources as the problem is debugged and the program is restarted. Thus, there is much motivation for pursuing better analysis techniques in this area.

Chapter 3

Related Work

3.1 Overview

This chapter describes various related fields of research. We begin by examining traditional static error analysis, which was later extended to interval and affine analysis, before looking at some dynamic runtime techniques. We then examine past work in manual mixed-precision implementations and alternate representations. We also include an overview of binary instrumentation frameworks, one of which forms the basis for our techniques. Finally, we conclude with a brief look at subsequent and concurrent research in the area of automated floating-point analysis.

3.2 Error analysis

The analysis efforts regarding floating-point representation and its accompanying roundoff error initially focused on manual backward and forward error analysis. This field was active as early as 1959 [27], with Wilkinson's seminal work in the area being published in 1964 [74]. This research was continued by others [46, 47, 49, 55] and recently summarized by Goldberg and Higham [35, 40].

Forward error analysis begins at the input and examines how errors are magnified by each operation. The result of a floating-point operation $fl(x_1 \diamond x_2) =$

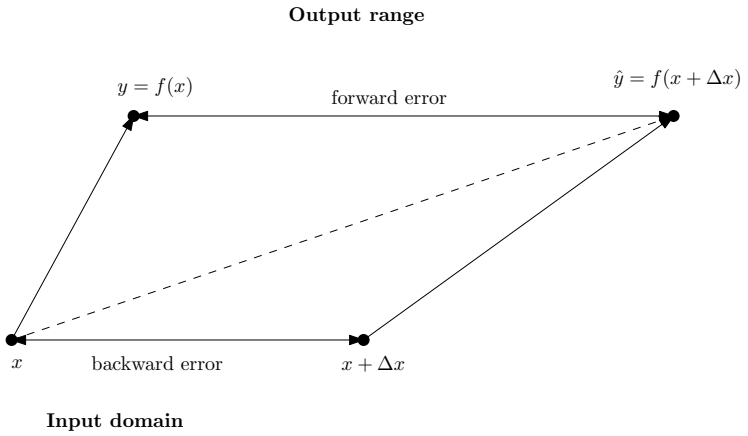


Figure 3.1: Backward and forward error (dashed line indicates floating-point computation)

$(x_1 \diamond x_2)(1 + \epsilon)$, 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 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 3.1 demonstrates these analyses graphically. Higham [40] describes examples of these analyses for a variety of different numerical analysis problems. Unfortunately, the results of these analyses are difficult for a programmer to understand or to apply without extensive training or error analysis background, because

numerical analysis of complex algorithms requires the application of a wide range of techniques as well as familiarity with a large body of mathematics.

3.3 Interval and affine analysis

Researchers have attempted to model the behavior of a program using a technique called "interval arithmetic," [48, 65, 47] 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 [9], and the difficulty of handling intervals containing zero [40]. For instance, 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 $\hat{x}_1 = f_i(x_i)$ will be $[x_1 - \delta_2, x_2 + \epsilon_2]$, where δ and ϵ 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 $(-\infty, +\infty)$, 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. Thus, interval analysis by itself is usually of little value to programmers who are merely interested in the practical behavior.

Interval arithmetic was later improved by Andrade and others [9] 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 ϵ_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.

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

The authors relate simple formulas for resolving operations on affine numbers.

These formulas are composed of simple linear functions involving affine numbers $\hat{x}$ and $\hat{y}$ and scalars $\alpha \in \mathbb{R}$:

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

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

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

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 approximation causes the analysis to be less precise than an optimal analysis.

Other researchers have proposed extensions or variations on interval arithmetic, but few have proved long-lived. Richman [65] 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 [7] 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 [44], 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 [45].

More recently, Goubault and Martel and others [31, 36, 37, 57, 58] 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 [59] describes a system that can effect 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, none of these static analyses are dataset-sensitive, so they will produce conservative results 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.

3.4 Runtime techniques

FloatWatch [21, 22] is a dynamic instrumentation approach that uses the Valgrind framework to monitor the minimum and maximum values that each memory location holds during 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. FloatWatch no longer appears to be in active development. In Section 4.7.3, we show how this type of range-tracking analysis can be implemented using our framework.

Rinard also presents work on fault-tolerant computing with probabilistic accuracy bounds [66]. This effort attempts to measure in a probabilistic model the failure rate of particular portions of a program, called "task blocks." Once the failure rates are known, this system can preemptively 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. The author does not describe how this technique could be extended to floating-point roundoff analysis, where error detection is the core issue.

3.5 Manual mixed precision

In recent years, many researchers [13, 24, 25, 41, 54, 69, 70] have demonstrated that mixed precision (using double-precision in some parts of a program and single-precision in others) can achieve similar results as using only double-precision arithmetic, while being much faster and memory-efficient. They usually present linear solvers (particularly sparse solvers) as examples, showing that most operations can be performed in single-precision. These solvers have been applied to a wide range of problems, including fluid dynamics [11], lattice quantum chromodynamics [28],

1: $LU \leftarrow PA$
2: solve $Ly = Pb$
3: solve $Ux_0 = y$
4: **for** $k = 1, 2, ...$ **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 3.2: Mixed precision algorithm (stars/red indicate double-precision steps)

finite element methods [34], and Stokes flow problems [32]. Often, graphical processing units (GPUs) are cited as the target of these optimizations because of their streaming capabilities [11, 28, 33].

In the iterative algorithm shown in Figure 3.2, for example, only the steps in red (lines 5 and 8) must be executed in double-precision. The authors observe 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% [13].

Researchers in computer graphics have also found that mixed-precision algorithms can improve performance [38]. 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.

In recent work, Jenkins and others [43] describe a novel scheme for reorganizing data structures by numerical significance. Their technique splits up floating-point data structures into striped blocks on byte boundaries. All pieces of corresponding significance are stored consecutively in these memory blocks for storage and I/O, and the original values are re-assembled only when needed for calculation. Thus, the developer can vary the precision of floating-point data during data movement by truncating the lower-precision blocks. In their experiments, they found that some applications can use as few as three bytes (24 bits) of floating-point data and retain an acceptable level of accuracy. This work focused on the I/O implications, however, and did not address the possibility of single-precision arithmetic. Their system also incurs overhead during data re-assembly.

3.6 Alternate representations

Finally, some solutions avoid floating-point representation entirely. For instance, multi-precision libraries allow large or even variable precisions [4, 5, 15, 16]. Some of these libraries also provide a rational representation, storing real numbers using a ratio of integers. Both of these approaches provide higher numerical accuracy at

the cost of performance. Converting a legacy code base to use a numeric library usually also incurs a high cost in developer time, although some researchers have developed automated or semi-automated tools for this purpose [15, 18, 68].

More recently, Le Grand et al. presented a new model for fixed-precision arithmetic in certain molecular dynamics applications [53]. They represented real numbers as 64-bit integers in a fixed format with either 24 bits or 34 bits on the left side of the decimal point, and the remainder of the 64 bits on the right side of the decimal point. They also took advantage of some hardware-specific atomic operations in Kepler GPUs. The new model and implementation resulted in 60–80% higher computational throughput and a reduced memory footprint.

3.7 Binary instrumentation

Binary instrumentation frameworks provide the ability to parse and to instrument a compiled program in binary format, allowing a tool developer to implement analysis techniques in a machine-independent way. These frameworks usually provide a machine-independent representation of program semantics, as well as a scripting language or interface for coding instrumentation routines. The first such framework was the Executable Editing Library (EEL) [50], which was the first general-purpose and cross-platform binary editing framework. EEL provided a system based on C++ for instrumenting programs on various SPARC systems, but does not have support for current architectures and is no longer used. Dyninst [23] is a current

binary instrumentation system, providing interfaces for binary parsing, instrumentation, modification, and code generation on various architectures including x86_64. In this dissertation, we use the capabilities of Dyninst to provide generic binary manipulation, allowing us to focus on implementing floating-point analyses.

Other current binary instrumentation and modification systems include LLVM [51], ROSE [64], Pin [56], Valgrind [62, 63], and PEBIL [52]. LLVM and ROSE are both compiler infrastructures that provide cross-platform abstractions for program transformation and analysis; they provide limited instrumentation support. Pin differs from Dyninst in that it does just-in-time recompilation when new code segments are accessed rather than doing all of the binary modification during a single pass, resulting in a high per-run instrumentation cost. Valgrind differs in that it runs the targeted program in an emulation environment, resulting in a heavyweight and high-overhead framework. PEBIL is a recent development effort that focuses on optimization on a particular platform (Linux on x86/x86_64), although its binary instrumentation techniques are similar to Dyninst's.

None of these frameworks provide the floating-point-specific instrumentation features we needed when we began our research. We chose to use the Dyninst suite as the basis for our work, because of the lower overhead of its approach and its wide variety of analysis components. We use the parsing (ParseAPI), instrumentation (DyninstAPI), binary rewriting (SymtabAPI), and control flow modification (PatchAPI) components of Dyninst.

3.8 Subsequent and concurrent work

Since we began work on our system, Benz et al. [19] have presented another system for floating-point analysis, implemented using Valgrind. Their system supports side-by-side computation in a different precision using shadow variables analysis. The system also collects data related to program slicing. Compared to our earlier implementation (described in Chapter 5), their analysis can more easily identify malignant cancellation. However, the overheads are higher, with several benchmarks experiencing over a 500X slowdown.

In addition, Bao and Zhang [17] have presented a system for detecting and restarting computation in a higher precision based on cancellation detection. They call this process "precision hoisting." Their system tags values with boolean values indicating whether the value has become "substantially inflated," as measured using a cancellation bit threshold test. These values are propagated using a type system, and when an execution is determined to be unstable, it is halted and restarted in a higher precision. They based their implementation on GCC and the GIMPLE intermediate representation. They report an overhead of 3X-23X, which may not be practical at full scale. Unlike our approach, their system does not look for mixed-precision implementations, relying instead on the runtime detect-and-restart approach even in deployed, tuned applications.

Even more recently, Rubio-Ganzalez et al. [67] have built a mixed-precision autotuning system that resembles the one described in Chapter 6. The main differ-

ence is that they work at the source level, creating type configurations rather than instruction configurations. Their system is built on the LLVM compiler framework and requires source code annotations. They do not address whether the running times they report are comparable to those of binaries created by the GCC or Intel compilers that we use. For instance, they appear to have limited support for packed instructions, which our techniques can handle fully, and which can dramatically change the execution profile of an application. For some of their input programs, they use random input data to build approximate representative data sets. They use a variant of delta debugging for their search loop, which allows them a better lower bound on the number of configurations to test. Their verification is performed the same way as in our system, relying on a user-defined routine to check for correctness. Rather than exploring the program to find all parts that can be individually replaced, their search process focuses on finding a large subset of variables that can be replaced simultaneously. This work represents a similar effort in a new context with slightly differing goals. Extending our techniques into a compiler framework is part of the future work discussed in Chapter 8.

Chapter 4

System Architecture

4.1 Overview

This chapter describes our general framework for floating-point-based runtime binary analysis and modification. Our framework is called CRAFT: the Configurable Runtime Analysis for Floating-point Tuning. Figure 4.1 shows the basic workflow. The yellow components are provided by the end user, the blue components are provided by our system, and the green components are generated during analysis.

The process begins when the original binary executable file (hereafter called the "binary") is passed to a Dyninst-based "mutator." This mutator has various capabilities for modifying the binary's machine code to augment or modify the program's behavior, emitting the modified, rewritten binary back to disk. The particular modifications made by the mutator depend on what kind of analysis is desired. This dissertation describes many different analyses, including simpler analyses in Sections 4.7.1, 4.7.2, and 4.7.3, as well as more complex analyses in Chapters 5, 6, and 7. After running the rewritten binary, the system includes a graphical interface for viewing the results. For more complex analyses, the system also provides scripting capabilities for doing automatic tuning searches, streamlining the process of finding optimum floating-point configurations.

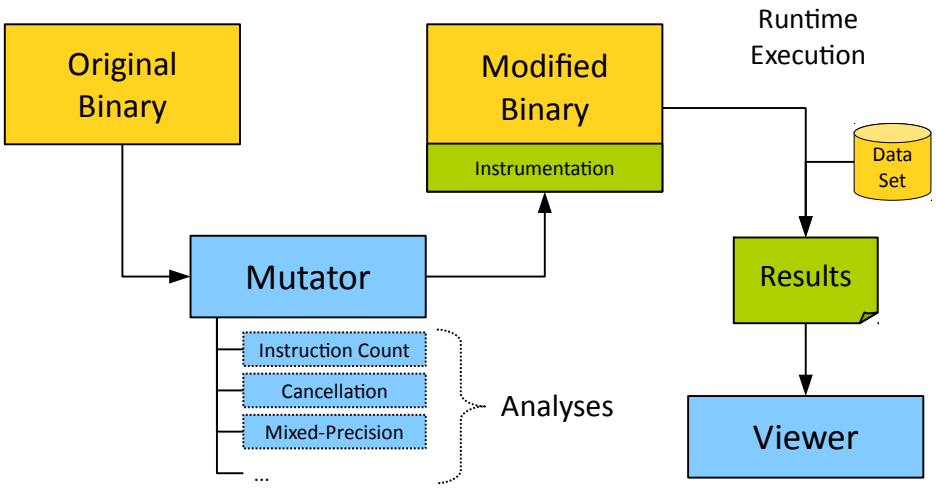


Figure 4.1: Runtime binary analysis overview

CRAFT is implemented in a blend of languages appropriate for the various parts: C/C++ for the main instrumentation and analysis libraries, Java for the graphical interfaces, and Ruby for the search automation scripts. The full system is over 32K lines of code and 5K lines of comments. Roughly 67% of the code is in C/C++, 20% is in Java, 10% is in Ruby, and the final 2–3% is in makefiles and Bash scripts. The system includes a test suite with over 4K lines of code that is capable of running over 300 test combinations with different mutatees, analysis modes and optimization levels. The entire system is stored in a single Git repository and has been made available under the GNU Library General Public License version 3.0 (LGPLv3) license on SourceForge [2].

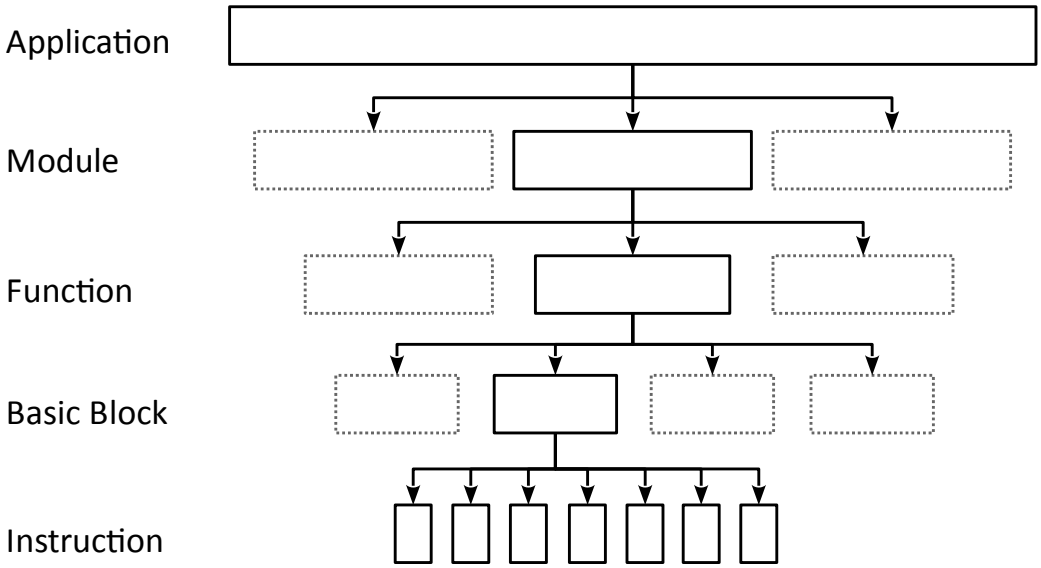


Figure 4.2: Program hierarchy

4.2 Parsing and semantics

We use Dyninst [23] to parse an executable file into its component parts (headers, code, data, etc.). The abstractions provided by Dyninst allow us to view the program's structure as a hierarchy. The hierarchy, depicted in Figure 4.2, consists of a series of subcomponent relationships, including the full application, code modules, functions, basic blocks, and instructions. Most analyses are primarily concerned with the lowest (instruction) level of this hierarchy.

After an application is parsed, our system extracts the semantics of each instruction. To build the instruction semantics, we use a combination of sources. The first source is the Dyninst parse itself, which provides instruction boundaries and raw

instruction bytes. At the time of implementation, however, Dyninst did not provide the detailed information that we needed for floating-point operations. Our second source of semantics is XED2, the instruction encoder and decoder from the Pin instrumentation toolkit [56]. XED2 provided a lightweight framework for extracting opcode and operand information. The final source of semantics is a hard-coded set of semantics for each instruction in the SSE instruction set. XED2 provides some of this information, but in an inconsistent manner. Most instructions from SSE1–4 have custom semantics encoded in our system, and the others (mostly irrelevant to current analyses) have fallback semantics.

The final instruction semantics are stored in an object called Semantics, which is composed of several Operation objects, which in turn contain sets of Operand objects. For unary operations, each set contains a single input Operand and a single output Operand. For binary operations, each set contains two input Operands and a single output Operand. In either case, if the instruction is "packed" (i.e., it operates on an entire 128-bit XMM register), the Operation object contains multiple Operand sets.

Figure 4.3 shows some examples of instructions and their corresponding semantics structures in our system. Each structure begins with the Semantics object for the overall instruction (the figure shows the assembly code representation of that instruction). The Semantics objects serve as the root of a hierarchy that includes Operation, Operand Set, and Operand objects, which are shown on the subsequent

```
Semantics: "sqrtss %xmm0, 0x3a0(rip)"
  Operation: Square Root
    Operand Set
      Operand (Input)  : 0x3a0(rip) memory (bits 0-31)
      Operand (Output) : %xmm0 register (bits 0-31)

Semantics: "addsd %xmm0, %xmm1"
  Operation: Add
    Operand Set
      Operand (Input)  : %xmm0 register (bits 0-63)
      Operand (Input)  : %xmm1 register (bits 0-63)
      Operand (Output) : %xmm0 register (bits 0-63)

Semantics: "mulpd %xmm0, %xmm1"
  Operation: Multiply
    Operand Set
      Operand (Input)  : %xmm0 register (bits 0-63)
      Operand (Input)  : %xmm1 register (bits 0-63)
      Operand (Output) : %xmm0 register (bits 0-63)
    Operand Set
      Operand (Input)  : %xmm0 register (bits 64-127)
      Operand (Input)  : %xmm1 register (bits 64-127)
      Operand (Output) : %xmm0 register (bits 64-127)
```

Figure 4.3: Examples of instruction semantic structures

lines with layers of indentation indicating containment relationships. The first instruction is unary, with a single input and output. The second instruction is binary, with two inputs and a single output. The third instruction is a packed binary instruction, with multiple operand sets.

4.3 Program modification

In this section, we discuss the system's capabilities for modifying a target binary. After parsing the binary, the system can insert instrumentation or do more complex modifications and replacements. This section describes the various aspects of binary

```
BPatch_snippet* buildIncrementSnippet(const char *varname)
{
    BPatch_variableExpr *varExpr = mainImg->findVariable(varname);
    BPatch_snippet *valExpr = new BPatch_arithExpr(
            BPatch_plus,   *varExpr, BPatch_constExpr(1));
    BPatch_snippet *incExpr = new BPatch_arithExpr(
            BPatch_assign, *varExpr, *valExpr);
    return incExpr;
}
```

Figure 4.4: Example of code snippet creation

program modification explored and utilized in this dissertation.

4.3.1 Snippets and binary rewriting

Dyninst provides an API for building "snippets," which are architecture-independent routines that can be inserted into a target binary. Snippets are created with a series of declarations; Dyninst provides a variety of snippet objects and they can be nested. For example, an arithmetic operation snippet may contain several variable snippet, and may itself be contained by an assignment statement. The API also provides control flow snippets, such as conditional if-statements and the ability to call other functions.

Figure 4.4 provides a simple example from our system's source code. The `buildIncrementSnippet` routine takes a variable name as an argument, and creates a Dyninst snippet that increments that variable. To build the snippet, the routine first creates a `BPatch_variableExpr` snippet object that references the variable named by the `varname` argument. The routine then creates a `BPatch_arithExpr`

that increments the variable by a constant value (1), and finally another `BPatch_arithExpr` that assigns the new value back to the old location.

To insert the snippet, Dyninst overwrites the original function with a jump to a newly-allocated space in memory. The code generator then emits the augmented function, containing both the original code and compiled machine code representing the instrumentation snippet.

4.3.2 Instrumentation vs. modification

As described in the previous section, the instrumentation capabilities of Dyninst allow the user to insert predefined snippet code at any point in a program's binary code. Although the snippet code itself may change the program state if desired, the insertion itself does not change program semantics. To ensure that the program's semantics are not inadvertently changed, the code generation system carefully analyzes the context of an inserted snippet and generates code around the snippet to save and load the program state. This state preservation code introduces some overhead in addition to the overhead of the instrumentation, but in practice the system is quite efficient, and the overhead of state preservation is usually far less than the cost of the instrumentation itself.

Dyninst also provides a way to modify the program's code in a way that does not preserve the original program's semantics. These capabilities are provided by the PatchAPI component. The two most important capabilities are 1) the ability to

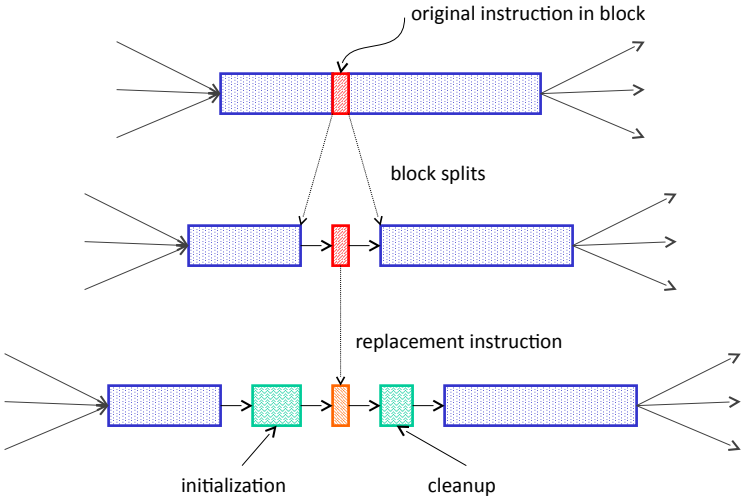


Figure 4.5: Basic block patching

insert raw machine code and 2) the ability to re-arrange basic blocks and the edges between them. The ability to insert raw machine code (sometimes called "binary blobs") rather than Dyninst snippets allows our system to insert highly efficient, custom machine code sequences, exploiting techniques that cannot be encoded in Dyninst snippets. The ability to re-arrange basic blocks allows our system to remove instructions from the original program and to replace them with newly-generated ones. Both of these abilities are crucial to the implementation of the techniques discussed in this dissertation.

4.3.3 Basic block patching

To modify a binary and insert our code snippets, we use Dyninst's CFG-patching API. This API allows us to split the original program's basic blocks at arbitrary

points and to re-arrange the edges between blocks. To insert our code in the place of an instruction, we first split the basic block that contains the instruction into three blocks: 1) any instructions before the original instruction, 2) the original instruction, and 3) any instructions after the original instruction. This segmentation allows us to insert our own code and re-arrange the edges from the surrounding parts of the original basic block to point to our new code instead of the original instruction. Figure 4.5 illustrates this process.

After finishing the patching process, we use Dyninst's binary rewriter to create a new executable with the replaced code. The rewriter can also output modified shared libraries, allowing us to instrument and to modify functions in external dependencies. Thus, we can analyze third-party libraries even if the source code is not available.

4.4 Extensible analysis framework

We designed the system for extensibility. We provide a single Analysis superclass that specifies all of the interface methods required to create a new type of analysis. These methods include three major query and callback routines for instrumentation, as well as corresponding runtime routines. To create a new analysis, a developer must create a new subclass of Analysis, implement these routines, then add some glue code to the main mutator to inform it of the new analysis.

```
bool shouldPreInstrument(Semantics *inst);
```

```
bool shouldPostInstrument(Semantics *inst);
bool shouldReplace(Semantics *inst);
```

These routines should return true if the current analysis can act on the given instruction. The mutator calls these functions for each enabled analysis while iterating over the target program's component tree. The first two functions (pre- and post-instrumentation) indicate that the analysis can add analysis code before or after the given instruction. These additions should not modify the semantics of the original program. The third function (replacement) indicates that the program can replace the instruction entirely. This replacement could potentially modify the semantics of the original program.

```
Snippet buildPreInstrumentation(Semantics *inst, BPatch_addressSpace app);
Snippet buildPostInstrumentation(Semantics *inst, BPatch_addressSpace app);
Snippet buildReplacementCode(Semantics *inst, BPatch_addressSpace app);
```

These routines are called by the mutator to build the instrumentation or modified machine code for insertion into a rewritten binary. These routines are separated from the previous functions so that all decisions can be made about instrumentation and modification before any code is generated. The Dyninst address space parameter is included so that the routines can allocate memory in the rewritten program if desired. These routines should return a Dyninst snippet. This snippet could be a standard API-built snippet as discussed in Section 4.3.1, or a binary blob snippet as discussed in Section 4.3.2. Usually, the pre- and post-instrumentation snippets are standard API snippets, while the replacement snippets are binary blobs. These

routines can also return null, in which case the mutator builds a default snippet, which will make a function call at runtime to one of the runtime handlers:

```
void handlePreInstruction(Semantics *inst);
void handlePostInstruction(Semantics *inst);
void handleReplacement(Semantics *inst);
```

These routines are compiled into a shared library and called at runtime to handle analysis tasks that the developer has chosen not to encode as a Dyninst snippet. Using these library calls incurs high overhead, but allows for quicker development and freedom from snippet API restrictions.

4.5 GUI viewers

The system includes graphical interfaces for viewing results and configuring analysis runs. These interfaces were developed in Java using the Swing toolkit for ease of development and cross-platform support. The latter feature is important because it allows allows users of the system to view results on any platform, without being constrained to the platform on which the analysis took place. This portability is helpful in many HPC contexts where for technical reasons you cannot visualize results on the same platform as the analysis. The interfaces have been tested on Ubuntu Linux, Mac OS X, and Microsoft Windows.

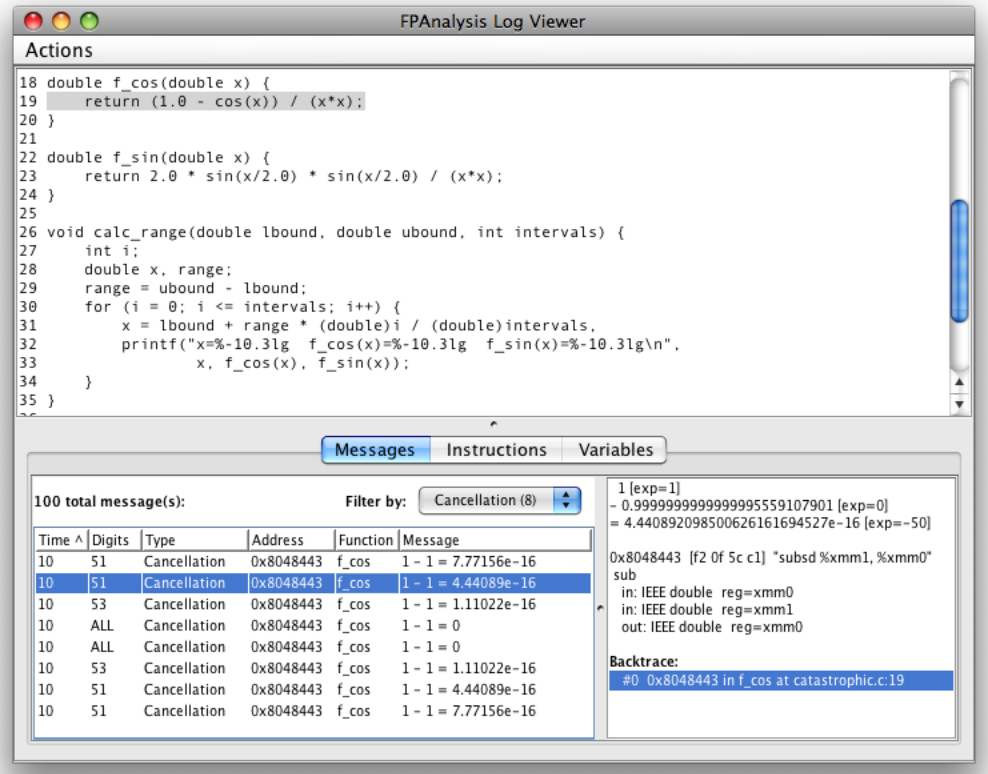


Figure 4.6: Log viewer

4.5.1 Log viewer

Every instrumentation and analysis run generates a log file with information about the results of the run.

Fig. 4.6 shows a screenshot of the log viewer interface. The lower portion displays all events logged during execution. Each event is displayed in the list in the lower-left corner, along with summary information about the event. Clicking on an individual event reveals more information in the lower-right corner and also loads the source code in the top window if the debug information and the source

```
sv_inp=yes
sv_inp_type=config
^  APPLICATION #1: 0 cg.W.x "cg.W.x"
^    MODULE #31: 0x400000 cg.f "cg.f"
^      FUNC #250: 0x404810 conj_grad "conj_grad"
^        BBLK #21841: 0x404bde
^s         INSN #2877: 0x404be6 "pxor xmm1, xmm1  [cg.f:491]"
^s         INSN #2878: 0x404bea "pxor xmm0, xmm0  [cg.f:491]"
^        BBLK #21842: 0x404bf2
^s         INSN #2882: 0x404bfb "mulpd xmm2, xmm2  [cg.f:509]"
^s         INSN #2883: 0x404bff "mulpd xmm3, xmm3  [cg.f:509]"
^s         INSN #2884: 0x404c03 "addpd xmm1, xmm2  [cg.f:509]"
^s         INSN #2885: 0x404c07 "addpd xmm0, xmm3  [cg.f:509]"
^s         INSN #2888: 0x404c19 "mulpd xmm4, xmm4  [cg.f:509]"
^s         INSN #2889: 0x404c1d "mulpd xmm6, xmm6  [cg.f:509]"
^d         INSN #2890: 0x404c21 "addpd xmm1, xmm4  [cg.f:509]"
^d         INSN #2891: 0x404c25 "addpd xmm0, xmm6  [cg.f:509]"
^        BBLK #21843: 0x404c2e
^s         INSN #2892: 0x404c2e "addpd xmm1, xmm0  [cg.f:491]"
^s         INSN #2895: 0x404c39 "addsd xmm1, xmm0  [cg.f:491]"
^        BBLK #21846: 0x404c4d
^s         INSN #2898: 0x404c55 "mulsd xmm0, xmm0  [cg.f:509]"
^d         INSN #2899: 0x404c5c "addsd xmm4, xmm0  [cg.f:509]"
```

Figure 4.7: Excerpt from an example configuration file

files are available. If possible, the interface also highlights the source line that contains the selected instruction. The tab selector in the middle allows access to other information, such as a view of event metrics aggregated by instruction. During instrumentation, the system logs all instrumentation and modifications done to the rewritten binary. During execution, the system can log instruction count information as well as other analysis-specific metrics.

4.5.2 Configuration editor

Some of our analyses require a configuration file that specifies parameters for the analysis. This file may also contain a representation of the target binary, similar to the program structure discussed in Section 4.2 and illustrated in Figure 4.2. The configuration file is stored in plain text for human-readability, and contains a single line for each component with details about the desired instrumentation or modification for that component. The file also contains general instrumentation options in a standard `[key]=[value]` format.

Figure 4.7 contains excerpts from an example configuration file. The first two lines specify general options (in this case, activating configuration-driven inplace mixed-precision analysis). The other lines (all beginning with a caret symbol) correspond to components in the original program. Each component is labeled with its type (application, module, function, block or instruction), a unique ID, and other information like address and disassembly. In this example, the instructions are marked with an "s" for single precision or "d" for double precision.

Configuration files are generated by a utility included with the system, and can be hand-edited if desired. We also designed a simple Java-based GUI to increase the ease of viewing and modifying configuration files. Figures 4.8 and 4.9 show two different views provided by this interface.

The first view (Figure 4.8) shows the program component tree parsed from the target binary, as discussed in section 4.2. Each node in the tree represents

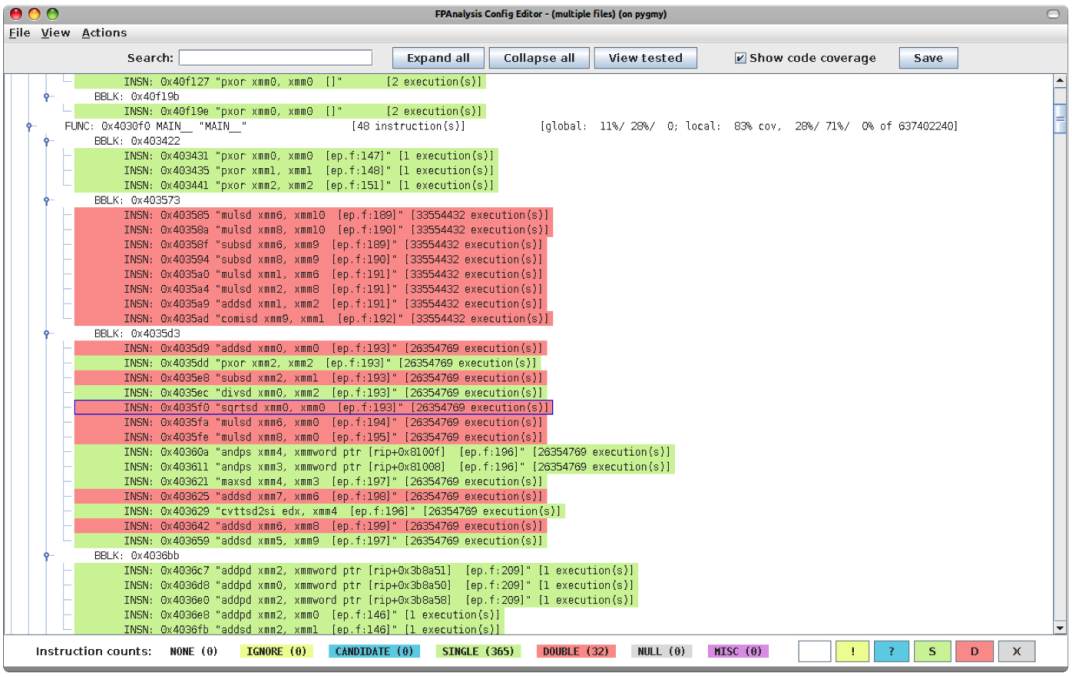


Figure 4.8: Configuration editor (instruction view)

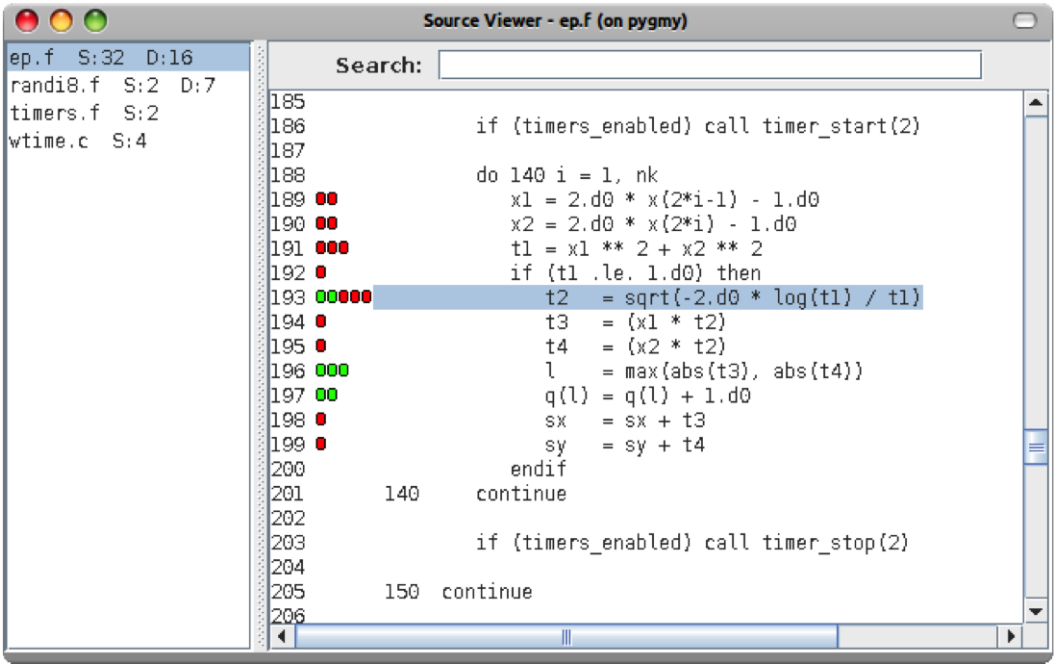


Figure 4.9: Configuration editor (source code view)

a component of the hierarchy and reflects a single line in the configuration file. The tree can be folded for easier viewing, and the interface provides the capability to search for a particular component. The view also provides several options for filtering the display and modifying the configuration. If present, the tree view can also display profiling information in the form of instruction counts as well as local (per function) or global execution percentages. The nodes are colored based on the type of analysis or replacement that the configuration specifies for that program component.

The second view (Figure 4.9) shows the program's original source code. This view collapses the instruction information from the first view and aggregates it by source line. The view is only available if the target binary was compiled with debugging information and if the original source files are available. The primary text view provides the source code in a scrollable window, with indicators along the left side that change color depending on the values from the current configuration file. The left pane provides a list of all source files used to build the target binary and some summary information about each of them.

Taken together, these interfaces provide a useful method of visualizing a program's control structure alongside configuration information. These interfaces are crucial during both the initial setup and final visualization phases in several of the analyses described in later chapters.

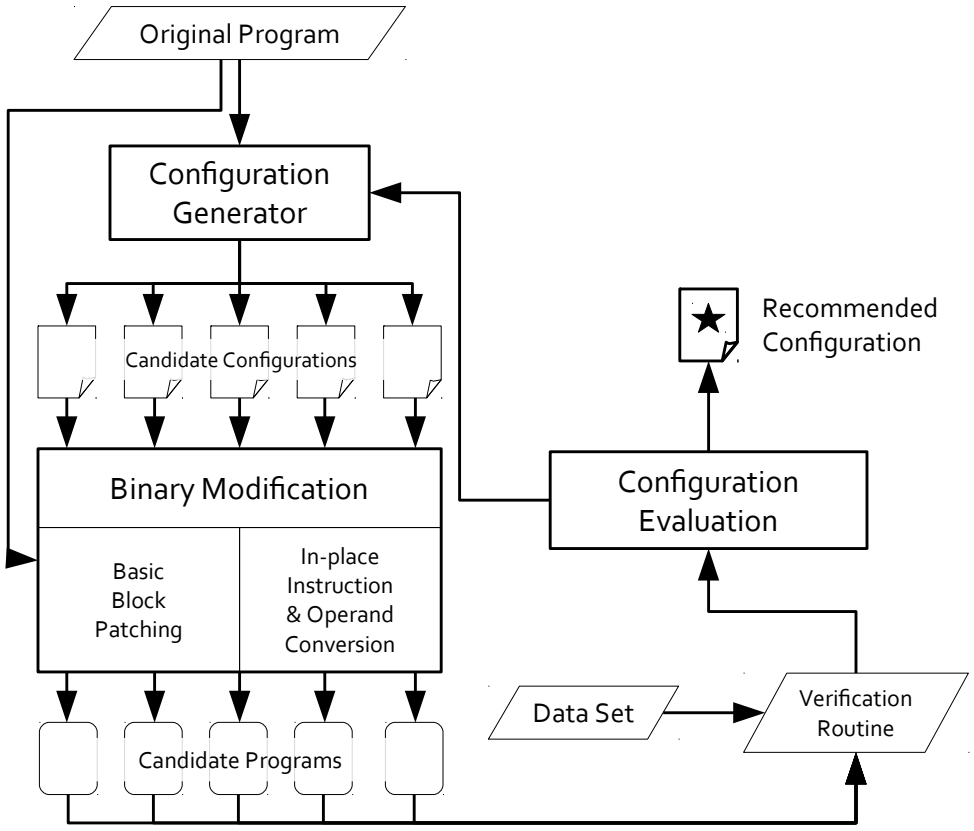


Figure 4.10: Overview of autotuning search process

4.6 Automatic search

While some of our analyses are intended to be run a single time on each target binary, others have configurations or parameters that can be changed. These parameters usually have some effect on a particular performance or evaluation metric, and thus our system provides the ability to tune these parameters using an evaluation loop. This loop executes many tests on variants of the original program, guided

by an optimization metric. This process is often called "empirical autotuning." Figure 4.10 shows the basic workflow for the process.

The basic search process is directed by a manager script, which executes initialization and profiling analyses before seeding a work queue with experimental program configurations. The manager process then spawns worker processes, which use the configurations from the work queue to build rewritten variants of the original target binary. The worker processes then run the variants using a user-provided evaluation script, recording the results. An evaluation routine examines these results and determines whether to generate more configurations for the work queue. When the work queue is exhausted, the worker processes exit and the manager process builds a final configuration based on the results of the search loop. The results are then presented to the user in a configuration file that represents the "best" combined configuration found during the search.

This process is optimized in several generic ways:

- If a single variant is split into many new variants after testing, these variants are combined into two aggregate variants. This aggregation represents a binary search approach, and usually it reduces the total number of variants tested. This behavior can be disabled if desired.

- In certain cases, the results of a test variant can be statically determined without executing the variant. This situation can happen when the profiling run reveals that the portion of the program in question is never executed,

or when the variant has been executed previously. The latter is especially useful for performing incremental searches; i.e., using the cached results from a shorter, shallow search to jump-start a longer, deeper search.

- The queue is usually sorted in descending order of execution percentage. This sorting enforces that the variants that test more frequently-executed portions of a program are given priority, because presumably these portions are of higher interest to the developer. As a result, the search converges faster on program execution time coverage.

4.7 Demonstration

In this section, we discuss several simple analyses created using the framework described earlier in this chapter. Although they are not novel analyses, they show the flexibility of the tool architecture presented in this dissertation to implement various analyses.

4.7.1 Instruction count analysis

In this section, we describe a simple instruction count analysis for floating-point instructions. This analysis resembles capabilities provided by most profiling tools, but filters out all instructions except for floating-point arithmetic or data movement. This provides a simple example of how to write an analysis using CRAFT, and provides a useful piece of analysis that we later use in conjunction with other more

Benchmark	Original time (s)	Overhead (X)
bt.A	60.1	51.8
cg.A	2.9	8.4
ep.A	9.1	13.3
ft.A	5.2	28.0
lu.A	48.3	31.1
mg.A	2.3	36.0
sp.A	42.9	28.3
ua.A	28.1	25.8

Figure 4.11: NAS benchmark overhead for instruction count analysis

complex analyses.

The analysis allocates a counter variable in the rewritten binary for each floating-point instruction. It then inserts pre-instruction instrumentation using a Dyninst snippet similar to the one shown in Figure 4.4. The analysis also contains output routines for reporting the results at the end of a run. The implementation for this analysis (`FPAnalysisCInst`) requires fewer than 200 lines of C++ code. Figure 4.11 shows example performance overhead results for this analysis. The second column shows the original wall time for the benchmark, and the third column shows the slowdown incurred by instrumentation as a multiple of the original wall time. For these single-core trials, the benchmarks were compiled using the Intel compiler with -O3 optimization and the results were averaged over five runs each. The overheads are higher than one might expect for such a simple analysis, but the ease of implementation shows that our framework provides expressiveness. Other analyses in this dissertation include instruction counting components that are implemented using binary snippets and have lower overhead.

4.7.2 NaN detection analysis

In this section, we describe how CRAFT can be used to detect Not-a-Number (NaN) values while a program is running. A NaN value is a specially-tagged floating-point value that generally indicates the occurrence of some kind of error during floating-point computation, such that the result cannot be represented in the current format. Sometimes these values arise as part of intended computation and are handled by the program itself; at other times they are considered anomalies and require debugging. In either case, developers benefit from an analysis that can detect NaN numbers when they occur, logging them without stopping execution. We implemented such an analysis using CRAFT by inserting a call to a shared library after every floating-point operation. The shared library examines the output of the operation and creates a log entry if a NaN value is detected.

Figure 4.12 shows example performance overhead results for this analysis. For these single-core trials, the benchmarks were compiled using the Intel compiler with -O3 optimization and the results were averaged over five runs each. The overhead for this analysis is generally much larger than the overhead for the analysis discussed in Section 4.7.1 because this analysis makes a call to an external analysis library. Some overhead is also due to event logging, because the overhead is related to the number of NaN values detected.

Benchmark	Original time (s)	Overhead (X)	NaNs detected
bt.A	61.5	805.0	18,021,075
cg.A	2.7	183.3	311,044
ep.A	9.3	143.8	0
ft.A	5.1	346.7	2,978,199
lu.A	48.9	524.0	26,131,409
mg.A	2.5	439.2	1,770,274
sp.A	49.0	363.8	21,914,446

Figure 4.12: NAS benchmark overhead for NaN detection analysis

4.7.3 Range tracking analysis

In this section, we describe how CRAFT can be used to implement the range-tracking analysis described by Brown et. al. [21] For this analysis, we insert instrumentation after each operation. The instrumentation examines the result of the operation, comparing it against the minimum and maximum values already seen for that instruction and replacing those values if it is the new minimum or maximum. The output of the analysis is a list of the minimum and maximum values for each instruction. These results inform developers of the dynamic range of each instruction. A low dynamic range indicates that the instruction may be a good candidate for lower precision or fixed-precision arithmetic.

Initially, we implemented this analysis using the same technique as the NaN-detection analysis, making a call to a shared library after each floating-point operation. The shared library contained a routine that compared the results of an operation with previous minimum and maximum values. However, this approach proved to have a high overhead, so we decided to use binary blob snippets as discussed in Section 4.3.2. This alternate approach reduced the overhead dramatically,

Benchmark	Original time (s)	Overhead (X)
bt.A	62.2	33.5
cg.A	2.5	6.5
ep.A	9.3	4.9
ft.A	5.3	9.1
lu.A	49.5	15.1
mg.A	2.4	12.3
sp.A	42.1	11.9
ua.A	33.1	12.6

Figure 4.13: NAS benchmark overhead for range tracking analysis

and is the approach used for the performance results in this section.

Figure 4.13 shows example performance overhead results for this analysis. For these single-core trials, the benchmarks were compiled using the Intel compiler with -O3 optimization and the results were averaged over five runs each. The overheads for this analysis are generally lower than the analyses from Sections 4.7.1 or 4.7.2 because this analysis is implemented using the binary blob snippet format. These overheads could potentially be reduced even further with better cache locality, because every operation in the original program is now accompanied by at least two extra memory accesses (for the min/max comparisons).

4.8 Conclusion

We have described CRAFT, a general framework for runtime binary floating-point program analysis. We have shown the value of this framework by presenting several useful demonstration analysis techniques. This framework serves as a base for all other analyses presented in this dissertation.

Chapter 5

Cancellation Detection

5.1 Overview

Cancellation was introduced in Section 2.5, and defined as the subtraction of two numbers of similar magnitude. Such subtraction causes a loss of significant digits that may negatively affect the accuracy of future computations or signal that an unacceptable loss of precision has already occurred. This chapter discusses our techniques for detecting cancellation as well as our implementation of these techniques in the CRAFT framework. We also present various results and benchmarks.

5.2 Techniques

Our approach involves examining the runtime values of operands to detect cancellation. We build an augmented version of the original program, inserting instrumentation before each addition and subtraction instruction. At runtime, this instrumentation examines the values of the instruction's operands and compares their relative magnitudes. If cancellation is detected, the system stores information regarding the cancellation in a log. This section describes our techniques for inserting the instrumentation and performing the runtime analysis. We also describe a GUI viewer for exploring and analyzing this log after the program is finished running.

5.2.1 Binary instrumentation

Our analysis detects and reports cancellation events, which are defined as follows. Assume that an addition or subtraction operation involves two values, stored in floating-point representation as $v_1 = (sig_1 \cdot 2^{exp_1})$ and $v_2 = (sig_2 \cdot 2^{exp_2})$. The result of the operation is stored as $v_r = v_1 + v_2 = (sig_r \cdot 2^{exp_r})$. If the operation was subtraction and the operands have the same sign, or if the operation was addition and the operands have opposite signs, then cancellation is possible. Our technique 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., $exp_r < \max(exp_1, exp_2)$), a cancellation event has occurred. This conditional test works regardless of the precision level of the individual instruction (i.e., single vs. double precision).

Further, we define the *priority* as $\max(exp_1, exp_2) - exp_r$, a measure of the severity of a cancellation. The analysis ignores any cancellations under a given minimum threshold. Unless otherwise noted, we use a threshold of ten bits (approximately three decimal digits) for the results in this dissertation.

To implement this analysis, we instrument every floating-point addition and subtraction operation in a target program, augmenting it with code that retrieves the operand and result values at runtime. After each operation, the system checks for the cancellation criteria described above. If the analysis determines that cancellation has occurred and that the priority is above the reporting threshold, it saves an entry

to a log file. This entry contains information about the instruction, the operands, and the current execution stack. The stack trace results are more informative if the original executable was compiled with debug information. The analysis also maintains basic instruction execution counters for the instrumented instructions.

Because many programs produce thousands or millions of cancellations, reporting the details of every single one is impractical and unhelpful. Instead, we use a sample-based approach. Unfortunately, the number of cancellations usually differs considerably between various instructions. Some instructions may produce fewer than ten cancellations during a single run while others produce millions. Thus, a uniform sampling strategy does not work, so we use a logarithmic sampling strategy. This strategy reports the first ten cancellations for each instruction, then every tenth cancellation of the next thousand, then every hundred thousandth cancellation thereafter. We found that this strategy produces an amount of output that is both useful and manageable. We emphasize that all cancellations are counted and that the sampling applies only to the logging of detailed information such as operand values and stack traces.

5.2.2 Results viewer

We have also created a log viewer 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 operands and stack traces. It also aggregates count

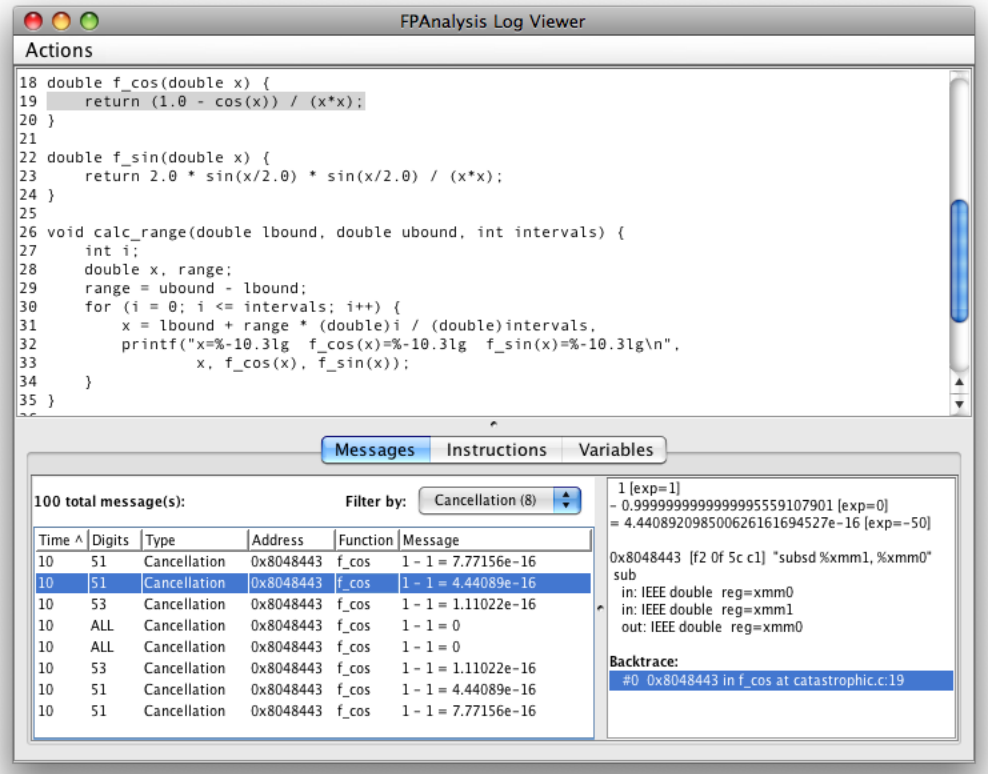


Figure 5.1: Sample log viewer results

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 we define 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 canceled bits) for all cancellations at each instruction. This average gives an indication of how severe the cancellations induced by that instruction were.

Figure 5.1 shows a representative screenshot of the log viewer interface. The

Benchmark	Original time (s)	Overhead (X)
cg.A	2.5	114.0
ep.A	9.1	69.9
ft.A	5.1	208.5
lu.A	48.9	243.1
mg.A	2.4	428.4

Figure 5.2: NAS benchmark overhead for cancellation detection analysis

Name	Original	Overhead (X)
soplex	1s	10
povray	2s	85
lbm	20s	70
milc	44s	75
namd	95s	160

Figure 5.3: SPEC benchmark overhead for cancellation detection analysis

lower portion displays all events logged during execution. Each event is displayed in the list in the lower-left corner, along with summary information about the event. Clicking on an individual event reveals more information in the lower-right corner and also loads the source code in the top window if the debug information and the source files are available. If possible, the interface highlights the source line containing the selected instruction. The tab selector in the middle allows access to other information, such as a view of cancellations aggregated by instruction.

5.3 Benchmarking

Figures 5.2 and 5.3 show performance overhead results for this analysis. For the NAS single-core trials, the benchmarks were compiled using the Intel compiler with -O3 optimization and the results were averaged over five runs each. We used the

"A"-sized problems for the NAS benchmarks. For the SPEC CPU 2006 trials, the benchmarks were compiled using GCC and the default compilation options, and tested using the provided "test" data sets. We used these smaller sets so that we could complete the analyses in a reasonable time. The overheads vary depending on the amount of addition and subtraction operations in the benchmark, but are generally between 10–250X. This overhead is significant, but it is not impractical for occasional analysis.

5.4 Results

In this section, we discuss some of the results obtained by applying cancellation detection analysis to various applications and problems.

5.4.1 Simple cancellation

Our first test case is a simple example of cancellation. This sort of example is well-known to numerical analysts, with many known workarounds. Here it serves as an introductory demonstration of our techniques.

$$y = \frac{1 - \cos\ x}{x^2} \tag{5.1}$$

Figure 5.4 (left side) shows the graphical representation of the function given in Equation 5.1. This function is undefined at $x = 0$ because this triggers a division by zero, but as it approaches that point the function value becomes infinitely close

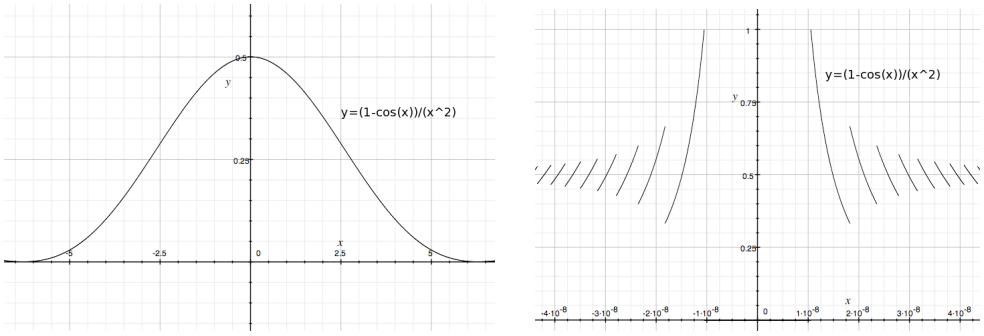


Figure 5.4: Graphs of Equation 5.1: at normal zoom (left) and zoomed to the area of interest (right).

to 1/2. In floating point, the subtraction operation in the numerator results in cancellation around $x = 0$ because $cos\ 0 = 1$. This cancellation causes the divergent behavior shown in Figure 5.4 (right side). The jagged appearance of the divergence is a result of the discretization of the cosine function near machine epsilon. The preferred way to avoid this behavior is to rewrite the function to avoid the cancellation. In this case, trigonometric identities allow it to be written to use the sine function, which does not suffer from the same cancellation issues at $x = 0$.

We wrote a simple program that evaluates this function at several points approaching $x = 0$ from both sides, and ran our cancellation detector on it. The analysis reported all cancellation events that we expected. The output log included details about the instruction, the operands, and the number of binary digits canceled. Figure 5.1 shows a screenshot of the log viewer interface after analysis.

This simple example confirmed our expectations and demonstrates how our analysis works. The highlighted message reveals a 51-bit cancellation in the subtraction operation on line 19 of `catastrophic.c`. The two operands involved were

two XMM registers with values that were both close to 1.0 (the first was exact and the second diverged around the sixteenth decimal digit). Selecting the other events reveals similar details for those cancellations. Being able to examine cancellation at this level of detail is valuable in analyzing the numerical stability of a floating-point program. In this case, the results alert us that that the results of the subtraction operation on line 19 may cause a cancellation of many digits. Because the resulting value is later used on the same line to scale another value, we may deduce that this code needs to be rewritten to avoid the loss of significant digits.

5.4.2 Approximate nearest neighbor

To investigate the ability of our analysis to detect change in the cancellation behavior of a program based on input data, we examined an approximate nearest-neighbor software library called ANN [12]. This computational geometry library takes as inputs 1) a series of data points and 2) a series of query points. The software then finds the nearest data point neighbor (by Euclidean distance) to each query point using an approximate algorithm. This program is of interest to researchers in high-performance computing (HPC) as well as computational geometry. Algorithms like ANN are often used in HPC for autotuning, image processing (classification and pattern recognition), and DNA sequencing.

We ran this program instrumented with our cancellation analysis twice with different sets of points. Each set included 500,000 data points and 5,000 query

points. The first data set was composed of points randomly generated uniformly throughout the square defined by x- and y-coordinate ranges of $[-1, 1]$. The second data set was composed of points randomly generated close to the same square (i.e., most x- and y-coordinates were nearly identical, and close to either -1 or 1). The expectation was that the second input would lead to many more cancellations for certain instructions in the distance calculation, because the coordinates are much closer.

This expectation was confirmed. The first data set caused cancellation in less than 1% of the executions of the instructions of interest, and the average number of canceled bits was less than 15. The second data set caused cancellations in 100% of the executions for the same instructions, and the average number of canceled bits was 46. These differing results show that the analysis can expose differences in floating-point error on the same code resulting from varying data sets, which static analysis techniques cannot do.

5.4.3 Gaussian elimination

The ability of cancellation detection to shed light on a particular algorithm has limitations, for two principal reasons. 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. In this section, we examine these limits by looking at two

```
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 5.5: Classical Gaussian elimination with partial pivoting

```
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 5.6: Bordered algorithm for Gaussian elimination

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

(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 5.7: Example of cancellation in Gaussian elimination

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.

Matlab code for Gaussian elimination with partial pivoting is given in Figure 5.5. Because Matlab is an interpreted language, we also wrote a C version of this code so that we could apply our binary analysis. The result of this code is a unit lower triangular matrix

$$L = tril(A, -1) + diag(ones(1, n))$$

and an upper triangular matrix `U = triu(A)` such that

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

The purpose of the partial pivoting in lines 3–5 of Figure 5.5 is nominally to avoid division by zero in line 6. However, if `A(k,k)` is small, the algorithm will produce inaccurate results, which cancellation will signal. Consider what happens when we omit lines 3–5 in Figure 5.5 and apply it to the matrix shown in Figure 5.7(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 5.7(b). Because 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 5.7(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 computations for this example were done in six-digit decimal floating-point arithmetic using the Matlab package Flap [3].

The cancellation itself introduces no significant errors. Rather, the loss of precision occurred in passing from the data shown in Figure 5.7(a) to that of Figure 5.7(b). 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. In this way, cancellation is a lot like a null pointer dereference, where the null pointer exception is not 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 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. Thus, 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 that we expect due to the bad pivot by determining the dimensions of the array in which the cancellation will occur. Our array is of order $n - p - 2$, and so 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 5.8. The rows labeled "Count" give the cancellation counts for the entire elimination while the rows labeled "Trunc" give the count for the truncated elimination. The rows labeled "Est" contain the cancellation count 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. This overestimation is 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 is diminished; cancellation can inform the choice of where and when to pivot.

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, for which pivoting is not required to guarantee stability.

Let us suppose that we have a positive definite matrix A whose eigenvalues descend in geometric progression from one to $10^{-\text{logkap}}$, where logkap is a constant

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 5.8: Cancellation for unpivoted Gaussian elimination

that we use to manipulate the conditioning of the matrix. In this context, the matrix A has the condition number $\kappa = \|A\|\|A^{-1}\| = 10^{\text{logkap}}$. When Gaussian elimination computes the LU-factorization of A, the diagonals of U generally track the eigenvalues of A. Because the elements of A are of order one, the diagonals of U (which become progressively smaller) are calculated with cancellation.

One opportunity for analysis here is that Gaussian elimination has many variants. Consider, for example, the code in Figure 5.6 that does 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 cancellation in different ways.

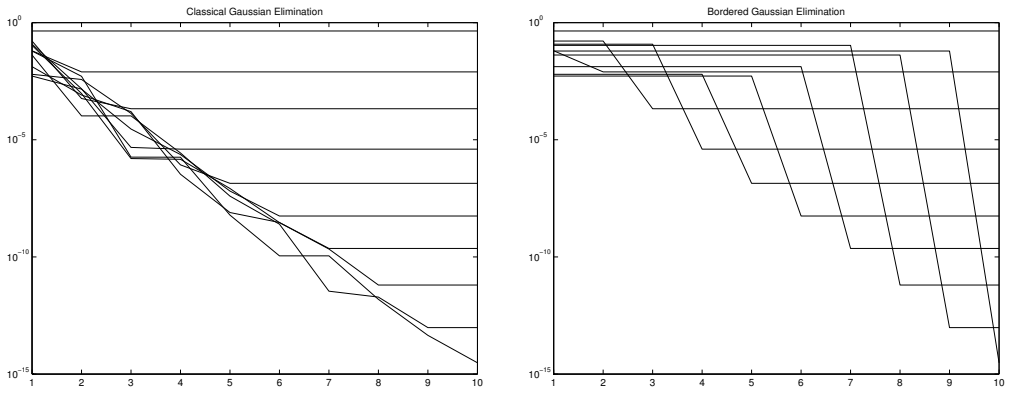


Figure 5.9: Diagonal elements for classical (left) and bordered (right) Gaussian elimination

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

The plots in Figure 5.9 contain histories of the diagonal elements of the reduction of the matrix A described above with $n = 10$ and logkap $= 15$. 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, the first diagonal remains constant during the first iteration while the others decrease by roughly the same amount. In the second iteration, the second diagonal peels off and remains constant, while the others decrease. Thus, in the ith iteration, the ith diagonal becomes constant while all lower diagonals continue to decrease. In the end, each diagonal contains a rough approximation to

its corresponding eigenvalue. In the border variant, on the other hand, all diagonals remain constant during the ith iteration except the ith value, which drops to its final value and remains constant thereafter. Thus each diagonal makes only one transition (from its initial value to its final value). The initial and final values for both methods are identical. 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.

These results suggest that cancellation detection works better for the bordered variant. Figure 5.10 contains counts for both variants of the cancellation detection threshold and logkap. Counting the drops in the graph for the border method, we see that our detection should register nine cancellations, which it does 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, and small ones may fall by the wayside due to being under our priority threshold. Only with logkap equal to 15 and a threshold of one bit, does it come near 45.

From these experiments, we learn several things. First, varying the threshold is important. Most computations have a background of small cancellations, which overwhelms more important cancellations if the threshold is set too low. Trying different thresholds may give a better view of what is happening. Second (and 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. Third, 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.

Finally, we acknowledge that not all cancellations are bad. A good example is the computation of a residual to determine the convergence of an iterative method. Because a small residual means convergence, a large cancellation encountered while computing it indicates that the algorithm has computed an accurate answer.

5.4.4 NAS and SPEC benchmarks

To show our framework's ability to analyze larger programs, we also ran it on the SPEC CPU2006 benchmark suite [6] and the NAS Parallel Benchmarks [14]. Section 5.3 shows the overhead results for these experiments.

One interesting discovery was a section in the "povray" (ray-tracer) SPEC benchmark where a color calculation showed cancellation. In this routine, given values were subtracted from 1.0 to give percentage components in red, green, and blue. Thus, complete cancellation in all three variables indicates the color black, and had nothing to do with numerical accuracy.

The most common result was that most cancellations occurred in a few of the floating-point instructions: usually fewer than twenty instructions. Often, there were several instructions that caused cancellations 100% of the time. Without domain-specific knowledge, we do not know whether these cancellations indicate a larger problem in the code.

5.5 Conclusion

We have described a technique for detecting cancellation events in floating-point programs. We have also presented an implementation of this technique using the CRAFT framework. We have demonstrated the usefulness and overhead of the analysis on several applications and benchmarks. This work provides runtime analysis capability and insights regarding cancellation that were not previously available. Later work by others [19] expands on cancellation detection, incorporating more information to make a better decision about which instructions are malignant and which are benign.

Chapter 6

Mixed-precision Replacement

6.1 Overview

As described in Section 3.5, mixed-precision configurations hold much promise for improving performance while maintaining acceptable levels of accuracy. Unfortunately, building prototype mixed-precision configurations can be time-consuming for developers. Additionally, it is not always obvious which parts of a program must be run in double-precision and which parts can be run in single-precision. This chapter describes our techniques that build mixed-precision configurations of a program automatically, as well as our implementation of these techniques using the CRAFT framework. The goal is to provide insight regarding which parts of a program can be replaced with single-precision arithmetic. We do not intend to achieve a performance gain while running the analysis, but rather to provide a rapid prototyping environment as well as a system for automated recommendations for mixed-precision adaptation.

Our approach starts with a target application written in double-precision arithmetic as well as a user-provided script that provides a verification routine for automated testing. The verification routine should exercise the program using representative data sets or inputs, and provide some formatted output indicating whether

the results are correct. Our system automatically generates various precision-level configurations and uses the verification routine to determine which configurations are acceptable.

Our mixed-precision configuration implementation is built on the CRAFT framework, and Figure 6.1 shows an overview of the binary editing process. We provide a Dyninst-based mutator that accepts a target double-precision program and a mixed-precision configuration. The configuration contains a mapping between the instructions in a program and the desired precision level for that instruction. The output of the CRAFT mutator is a rewritten version of the original binary where the desired portions of the program have been replaced by single-precision arithmetic. This mixed-precision version can be executed the same way as the original.

We also provide a search routine that determines how much of the original program can be run in single precision while still passing a user-provided verification test. The search uses an empirical autotuning loop, testing many different configurations and observing the results. The final output is a report that details which parts of the program can be individually replaced with single-precision arithmetic while still passing verification.

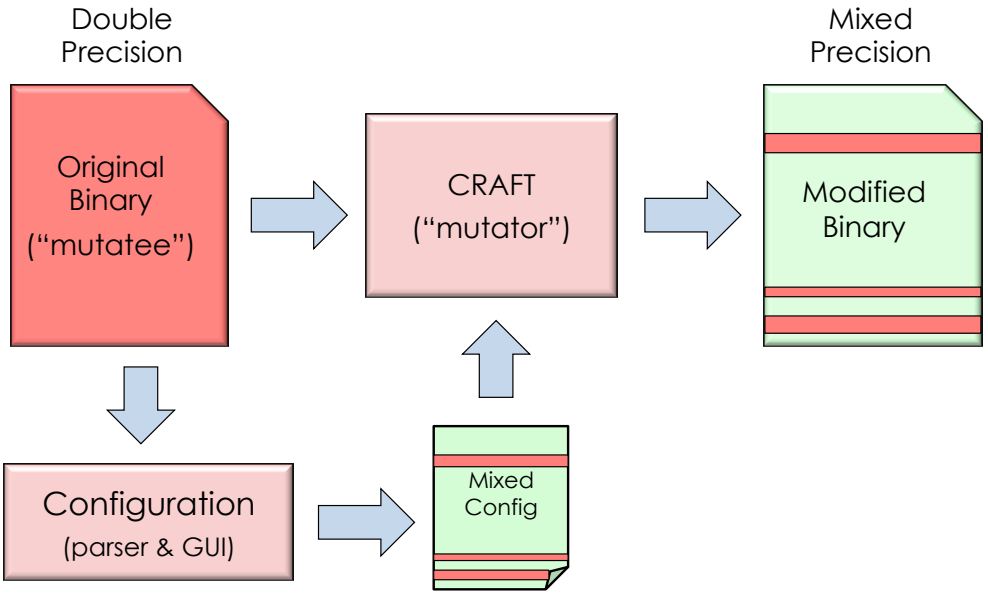


Figure 6.1: Overview of binary editing process

6.2 Techniques

6.2.1 Mixed-precision configurations

Our "precision configurations" provide a method of communicating which parts of a program should be executed in single precision and which parts should be executed in double precision. A configuration is a series of mappings:

$$p \rightarrow \{single, double, ignore\}$$

The mappings involve all points $p \in P_d$, where P_d is the set of all double-precision instructions in program P. Because the program structure contains natural parent-child containment aggregations (i.e., instructions are contained in basic blocks, which are contained in functions), the configuration allows decisions to be made about these aggregate structures, overriding any decisions about child members. The configuration controls 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 the 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 option is useful for flagging unusual constructs that manipulate floating-point numbers with bitwise operations, such as random number generation routines.

We use a simple, human-readable exchange file format to store these configurations (Figure 6.2 shows an example). The file is plain text and lists the program's functions, basic blocks, and instructions, using indentation to improve readability. The list of these structures is generated using a simple static analysis that traverses the program's control flow graph. The first column contains flags such as "d" (double precision), "s" (single precision), or "i" (ignore) that control the precision of the code structures during instrumentation. The format supports simple toggling of larger aggregate structures like functions. If an aggregate entry has a flag in the first column, it overrides any flags specified for its children; if the aggregate entry has no flag, each child's flag applies individually.

In the example configuration shown in Figure 6.2, 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.

We also built a GUI (shown in Figure 6.3) that displays a tree corresponding to the program structure, allowing a developer to adjust a configuration quickly without having to edit a lengthy text file. The GUI also allows the developer to visualize the results of our automatic search to understand what parts of the code can be changed

```
    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 6.2: Example replacement analysis configuration file

to single precision. The color green indicates a single-precision instruction and the color red indicates a double-precision instruction. If debug information is available, the GUI can also present a view that shows the corresponding source code location for a particular instruction. This capability aids in the conversion of particular code regions to single precision.

6.2.2 Binary modification

Our general strategy for implementing mixed-precision configurations in existing binaries is to replace some double-precision instructions selectively with their single-precision equivalents, and to replace some double-precision operands with their single-precision equivalents in memory. These narrowing conversions (double precision to single precision) allow our analysis to store the new (lower-precision) value

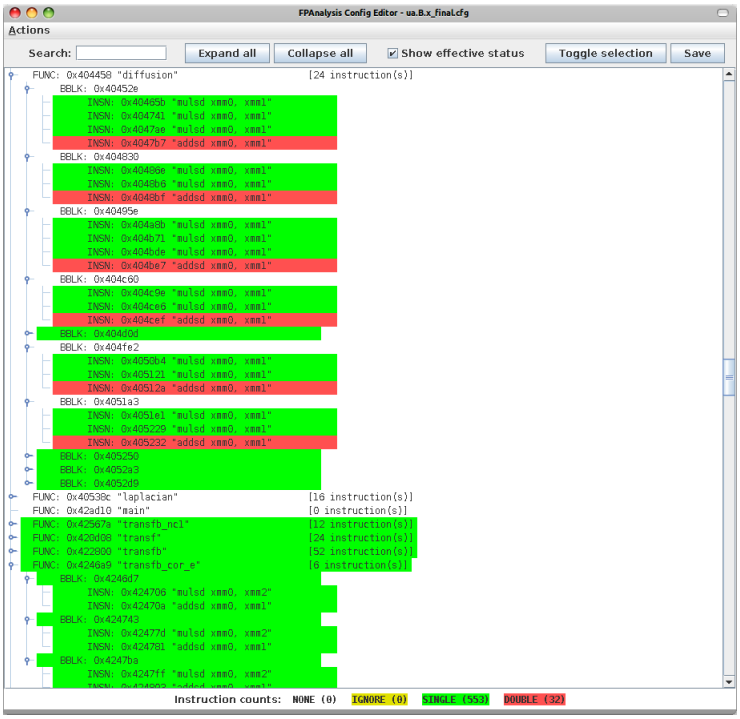


Figure 6.3: Graphical configuration editor, viewing a configuration for one of the NAS benchmarks

in the same location as the old value. Thus, the 32 bits of the new single-precision value are stored in the lower 32 bits of the original 64-bit double-precision register or memory location. The remaining high 32 bits are set to a specific bit pattern (`0x7FF4DEAD`) to indicate to future instructions that this value has been replaced. The first four hex digits (`0x7FF4`) encode a NaN, ensuring that the program never silently propagates incorrect values. The second four hex digits (`0xDEAD`) form a common human-readable value that is easy to spot in a hex dump. Figure 6.4 illustrates this process. This technique works for single values as well as "packed" floating-point values in 128-bit XMM registers.

To implement a replaced instruction, our framework inserts a streamlined "bi-

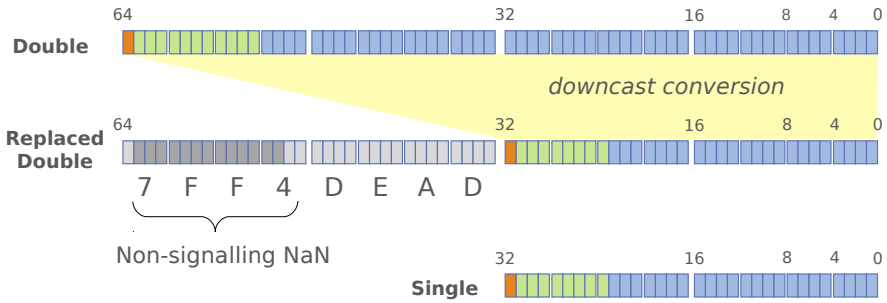


Figure 6.4: In-place downcast conversion and replacement

nary blob" snippet of machine code instructions. This snippet checks the operands, replacing them if necessary, and runs the original instruction in the desired precision. The desired precision for each floating-point instruction can be specified by a configuration file, as described in Section 6.2.1. These new machine code instructions are generated by a simple snippet compiler, which implements routines for building flag checks and for re-writing instruction opcodes in lower precisions. Figure 6.5 shows the template for these snippets in the case where we emulate the instruction in single precision.

Because most of the snippet operations are integer instructions, the snippets impose a minimal overhead, and the downcast operation is performed only when the input has not already been replaced. To avoid hard-to-find synchronization bugs or attempting to write to unwritable memory, the analysis copies any memory operands into a temporary register, and modifies the replaced instruction to use only register operands. Once we replace any instruction with its single-precision equivalent, we must usually replace all floating-point instructions with our snippets, even the ones

```
  push %rax
  push %rbx

  <for each input operand>
    <copy input into %rax>
    mov %rbx, 0xffffffff00000000
    and %rax, %rbx               # extract high word
    mov %rbx, 0x7ff4dead00000000
    test %rax, %rbx              # check for flag
    je next                      # skip if 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>      # e.g. addsd => addss

  <fix flags in any packed outputs>
```

Figure 6.5: Single-precision replacement template

that are to be performed in double-precision. This change is necessary even if we do not replace a particular instruction with its single-precision equivalent, because we must add a check and possible upcast if any of the incoming operands were replaced with single precision by an earlier operation.

To modify the binary and to insert our code snippets, we use Dyninst's CFG-patching API as described in Section 4.3.3.

6.2.3 Automatic search

We developed an automatic search technique that attempts to replace as much of the program as possible using a breadth-first search through the entire program's configuration space. The basic search algorithm is described in Section 4.6. Figure 6.6 shows the specific autotuning process for mixed-precision analysis. For this analysis, the tuning parameters are the precision levels of all floating-point operations in the

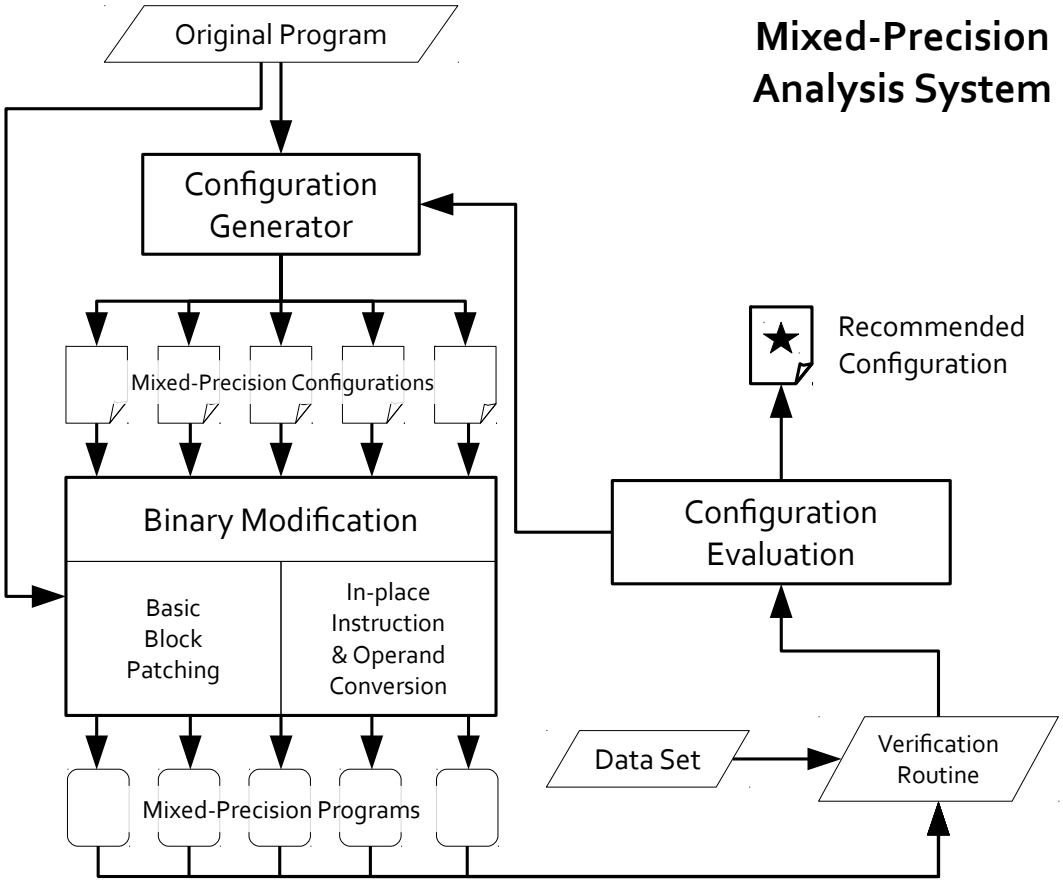


Figure 6.6: Overview of mixed-precision search process

target binary, and the optimization metric to be maximized is the percentage of the program that can be replaced by single-precision arithmetic.

Because every instruction could be executed in either single- or double-precision arithmetic, the search routine can build 2^n total unique configurations, where n is the number of floating-point instructions in the program. Each instruction that could be replaced is called a "candidate" for replacement search. Usually, every floating-point computation instruction is a candidate. Pure data movement instructions (without size conversions) are usually ignored because they do not effect the precision of the numbers. Because evaluating each test configuration requires a full program run, exhaustively testing every configuration is not feasible. Our breadth-first search strategy exploits control structure in programs for a faster search.

Our search maintains a work queue of possible configurations, testing them one by one and adding to the final configuration any individual configurations that pass the application-defined verification process. This process is highly parallelizable, and the system can launch many independent tests if cores are available. The search first generates configurations for each module in the program. Each configuration replaces the entire corresponding module with single precision, leaving the rest of the program in double precision. If any of these module-level configurations fail to pass verification, the routine begins to descend recursively through the program structure, testing function-level configurations before continuing to basic blocks and finally individual instructions as necessary. The recursion terminates when any

structure (module, function, or block) is replaced and passes verification, or when the search tests an individual instruction (which cannot be further subdivided). The search can also be configured to stop at basic blocks or functions, providing faster convergence with coarser results.

Performing a brute-force breadth-first search through the program's structure, our system finds the coarsest granularity at which each part of the program can successfully be replaced by single precision. The routine then assembles a "final" configuration by taking the union of all previously-found successful individual configurations. This configuration is also tested automatically, although it may not pass verification as-is because the precision levels of various instructions are not independent. In other words, decreasing precision in one part of a program may impact the sensitivity of other portions. However, the final configuration serves as a starting point for the developer to investigate because it represents an indicator regarding which parts of the original program can be individually replaced by single precision while maintaining the original desired level of accuracy.

6.2.4 Memory-based analysis

In addition to the core computation-based analysis, we also built a memory-centric analysis. This mode focuses on identifying specific memory locations and structures that could be stored in single precision rather than double precision, resulting in a space savings. In this mode, all computation is still done in double-precision; only

the memory writes are replaced. The motivation for this analysis lies in the observation that memory bandwidth is quickly becoming a large issue in high-performance computation. In fact, industry experts expect that data movement and storage bottlenecks will outweigh computational bottlenecks in exascale computing [30]. Our analysis attempts to find the largest subsections of a program's memory that can be stored in single precision while still passing verification, assuming that the arithmetic is still performed in double precision. The current analysis examines individual instructions; the results can then be traced back to memory locations or source code variables using debug information.

Another motivation is that insights regarding memory locations will translate more easily to code transformations than will insights regarding specific computation operations. Because code transformations usually involve the modification of variable or data structure types, they are usually difficult to specify at the level of individual operations. Recommendations regarding memory locations are therefore more actionable than recommendations regarding instructions.

The implementation of memory-based analysis resembles the normal mixed-precision analysis described in previous sections. One key difference is that data movement instructions are included in the analysis. These instructions can be ignored in a computation-centric analysis because movement instructions are agnostic to whether the double-precision floating-point number being moved has been replaced with our special single-precision encoding. Thus, the number of instructions

that must be replaced increases for memory-centric analysis, but the number of candidate instructions is reduced because only those with memory writes are considered for replacement.

To implement this analysis, CRAFT replaces all floating-point instructions that read or write memory operands. The write instructions are replaced with code that does the original operation, optionally saving the result using the replacement scheme described in Section 6.2.2. The read instructions are augmented with code that checks for replaced (truncated) values, upcasting them to double precision before saving them in a register or using them in an operation. Few instructions in the SSE instruction set both read and write memory operands, and those instructions are handled on a case-by-case basis. Values are always cast to full double precision when moved from memory into an XMM register or used directly from memory in a calculation. These modifications ensure that all computation is performed in double precision. Floating-point values are then optionally stored in single precision when moved from a register to memory.

6.3 Benchmarking

To examine the overhead of our techniques, we looked at scaling by replacing all instructions with double-precision snippets. This transformation does not affect the semantics or results of the program, but shows how much overhead our inserted code causes in the base case in which it makes no conversions. Figure 6.7 shows

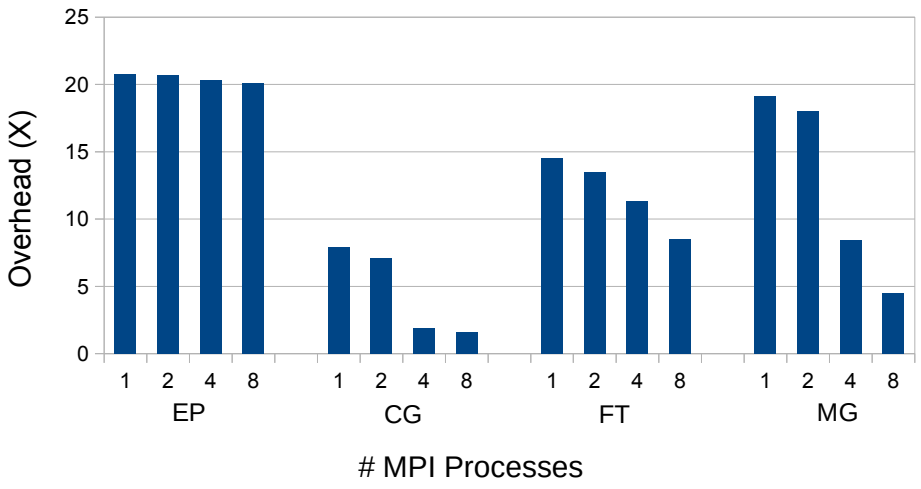


Figure 6.7: NAS MPI scaling results

the runtime overhead results from these experiments with Class A versions of the NAS benchmarks. The overall overhead decreases as the number of MPI processes increases. Figure 6.8 shows specific runtime overheads from individual configurations for Class A and C inputs.

All benchmarks in Figure 6.8 were MPI versions compiled by the Intel Fortran compiler with -O2 optimization enabled. Tests were performed on a distributed

Benchmark	Overhead (X)
ep.A	3.4X
ep.C	5.5X
cg.A	3.4X
cg.C	4.5X
ft.A	4.2X
ft.C	7.0X
mg.A	5.8X
mg.C	14.7X

Figure 6.8: NAS benchmark overhead results

cluster, each node with twelve Intel Xeon 2.8GHz cores and 24 GB memory running 64-bit Linux. Each run used eight cores and one MPI process per node. The overhead was calculated as the ratio between the instrumented and original execution user CPU times as reported by the "time" command. Usually, these overheads are under 20X, making this technique viable for test and trial runs on real data.

6.4 Results

6.4.1 NAS benchmarks

We first verified the correctness of our replacement on several NAS benchmarks [14] by manually converting the codes to use single precision and comparing the outputs to that of the instrumented version. The final results were identical, bit-for-bit, indicating that the instrumented versions were performing the same operations as the manually-converted versions of the original programs.

As a part of this verification, we developed a small script that attempted an automatic translation of Fortran source code to use single precision instead of double precision. However, we still had to tweak some files by hand. Sometimes, while examining the results of the comparison we found errors in our manual conversion, which needed to be corrected before the results matched. This process demonstrates that even by itself, the whole-program replacement routine is valuable as an automation of an error-prone manual (or semi-automated) process.

We also ran our automatic mixed-configuration search on the NAS bench-

Benchmark	Candidates	Configurations Tested	Instructions Replaced Static	Instructions Replaced Dynamic	Final Verification
bt.W	6,647	3,854	76.2%	85.7%	fail
bt.A	6,682	3,832	75.9%	81.6%	pass
cg.W	940	270	93.7%	6.4%	pass
cg.A	934	229	94.7%	5.3%	pass
ep.W	397	112	93.7%	30.7%	pass
ep.A	397	113	93.1%	23.9%	pass
ft.W	422	72	84.4%	0.3%	pass
ft.A	422	73	93.6%	0.2%	pass
lu.W	5,957	3,769	73.7%	65.5%	fail
lu.A	5,929	2,814	80.4%	69.4%	pass
mg.W	1,351	458	84.4%	28.0%	pass
mg.A	1,351	456	84.1%	24.4%	pass
sp.W	4,772	5,729	36.9%	45.8%	fail
sp.A	4,821	5,044	51.9%	43.0%	fail

Figure 6.9: NAS benchmark results

marks. For simplicity and speed of execution, we used the single-threaded versions of the benchmarks in this experiment. For each benchmark, we tested two input set sizes: class W and class A. All benchmarks were compiled by the Intel Fortran compiler with optimization enabled, and the tests were performed on a shared-memory workstation with Intel Xeon processors and 48 GB of memory running 64-bit Linux.

The benchmarks provided a wide range of replacement results. The percentages in Figure 6.9 indicate how sensitive the benchmark is to the precision level used. The columns contain the number of instructions that were candidates for replacement, the total number of configurations tested, the percentage of instructions replaced with single precision (measured statically), the percentage of instruction executions replaced (measured dynamically at runtime), and the verification result

of the final composed configuration.

In nearly all cases, the system tested fewer configurations than replacement candidates, showing that our techniques for pruning the search space are effective. The one exception, SP, had many instruction sequences in the final configuration that alternated between single- and double-precision instructions. This alternation pattern caused our search engine to spend inordinate time searching individual instructions rather than aggregate structures, and thus the number of tested configurations was higher than the actual candidate count.

Some benchmarks (such as CG and FT) seem to be highly sensitive, because only a negligible percentage of instruction executions can be replaced. Others seem to hold more promise for building a mixed-precision version. The ones that fail the final verification illustrate that using the simple union of all individually passing instructions does not automatically guarantee a passing configuration. This observation suggests that a second search phase may be useful, to determine the largest subset of individually-passing instruction replacements that may be composed to create a passing final configuration.

Figure 6.10 shows the results of running the memory-based analysis described in Section 6.2.4 on the NAS benchmarks. The replacement percentages are higher than the corresponding results from the computation-based analysis. The higher percentages imply that there may be more opportunities for single-precision replacement for data storage than for computation.

Benchmark	Candidates	Configurations Tested	Instructions Replaced Static	Instructions Replaced Dynamic	Final Verification
bt.A	2,342	300	98.3%	97.0%	fail
cg.A	287	68	96.2%	71.3%	fail
ep.A	236	59	96.2%	37.9%	fail
ft.A	466	108	94.2%	46.2%	fail
lu.A	1,742	104	98.5%	99.9%	fail
mg.A	597	153	95.6%	83.4%	pass
sp.A	1,525	1,094	81.1%	75.1%	fail
ua.A	4,310	741	94.5%	88.9%	fail

Figure 6.10: NAS benchmark results for memory-based analysis (columns same as Figure 6.9)

We also attempted manual code conversion for several of these benchmarks. In two cases (CG and MG), the analysis found intermediate result data structures that could be stored in single precision with minimal negative effects on the final results. After changing the type definitions of these structures in the original source and re-compiling the benchmarks, the new versions passed the self-verification routines. The modified versions did not run significantly faster nor did they use a significantly smaller amount of memory, but this result is not surprising given the single-core nature of the benchmarks and our lack of expertise in the problem domains. Nevertheless, this shows that automated techniques can provide actionable recommendations, leading to mixed-precision versions of the original programs that pass verification. We expect that a programmer familiar with the problem domains would be able to achieve even better results based on the insights provided by our analysis.

6.4.2 AMG microkernel

To conduct an end-to-end test of our techniques with a subsequent code modification by a programmer, we looked for a program that could run entirely in single precision, to simplify the manual conversion process. It was also necessary that the program include a verification routine that could be analyzed by our analysis. The Algebraic MultiGrid microkernel [1], which implements the critical sections of a multigrid solver, met our needs. For our experiments, we used 5,000 iterations on eight cores.

As expected, our system verified that the kernel could be replaced with single precision. The overhead of our analysis was only 1.2X for this benchmark, with all instructions replaced by single precision. We verified the results by manually converting the entire program to single precision and re-compiling. This conversion includes changing the verification routine to single precision, but in some circumstances this change is acceptable. In this instance, the adaptive nature of the multigrid method corrects for numerical inaccuracy by iterating to increasingly accurate results. For the microkernel, we observed a user CPU time decrease from 175.48s for the double-precision version to 95.25s for the single-precision version, a nearly 2X speedup.

6.4.3 SuperLU

To evaluate our techniques further, we found a software library that is already implemented in both single- and double-precision versions: the SuperLU [29] general

	Instructions Replaced		
Threshold	Static	Dynamic	Final Error
1.0e-03	99.1%	99.9%	1.59e-04
1.0e-04	94.1%	87.3%	4.42e-05
7.5e-05	91.3%	52.5%	4.40e-05
5.0e-05	87.9%	45.2%	3.00e-05
2.5e-05	80.3%	26.6%	1.69e-05
1.0e-05	75.4%	1.6%	7.15e-07
1.0e-06	72.6%	1.6%	4.77e-07

Figure 6.11: SuperLU linear solver memplus results

purpose linear solver. LU decomposition and linear solving often comprise the most computationally expensive portions of larger scientific codes. The SuperLU library does such operations, and it includes a linear solver example program that can be compiled to use either single- or double-precision (but not mixed-precision). The program also reports an error metric that is useful in comparing the sensitivity of various mixed-precision configurations.

For these experiments we used the "memplus" memory circuit design data set from the Matrix Market [20], which contains nearly 18K rows. The linear solver for this data set runs for around three seconds on our machine, which allows us to test many configurations. The single-precision manually recompiled version achieves a 1.16X speedup over the double-precision version, which is equivalent to a 150 MFlops improvement. The reported error for the double-precision version of the solver is 2.16e-12, and the reported error for the single-precision version is 5.86e-04.

To run an automated search on the linear solver program, we wrote a driver script that ran the program and compared the reported error against a predefined

threshold error bound. Using an error bound just above the error returned by the single-precision version, our search found that 99.1% of the double-precision solver's floating-point instructions could be replaced by single-precision. This replacement included 99.9% of all runtime floating-point operations. That percentage matches the precision profile of the single-precision version of the program, and shows that our analysis can find all replacements inserted manually by an expert.

Figure 6.11 shows the results of running our automated mixed-precision search using various error thresholds. The general trend is that when the error threshold is stricter, the search finds fewer static or dynamic instructions that can be replaced. For this application, the error of the final run (using the union of all passing configurations) tends to be much lower than the threshold used during the search. This shows that our techniques can illuminate the tradeoff between single-precision replacement and computational error. In the future, precision levels could be treated as another variable "control knob" during program performance tuning.

6.5 Conclusion

We have described techniques for building mixed-precision configurations of existing double-precision binaries, as well as for searching an application automatically for portions that can be replaced with single-precision arithmetic. These techniques are applicable to both computation and data movement. We have also described our implementation of these techniques using the CRAFT framework. Benchmark

overhead results show our techniques' feasibility, and we have described an experiment that demonstrated a speedup gained by an analysis-guided conversion. This work provides insights that were not previously available, and improves the ability of floating-point application developers to make informed decisions regarding the behavior of their code and the necessary precision levels for various parts of their programs.

Chapter 7

Reduced-precision Replacement

7.1 Overview

The mixed-precision analysis techniques in Chapter 6 have proven useful, but sometimes the single-vs-double precision dichotomy is too strict. Often, developers would find it useful to explore at a finer granularity how sensitive the various parts of a program are to changes in the floating-point precision level. In this chapter, we describe our techniques for general precision-level analysis based on binary modification and reduced precision. We also describe our implementation of these techniques using the CRAFT framework. These techniques have lower overhead than mixed-precision analysis, leading to quicker results.

7.2 Techniques

Our technique builds a version of a target program where every instruction can be executed at any level of precision lower than the original precision (i.e., "reduced" precision). Using an automated search, we use this reduced-precision capability to detect the smallest level of precision that any particular instruction requires to pass verification, assuming that the rest of the program runs in its original precision. The results give an indication of the precision level requirement of each instruction,

```
^      FUNC #7: 0x400b60 "MAIN__"
^        BBLK #87: 0x401088
^r         INSN #37: 0x401096 "mulsd xmm6, xmm10  [ep.f:189]"
inst37_precision=39
^r         INSN #38: 0x40109b "mulsd xmm8, xmm10  [ep.f:190]"
inst38_precision=37
^r         INSN #39: 0x4010a0 "subsd xmm6, xmm9  [ep.f:189]"
inst39_precision=29
^r         INSN #40: 0x4010a5 "subsd xmm8, xmm9  [ep.f:190]"
inst40_precision=27
```

Figure 7.1: Example of reduced-precision configuration

which can be generalized to larger structures by taking the maximal value across all children.

7.2.1 Reduced-precision configurations

Reduced-precision arithmetic can be approximately simulated by truncating the significand to the desired number of bits after each operation. This simple insight provides a way to simulate precisions with levels up to the original precision with minimal program modification. We chose to use truncation rather than rounding because it is faster, simpler to implement, and more conservative than rounding. The exponent is unaffected in the current implementation.

To implement this analysis, we extended our basic configuration file format to allow the user to specify the number of bits for each operation rather than the simple, binary single- or double-precision flag used in the mixed-precision analysis from the previous chapter. Figure 7.1 shows an example excerpt from a reduced-precision configuration file. The lines beginning with a caret are regular program point control

lines, while the other lines give precision levels (in bits) for the operations that will be truncated. A unique instruction ID allows these lines to be correlated despite their relative locations in the configuration file.

The bit precision values range between 0–52. Zero indicates that all of the stored significand should be wiped, leaving the single implied bit to represent a numerical value of one. A bit value of 52 indicates that the entire significand should be preserved, so the analysis does not truncate the result. The other bit value of particular interest is 23, which is the number of bits in a single-precision significand. Truncating a calculation's result to 23 bits represents a rough estimate of using single-precision arithmetic to do that calculation. In fact, it would be a conservative estimate because true single-precision arithmetic would be rounded instead of truncated.

The exponent is unaffected, for both speed and simplicity. If the developer is concerned that single-precision exponents are too narrow, range tracking (as described in Section 4.7.3) could be used to verify that the truncated value magnitudes lie in the range of single-precision numbers.

We also modified the default configuration GUI described in Section 4.5.2 to show these bit values. The GUI uses a white-to-blue gradient to represent bit values between 0–23 (single precision) and a green-to-red gradient to represent bit values between 24–52 (double precision). This view extends the color scheme used in the normal mixed-precision configuration visualization from Section 6.2.1.

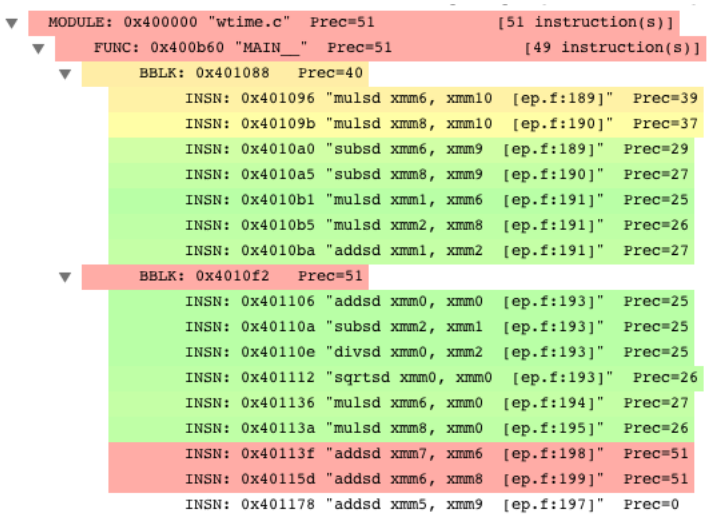


Figure 7.2: Reduced-precision configuration viewer

Figure 7.2 shows the view of an example reduced-precision configuration. In this example, most of the instructions are configured to run close to single precision (bit values in the mid-high 20s). Two instructions (multiplications at the beginning of the excerpt) are configured to run with 37 and 39 bits of precision, and two other instructions (additions at the end) are configured to run with nearly-full double precision.

7.2.2 Binary modification

To reduce the precision of an individual instruction, we augment it by adding code after the instruction that uses bit masking to truncate the resulting floating-point value to the desired number of bits. For instance, if the desired number of bits is 40, then the floating-point value is masked using the binary value `0xfffffffffffff000`, which leaves all bits intact except for the last twelve ($52 - 40 = 12$). To simulate

```
andsd %xmm2 %xmm1              % original instruction

# new inserted code
push %rax
<save %xmm15 to stack>
mov %rax, 0xfffffffe0000000   % truncation constant
pinsrq %xmm15, %rax, 0
andpd %xmm2, %xmm15            % do truncation
<restore %xmm15 from stack>
pop %rax
```

Figure 7.3: Example reduced-precision replacement snippet

single precision (23 bits of significand), the mask value is `0xfffffffe0000000`.

Because the machine code instructions that implement these operations are integer bitwise operations that do not do memory accesses or comparisons, the overhead is low compared to that of the mixed-precision techniques from Chapter 6. Additionally, the search phase overhead is generally smaller because the mutator only needs to replace the instructions that are being analyzed. Because the truncated value is still a valid double-precision floating-point number, the rest of the instructions do not need special treatment as they did with our mixed-precision techniques. The overhead will be directly proportional to the number of instructions that are selected for a reduction in precision.

Figure 7.3 shows an example of a reduced-precision truncation snippet. In this case, the configuration specifies that this instruction (an addition of two XMM registers) should be truncated to 23 bits (roughly single precision). The original instruction comes first, unmodified, followed by the truncation snippet. The snippet is surrounded by saving and restoring instructions to avoid clobbering registers or

unintentionally modifying program state. The snippet then loads a temporary XMM register with the appropriate mask constants and does the actual truncation using a bitwise AND instruction.

7.2.3 Automatic search

Using the reduced-precision technique described in the previous section, we developed an automatic search routine that determines the general precision sensitivity of various program components using a breadth-first search similar to the mixed-precision search described in Section 6.2.3. For this analysis, the tuning parameter domain consists of the integer values between zero and 52 (double precision). The larger domain yields a larger search space, because there are now 52^n total possible configurations to test, rather than 2^n. However, the overhead for each individual test is lower. In addition, the breadth-first nature of the search means that the search will obtain overall precision results for the larger program structures first. For instance, the search will first determine the lowest precision level required by the entire program, followed by the lowest precision level required by individual modules, and so on. Thus, the search functions as a refining process, with the results becoming more detailed as more tests are executed. In this way, the search need not run to completion to be useful.

We also use several other techniques to reduce the search space and improve search convergence speed. These techniques include:

1. Using a binary search for each program component, rather than testing every individual precision level. This works because the precision levels are ordered and because we can assume that if $x > y$, then an instruction executed with x bits of precision will be more accurate than the same instruction executed with y bits of precision. The binary search will find the smallest precision level that passes verification for that component in at most six ($\lceil log_2\ 52 \rceil$) steps. This dramatically reduces the number of configurations tested during a search.

2. Discontinuing the breadth-first search when the precision level of the search point in question falls below a given threshold. The default threshold is 23 bits, which represents single-precision arithmetic. This default is motivated by the assumption that the most common use case is replacing double-precision arithmetic with single-precision arithmetic, rather than with half-precision arithmetic or any other level lower than single precision. Thus, the user is unlikely to need the exact precision level requirement once they know that the level is less than 23 bits. This behavior can be toggled with the `--rprec-split_threshold` search parameter.

3. Discontinuing the breadth-first search when the runtime instruction execution percentage of the search point in question falls below a given threshold. This metric measures each program point for its contribution to the total number of floating-point instructions executed during the program's runtime. Applying a threshold to this metric causes the search to focus on the parts of the program

that dominate its runtime execution. This behavior can be toggled with the `--rprec-runtime_pct_threshold` search parameter. By default, this option is not enabled, because the appropriate value will vary depending on the target application. In practice, we find that setting this threshold at around 5–10% reduces the overall search time significantly while having an insignificant impact on the results (see Section 7.4.1).

4. Providing an option for skipping the top-level (whole-program) configurations, because the overhead will be high relative to subsequent runs. The information gained by their analysis is also of minimal interest, because the end goal is to determine precision sensitivity for the subcomponents of a program. This behavior can be toggled with the `--rprec-skip_app_level` search parameter.

5. Allowing the search to use cached results from previous searches to expedite the current search. Used in conjunction with the runtime execution threshold and the top-level configuration skip described in previous paragraphs, the user may run a search with little initial time commitment. For instance, the user could skip the application level, and stop the search before it explores any program point responsible for less than an arbitrary share (say 5%) of the program's execution. In our experience, such a search could take as little as an hour. After the initial search, the user may choose to look deeper by decreasing the runtime execution threshold. The new search can be configured to use the results from the initial search, essentially bypassing all of the tests

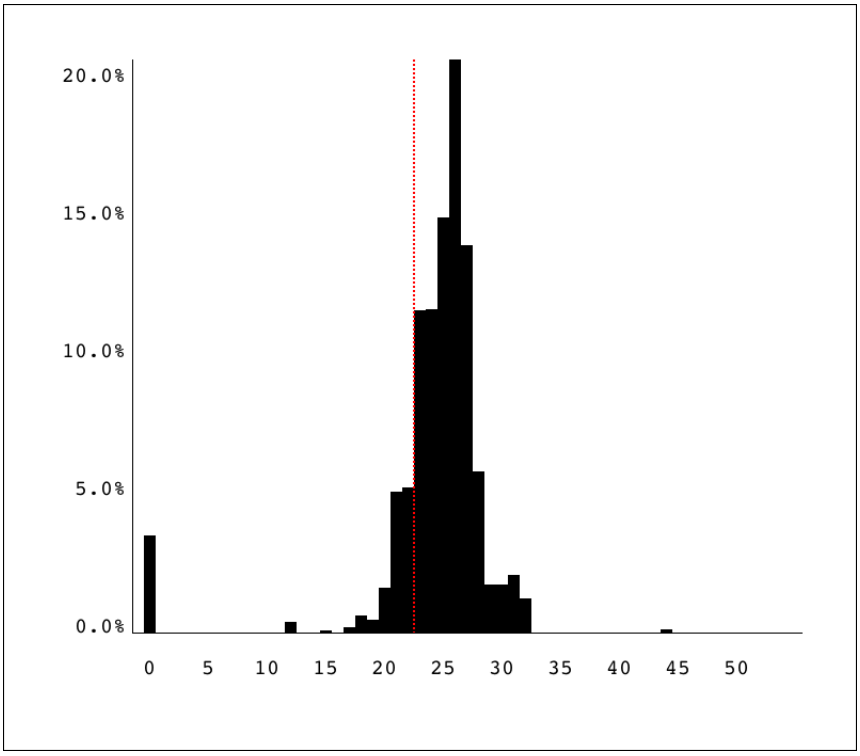


Figure 7.4: Reduced-precision histogram

run by the first search and avoiding any work duplication. We use the term "incremental search" to refer to a search that runs in several iterations using different percentage thresholds, with each iteration utilizing the result cache from previous iterations to avoid test duplication.

7.2.4 Visualization

The results of an automated reduced-precision search are visualized in the standard GUI from Section 4.5.2 using colors as shown in Figure 7.2. This interface allows the user to browse a program's control structure to identify the lower-precision areas by their graphical appearance and to drill down quickly into the regions of interest.

These include regions with high precision requirements that may benefit from higher precision or algorithmic changes, as well as regions with low precision requirements that can probably be replaced with lower precision arithmetic.

We also provide a histogram-based visualization of whole-program sensitivity. The histograms provide an overview of a program's floating-point sensitivity profile as detected using the search described in Section 7.2.3. Figure 7.4 shows an example histogram from the MG NAS benchmark.

The histogram shows the percentage of floating-point instructions executed (vertical axis), grouped into bars by their final reduced-precision configuration value (horizontal axis). The exact values on the vertical axis are less important than the relative sizes of the bars in relationship to the horizontal axis. If the distribution lies mainly to the left of the red bar, then the program is rarely dependent on full double precision. A clustering around the extreme left side of the graph indicates that most of the program uses relatively little precision, showing a high potential for single-precision replacement. Conversely, a clustering around the extreme right side of the graph indicates that most of the program relies on full double precision, indicating little chance for a mixed-precision implementation without major rewriting. A clustering around the red bar is the most interesting outcome, because it implies that the program needs just barely more than single precision accuracy, implying that perhaps a small amount of algorithmic reconfiguration could enable the use of single-precision arithmetic for large portions of the computation.

Benchmark	Original time (s)	Whole-program Overhead (X)	Search Trials	Avg. Search Overhead (X)
bt.A	60.8	33.5	11,610	1.2
cg.A	2.6	4.7	267	1.3
ep.A	9.2	4.0	94	1.3
ft.A	5.2	7.0	234	1.6
lu.A	48.2	11.3	7,246	1.7
mg.A	2.4	9.5	804	1.2
sp.A	42.9	8.9	122	2.1
ua.A	27.7	11.2	12,354	1.6

Figure 7.5: NAS benchmark overhead for whole-program reduced-precision analysis

The histogram values can optionally be binned into a smaller number of bars, and the zero-precision bar (i.e., floating-point computation that has been determined to be non-essential for the final computation) can be excluded. In the results we present in this chapter, we leave the values in their original bins (i.e., a bar for each precision level) to preserve all trends in the original data, and we include the zero-precision bar for completeness. The histogram also includes a red, dotted bar at 23 bits, which represents the cutoff for single precision.

Our visualization interface can also build histograms of the number of binary floating-point instructions by precision level, as opposed to the runtime execution count. This difference corresponds to the distinction between static and dynamic percentages in mixed-precision analysis. However, we have focused on the dynamic instruction execution count because of its greater relevance to overall performance.

	Wall Time (s)		
Benchmark	Mixed	Reduced	Speedup
cg.A	1,305	532	59.2%
ep.A	978	562	42.5%
ft.A	825	411	50.2%
lu.A	514,332	68,365	86.7%
mg.A	2,898	984	66.0%
sp.A	422,371	236,055	44.1%

Figure 7.6: Search wall time comparison

7.3 Benchmarking

Figure 7.5 shows overhead results from the NAS benchmark suite. For these single-core trials, the benchmarks were compiled using the Intel compiler with -O3 optimization. The table gives two overhead numbers. The first overhead number (third column) is from running whole-program reduced precision analysis, averaged over five runs each. The analysis was configured to use 52-bit precision so that the overhead could be measured without affecting the program's execution. The second overhead number (fifth column) is from running an automatic search on the entire program. The fourth column reports the total number of trials, and the fifth column is the overhead averaged over all of those trials. Because many trials apply truncation to only a few instructions, these overheads are low.

7.3.1 Mixed-precision comparison

To compare the overhead for reduced-precision analysis with the overhead for mixed-precision analysis from Chapter 6, we added the capability to run mixed-precision

searches using reduced-precision instrumentation. To implement this capability, we added a new mode for mixed-precision search that does 23-bit reduced-precision replacement wherever the search would normally call for single-precision replacement. Where the search would normally call for double-precision replacement, we instead do 52-bit reduced-precision replacement, which normally becomes optimized to a no-op because no bits are truncated. This mode dramatically reduces the number of instructions that are generated to replace the old instructions, partially because double-precision replacement usually becomes a no-op and partially because the 23-bit reduced-precision snippets are much simpler than the regular mixed-precision replacement snippets.

Figure 7.6 shows the speedup using this mode as measured by the overall search wall time, which ranged from several minutes to several days depending on the benchmark. All searches were run using multiple search threads, and the number of threads was kept constant between the original mixed-precision and reduced-precision runs. The speedup was consistently 40% or higher, a sizeable improvement.

We also examined how closely the results matched the original mixed-precision searches. Because 23-bit replacement is only a rough approximation of single precision, we did not expect the results to be identical. However, in most of the benchmarks, the results were similar, with less than 10% difference as measured using instruction executions. For two benchmarks (MG and SP), we found a roughly 20% difference in the results. In these results, the floating-point instruction exe-

cutions that required higher precision are explained by the conservative nature of reduced precision due to truncation, while the executions that required lower precision are attributed to the corresponding noise caused by different search paths through the parameter space. This experiment shows that reduced-precision analysis can dramatically reduce the time required to run mixed-precision searches, while still achieving similar results.

This experiment also suggests that a hybrid approach might yield good results. One could run a reduced-precision search first to get a general estimate of sensitivity, and then run full mixed-precision searches on select subsets of the program to get more accurate results. This approach would be a more complex variant of the incremental search described earlier in this chapter. Our framework provides the capability for such a hybrid approach, but more research would be required to determine the best point to transition between strategies.

7.4 Results

7.4.1 NAS benchmarks

Figure 7.7 shows histogram results for the NAS benchmarks [14]. These graphs give a more complete picture of the benchmarks' relative precision sensitivities than do the single percentages from Section 6.4.1. Some benchmarks (such as BT and LU) show a strong potential for mixed-precision configurations, with large portions of their sensitivities under the single-precision level. These results match the high

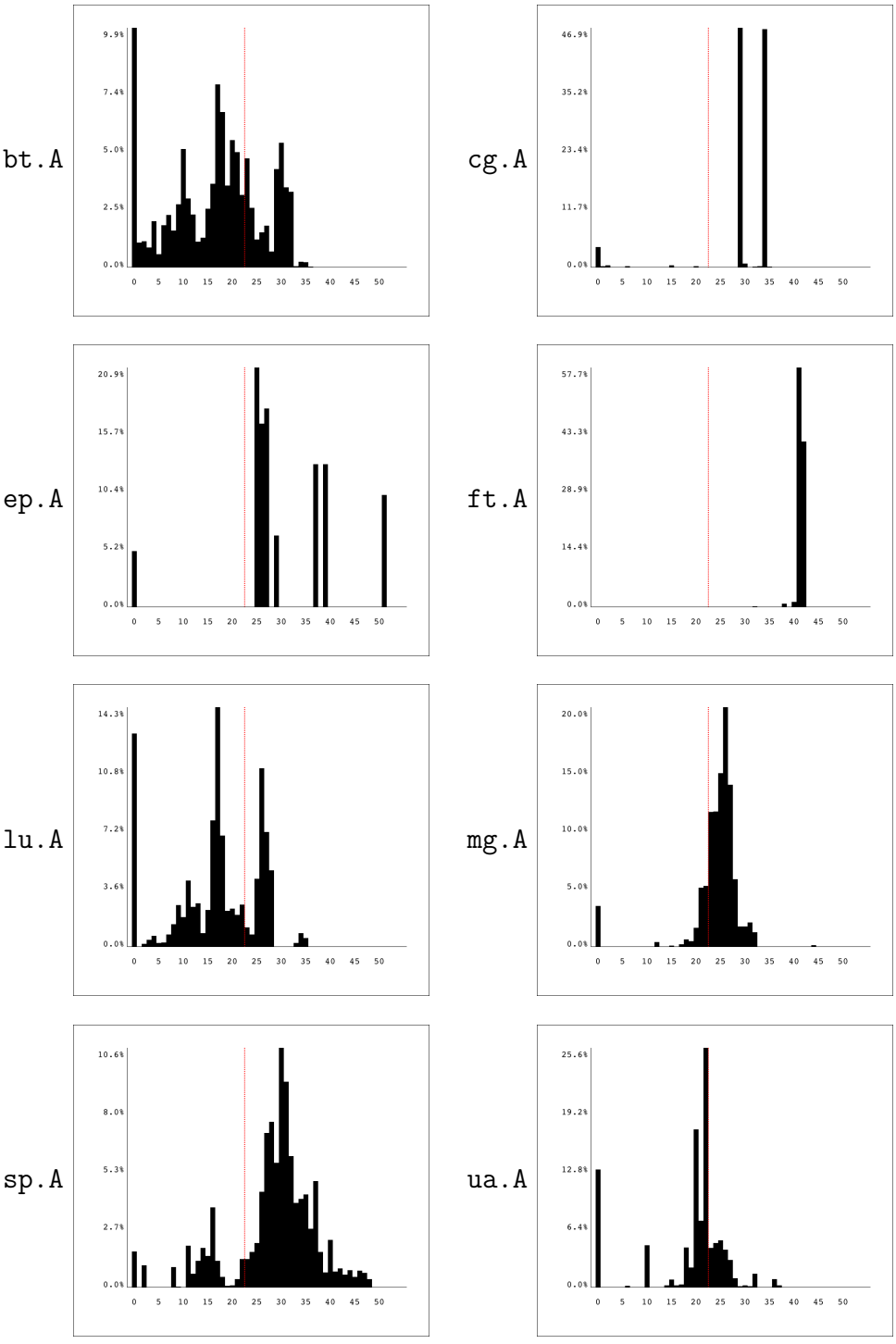


Figure 7.7: Reduced-precision histograms for NAS benchmarks

replacement percentages from Section 6.4.1. Conversely, some benchmarks (such as CG and FT) clearly require more than single precision for most of their computation; thus, these benchmarks seem to have little opportunity for a comprehensive mixed-precision implementation without a major algorithmic shift. Again, these results match the low replacement percentages from Section 6.4.1. Other benchmarks (such as MG and UA) have a clustering of sensitivities around the single-precision mark, indicating that if the computation could be re-written to be slightly less sensitive, the benchmark has a large potential for using single precision.

Figure 7.8 shows the histograms from running an incremental search on MG.W from NAS. Figure 7.9 provides the corresponding search wall times and configuration counts, including the number of configurations tested at a particular increment as well as the cumulative number of configurations tested over the entire search. The histograms show a pattern of refinement that we found to be typical of incremental search results, with the curve beginning heavily-weighted towards the right side of the graph and gradually migrating towards the left as more and more of the program is explored in detail. By the time the search explores any program point accounting for 0.5% or more of the program's instruction executions, the graph closely resembles its final form. Thus, the user may stop the process of incremental searching once the user is satisfied that further refinement of the histogram would provide little insight.

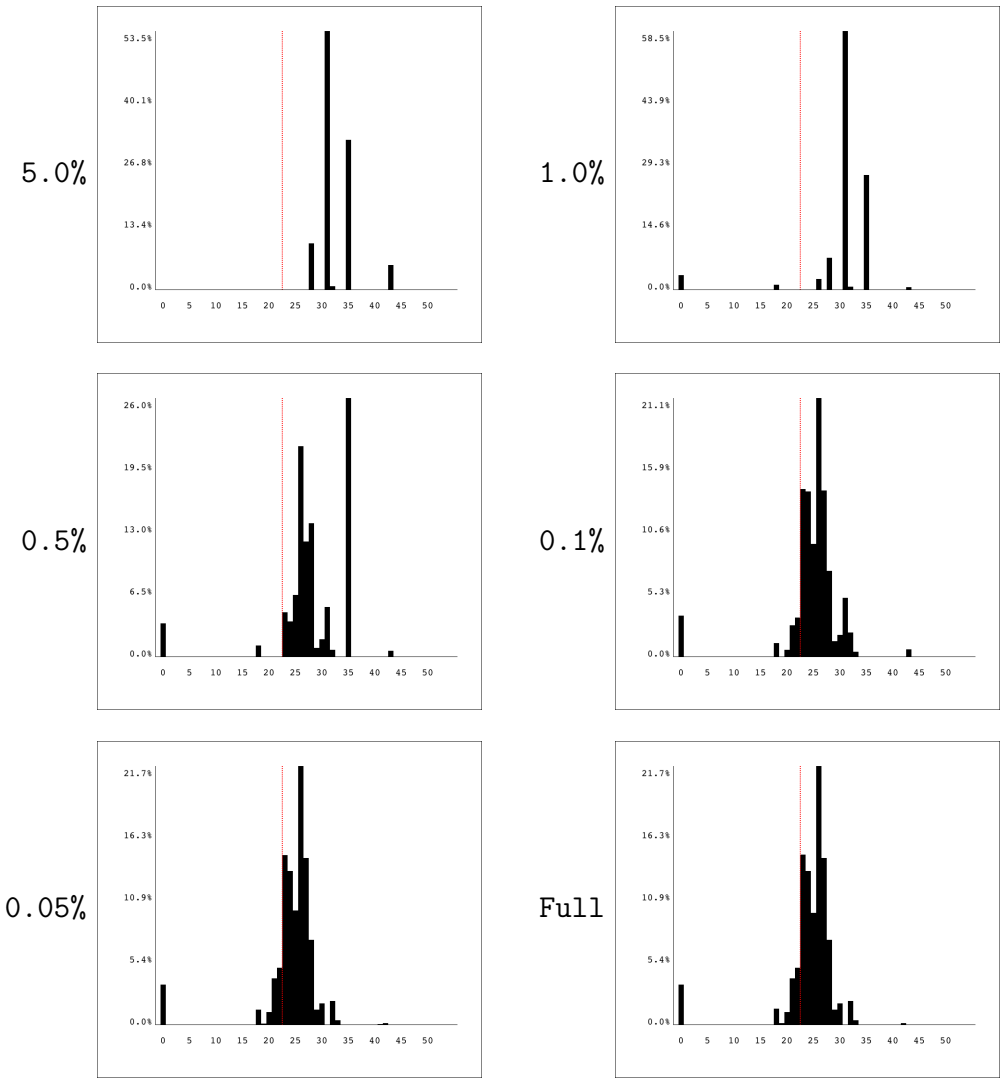


Figure 7.8: Reduced-precision histograms for MG.W incremental search

Benchmark	Threshold	Configs	Cumulative Configs	Time	Cumulative Time
MG.W	10.00%	35	35	00:03:45	00:03:45
	5.00%	12	47	00:01:21	00:04:66
	1.00%	34	81	00:01:27	00:05:93
	0.50%	350	431	00:03:52	00:09:45
	0.10%	546	977	00:06:00	00:15:45
	0.05%	665	1,642	00:08:15	00:23:60
	0.00%	310	1,952	00:05:11	00:28:71

Figure 7.9: Timing results for MG.W incremental search

Benchmark	Original time (s)
chain	1.83
chute	1.08
eam	9.01
lj	3.54
rhodo	61.12

Figure 7.10: LAMMPS benchmarks and running times

7.4.2 LAMMPS

We also ran tests on LAMMPS, a molecular dynamics code that is part of the ASC Sequoia benchmark suite [1]. In these tests, we show that we can obtain precision information about a program in a reasonable amount of analysis time. We examined the "30Aug13" version of LAMMPS with five provided benchmark work loads: chain, chute, eam, lj, and rhodo. We built the LAMMPS with the default compiler (GCC) and the default optimization levels; it contained 56,643 candidates for reduced-precision replacement. Figure 7.10 shows the original run times of the benchmarks.

For each benchmark, we ran several incremental reduced-precision searches, initializing the runtime percentage threshold at 10% and gradually lowering it to 5%, 1%, and finally 0.5%. Each individual search used the results of the previous search(es) as a cache, allowing it to skip directly to the incremental tests for the new threshold. We also skipped the application-level tests for these benchmarks, because they incur a large overhead and only yield information that can be deduced from lower-level results.

Benchmark (Original Time)	Threshold	Configs	Cumulative Configs	Time	Cumulative Time
chain (1.7s)	10.0%	28	28	00:35:05	00:35:05
	5.0%	35	63	00:09:13	00:44:18
	1.0%	104	167	00:11:56	00:56:14
	0.5%	81	248	00:09:03	01:05:17
chute (1.0s)	10.0%	28	28	00:34:49	00:34:49
	5.0%	34	62	00:08:57	00:43:46
	1.0%	99	161	00:11:36	00:55:22
	0.5%	229	390	00:21:28	01:16:50
eam (8.8s)	10.0%	33	33	00:48:10	00:48:10
	5.0%	11	44	00:06:23	00:54:33
	1.0%	440	484	00:47:49	01:42:22
	0.5%	12	496	00:11:34	01:53:56
lj (3.3s)	10.0%	22	22	00:39:42	00:39:42
	5.0%	12	34	00:05:30	00:45:12
	1.0%	160	194	00:21:33	01:06:45
	0.5%	10	204	00:08:25	01:15:10
rhodo (61.7s)	10.0%	41	41	02:14:29	02:14:29
	5.0%	17	58	00:23:34	02:38:03
	1.0%	222	280	00:53:07	03:31:10
	0.5%	188	468	00:45:45	04:16:55

Figure 7.11: LAMMPS benchmark results

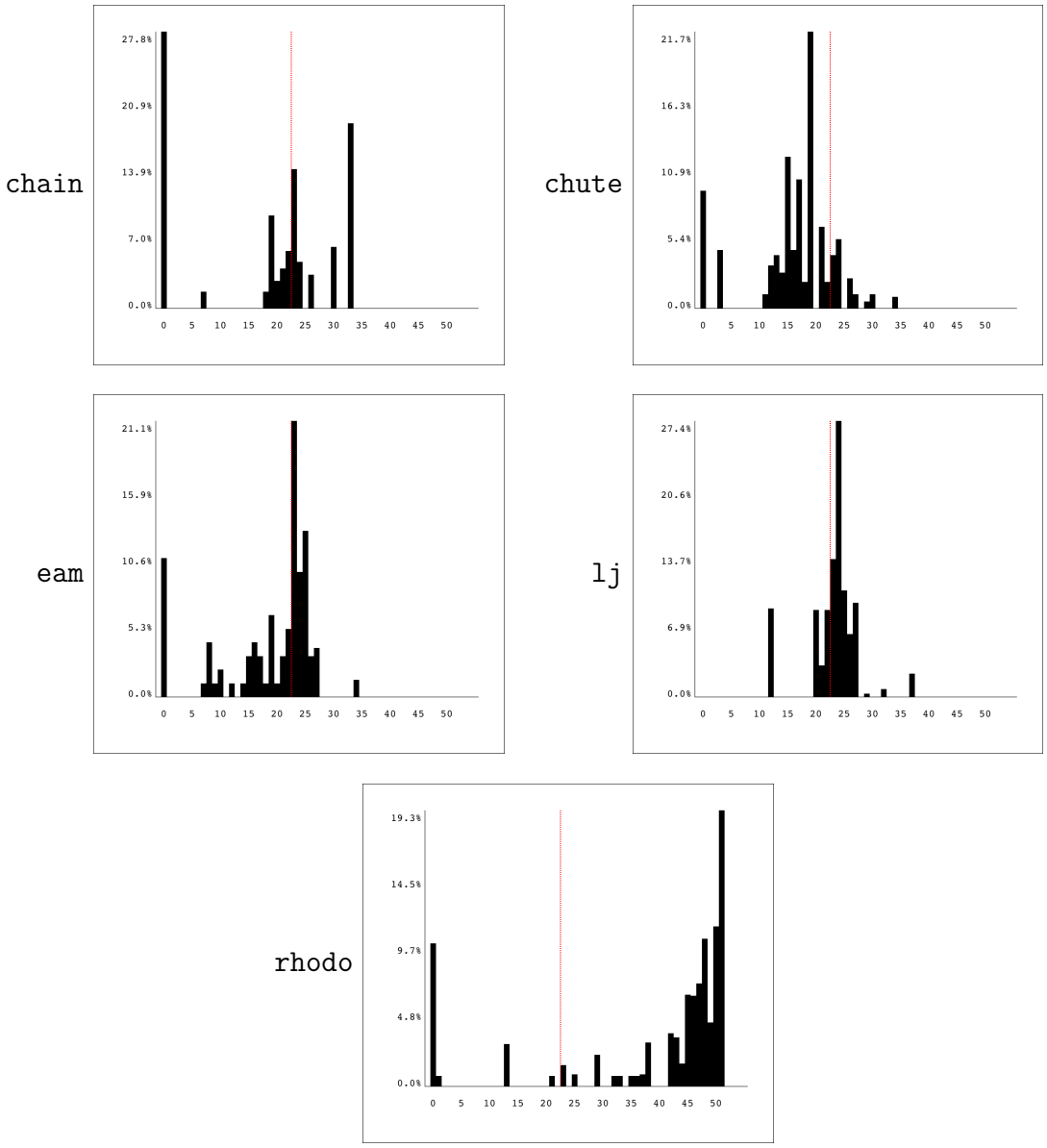


Figure 7.12: Reduced-precision histograms for LAMMPS benchmarks

	chute	eam	lj	rhodo
chute		96.0	99.3	95.9
eam	99.5		90.4	91.5
lj	99.7	88.2		91.0
rhodo	91.0	91.0	91.0	

Figure 7.13: LAMMPS profiling comparisons: unique execution percentages

Figure 7.11 shows the timing results of these tests, and Figure 7.12 shows the final histograms for each benchmark. For the timing results, we see the same general pattern as in Figure 7.9. The number of configurations tested (and therefore the time spent) at every iteration of the search varies, but does not increase dramatically in later iterations. This pattern suggests a "leveling off" of the search, indicating that further computation would be unhelpful. The time spent in individual iterations matches this trend.

The total number of configurations tested does vary, but there are no severe outliers; even benchmarks with widely differing profiles (such as "eam" and "rhodo") test similar numbers of configurations. The total wall time varies as expected with both the number of configurations tested and the original running time of the benchmark itself. The final histograms show a range of precision sensitivity, reinforcing again that floating-point behavior is highly sensitive to particular data sets.

To further explore this variance due to program input, we compared execution profiling data between runs. Figure 7.13 shows the percentage of floating-point instruction executions that were unique to the row-specific benchmark when compared with the column-specific benchmark. Uniqueness is determined using the instruc-

tion addresses in the binary. For instance, 96.0% of the floating-point instruction executions in the "chute" benchmark are attributed to instructions that were never executed in the "eam" benchmark. The high percentages throughout the table show that the benchmarks exercise highly disjoint portions of the full application, partially explaining the widely varying precision sensitivity results. In the future, it would be interesting to explore further analysis of program variance due to differing inputs. For now, the runtime nature of our analysis ensures that such behaviors are captured, provided that the analysis incorporates data sets that are representative of target use cases.

Usually, we were able to get results in under an hour for all portions of the program representing over 5% of the total floating point instructions executed. This timing result shows that the analysis crosses an important threshold for practical viability. An hour-long analysis can easily be fit into a regular workflow; for example, a developer could run the analysis while at lunch or in a meeting. Even for the benchmark that took over a minute to run initially, we had results at the 5% level in under three hours. These experiments show that our reduced-precision analysis techniques provide the user with realistic analysis time requirements as well as the capability to choose the granularity of the search and to benefit from a corresponding reduction in the time to solution.

7.5 Conclusion

We have described techniques for building reduced-precision configurations of floating-point programs, as well as an automatic search process that identifies the precision level sensitivity of various parts of an application. We have also described our implementation of these techniques using the CRAFT framework. Benchmark overhead results indicate that reduced-precision searches are significantly faster than mixed-precision searches. We also show that an incremental technique provides the user with the ability to choose the granularity of the search and to benefit from a corresponding reduction in the time to solution.

Chapter 8

Future Work

8.1 Overview

This chapter describes some of the many directions for future research based on the work presented in this dissertation. In some ways, this dissertation represents an initial contribution to a new sub-field of program analysis devoted to floating-point accuracy and performance. We see many potential opportunities for extending our techniques and integrating them into various other projects. This chapter explores short-term project possibilities as well as ideas for long-term research efforts.

8.2 Short-term work

This section describes short-term ideas for extending the work presented in this dissertation. These extensions would require a moderate amount of work but no major new techniques.

8.2.1 Binary optimization

Thus far, little time has been devoted to optimizing the snippet machine code generated for mixed- and reduced-precision analysis. The code generator is conservative because it must handle the wide variety of instructions in the SSE instruction set.

It saves state around the added code to avoid modifying program semantics, and it uses several temporary registers that must also be preserved. This code could be optimized to use fewer registers and to save less state. The Dyninst library provides a register liveness analysis that could be leveraged for some of this optimization. Some of the most common instruction patterns (`"addsd %xmm0, %xmm1"`, for instance) could also be hard-coded with hand-tuned machine code for even greater speed.

Another, potentially larger improvement could be achieved with greater use of static control flow analysis. Such analysis could determine unreachable paths, for instance, eliminating certain instructions from consideration. Data flow analysis could also reveal calculations that do not influence the final results and can be safely ignored by runtime analysis.

8.2.2 Search optimization

The automatic search routine incurs some overhead. The current implementation uses file-based configuration and work queue structures. Some of these files can be accessed concurrently, but others require exclusive locking to preserve queue semantics and priority ordering. This locking also causes bottlenecks when the underlying file system is slow, such as with some NFS implementations. This slowdown can be significant, especially if the search runs for many hundreds or thousands of iterations. The original file-based design aided in the rapid implementation of a parallel search

process, but could be improved. In particular, the use of a lightweight database component such as SQLite could provide the same transactional semantics required by the search routine at a lower cost than traditional file system locking.

There may also be an opportunity for a performance improvement in searching strategies. Some researchers have worked on improving search convergence times in empirical autotuners by changing or modifying the search algorithm [61, 71, 72]. Our parameter spaces are different than the spaces that these algorithms general explore. Floating-point precision spaces have many variables (instructions) and a few integral values (two in the case of mixed-precision, or fifty-two in the case of reduced-precision), while traditional spaces have fewer variables with wider (and possibly real-valued) ranges. However, some insights from these algorithms might lead to faster searches if applied appropriately to our domain.

8.2.3 Analysis extension

Our automated search processes focus on replacing individual program components. The final configurations presented to the user are a combination of many individual configurations, and the resulting combination may or may not pass verification as-is. At this point, the search process could refine the final configuration by making it more conservative to find a whole-program configuration that passes verification. In the case of mixed-precision results, this refining would mean experimenting with various combinations of individual replacements. We have no clear intuition about

how to build these combinations to find a passing configuration efficiently; this task would be a subject of research and experimentation. In the case of reduced-precision results, the search could start raising precision levels across larger components or the entire program.

8.2.4 Analysis composition

This dissertation has described both a general framework for floating-point program analysis as well as particular analyses implemented using this framework. Sometimes, we have included components of one analysis in others; for instance, most analyses implement instruction counting. However, we have not attempted to more generally apply multiple analyses simultaneously. This composition of analyses could lead to interesting side effects or new insights, but would pose some interesting challenges, particularly when both analyses replace the existing instruction rather than augmenting or instrumenting it. It is not clear what the semantics of the composition of such analyses should be. However, we imagine that good insights could result from certain combinations, such as running cancellation detection or range tracking alongside mixed-precision replacement. For instance, Benz et al. report good results running cancellation detection alongside a heavyweight shadow value analysis [19].

8.2.5 Platform ports

The CRAFT framework is implemented for the x86_64 and SSE instruction sets. These instruction sets form the predominate platform in high-performance computing; however, others are gaining prominence. The PowerPC platform, for instance, is the basis of the IBM Blue Gene architecture used by several of the top supercomputers. The AMD ARM platform is also gaining recognition in high-performance computing, partially because of the low power usage of ARM-based chips. The CRAFT framework could be modified to work on these architectures, although the code generator would need to be rewritten for each target architecture. Alternatively, it could be rewritten to use an external code generator that already supports these platforms. Dyninst's own code generator is an obvious choice, although it would first need to be extended to support more robust floating-point code generation.

In addition to traditional CPUs, high-performance computing is increasingly utilizing floating-point accelerators such as GPUs to build hybrid platforms. Nvidia cards and the CUDA framework are quickly rising in popularity for HPC application development [10, 28, 60]. More generic alternatives such as OpenCL support both GPU and CPU computation on several platforms including Nvidia, AMD, Intel, and ARM. CRAFT has no support for analyzing GPU code, but such support may become important if GPU computing continues to expand its market share in HPC. Adding analysis for GPUs may require some redesign in the framework to support

the master-slave nature of most CPU/GPU hybrid platforms.

8.2.6 Extended case studies

Although several proof-of-concept case studies appear in dissertation, we can always do more extended studies. In particular, none of the studies presented in this dissertation were run at large scale (hundreds or thousands of cores), which would require greater support for distributed programming frameworks like MPI and OpenMP. In the future, we want to work closely with an application development team to integrate analysis runs into the development cycle. This interaction would allow the domain experts to review the results of the analysis and make high-level algorithmic choices. We expect that such collaboration would result in significant performance improvements and bandwidth savings.

8.3 Long-term work

This section describes longer-term extensions of the work presented in this dissertation. In this section, we discuss the various aspects of an end-to-end floating-point tuning framework. Such a framework could be integrated throughout the development process, including development, testing, and verification, to help developers maximize both accuracy and performance.

8.3.1 Runtime adaptation

In Chapters 6 and 7, we presented techniques for finding mixed-precision configurations. These techniques are tailored to the specific input set and runtime parameters given by the user, and so the resulting configuration may not generalize to different inputs. One way to generalize such results is to generate several mixed-precision variants of a program and use a runtime adaptation system to switch between the variants based on accuracy feedback. Ideally, such a system would be able to pause or partially roll back execution, switch variants in memory, and migrate data and program state before resuming the program. Bao and Zhang [17] describe a less ambitious version of such a system; their system terminates the calculation when the data has become compromised by rounding error, restarting in a higher precision. It would be interesting to pursue a more flexible version of runtime adaptation.

8.3.2 Compiler-based implementation

This dissertation has addressed floating-point precision analysis of compiled binary targets. In the future, we want to integrate similar analyses into a compiler framework such as GCC or LLVM. Such an integration would result in great gains in portability and performance overhead.

Portability would be improved because the analysis could be encoded in the compiler's intermediate representation, allowing the analysis to work with any front-end language or back-end code generation platform provided by the compiler frame-

work. Analysis overhead could also be greatly improved with compiler integration. Running a mixed-precision configuration incurs a performance overhead. This overhead is acceptable because the techniques described in this dissertation deal with verifying the correctness of mixed-precision configurations rather than any actual performance gain associated with them. However, the integration of these analyses into a compiler framework would allow the compiler to incorporate analysis code during its optimization phases. This incorporation of analysis code would dramatically reduce the overhead of instrumentation because most of the state-preserving code would not be needed. In fact, the optimizations applied by a compiler framework could potentially enable performance gains even while testing mixed-precision configurations. Other researchers [67] have already reported some success using these frameworks to analyze floating-point behavior.

To implement support in a compiler framework, we imagine the addition of several compiler command-line options to control the mixed-precision implementation. Alternatively, the source code could be augmented with annotations to tag candidates for replacement. An external search framework would still be needed to generate configurations, perhaps as part of the application's standard test suite. The result could be a short tuning search that could potentially be executed during every compilation. Alternatively, a larger search could run before the release of each major or minor version. These searches would provide repeated feedback to the developers regarding the floating-point behavior of their program.

8.3.3 IDE and development cycle integration

With the compiler integration described in the previous section, running searches throughout the software development phase could be feasible. The results of these searches could be reported to the user in their IDE, highlighting regions of the program that are replaceable with single precision, as well as those regions that have particularly high or low precision sensitivities. These results would provide feedback to the developer as they develop the code. For instance, if a particular region can be entirely replaced with single precision, the analysis results may prompt the developer to transfer that logic to a GPU kernel. Alternatively, if a particular loop or subroutine has high precision sensitivity, the analysis results may prompt the developer to redesign that particular region for lower sensitivity.

Test suites could also do regular precision-level sensitivity regression analyses. This type of test would be similar to performance regression tests, which are intended to flag changes that significantly affected a program's performance. Often, these regressions were unintended side effects or bugs. Flagging regressions in floating-point sensitivity could highlight seemingly-harmless code edits that negatively affected numerical stability. In addition, the mere presence of the precision tests might incentivize programmers to consider more carefully the floating-point behavior of code that they write.

Once autotuning results have been integrated into IDEs and source code tools, the next step could potentially include automatic code transformation recommen-

dations. The goal of such a technique would be to take the precision-level information gained by mixed- and reduced-precision analyses and turn them into candidate source-code level modifications. This transformation could potentially require sophisticated static analysis and program slicing to determine which parts of source code are associated with the lower-level binary structures. The compiler itself could provide valuable insights for this process by exploiting intermediate representations and debug information. For instance, the compiler could tag all operands in an intermediate representation with their corresponding source variables. This information would make the instruction-to-variable mapping process much easier. We also anticipate opportunities to apply machine learning algorithms for recognizing code patterns to improve code transformation suggestions.

8.3.4 Performance modeling

Lowering the precision of a region of code does not always result in an overall speedup. Sometimes, the additional costs of data conversion at the region's entrance and exit outweigh the speed gains of single-precision arithmetic. Building a model of program performance could help identify and prevent such situations. Such a model would predict the performance of each mixed-precision variant. Poorly-performing variants could be skipped during the empirical validation runs.

8.3.5 Static analysis integration

While CRAFT is primarily a runtime analysis framework, the integration of static analysis could afford several opportunities to improve runtime analyses or prove properties about floating-point behavior. In previous sections, we discussed how static analysis might improve the overhead of our techniques by doing smarter instrumentation. In addition, we have some larger ideas for extending the CRAFT analyses using static analysis techniques.

In particular, the interval and affine arithmetic analyses discussed in Section 3.3 could provide initial conservative bounds on a program's arithmetic precision requirements. These results could be used to jump-start empirical runtime searches by identifying good candidates for replacement. Because static analyses do not take into account dataset-specific program behaviors, these results would need to be carefully examined and validated with actual program traces.

Our analysis techniques could also have some application to formal program verification. In recent years, there have been many research efforts in the programming language and security fields that have focused on automated program verification. This verification is usually stated in the form of various security properties, which are verified using a formal analysis method. Thus far, we are aware of no attempt to certify the floating-point accuracy of software using such a technique. The greatest obstacle is that floating-point behavior is highly dependent on the actual runtime data, and thus static verifiers have difficulty modeling such be-

haviors. However, a runtime environment like CRAFT could aid in the verification of floating-point properties by generating execution traces with probes added by prior static analysis. The results of these traces in conjunction with formal analysis proofs could potentially provide reasonable guarantees of floating-point correctness and performance properties.

Chapter 9

Conclusion

THESIS: *Automated runtime analysis techniques can inform application developers regarding floating-point behavior, and can provide insights to guide developers towards reducing precision with minimal impact on accuracy.*

We have constructively proven this thesis by developing techniques and tools for automated floating-point precision analysis. This dissertation showed the practicality of automated techniques for analyzing floating-point programs, and represents a first step towards the long-term vision of autotuned floating-point precision and performance.

Specifically, we built a software framework (CRAFT) that enables floating-point program analysis at the binary level with a variety of applications. The framework uses binary instrumentation and modification to analyze target programs at runtime, and provides graphical interfaces to interpret the results. We have demonstrated the framework's capability by using it to implement many types of analysis. This includes simple analyses such as instruction counting, NaN detection, and range tracking, as well as three more complex analyses:

1. We have developed a technique for detecting numerical cancellation. This technique instruments addition and subtraction operations, checking their

operands for cancellation events at runtime. The analysis reports individual cancellations as well as aggregate cancellation statistics. We have demonstrated this analysis in various contexts, showing its efficacy and potential applications.

2. We have developed techniques for implementing mixed-precision configurations and automatically finding valid mixed-precision replacements. This technique modifies a binary, replacing each floating-point instruction with new code that runs the original operation in the desired precision (single or double). Developers can now quickly prototype mixed-precision configurations without modifying the source code or recompiling. We have also described an automated search process that uses this prototyping technique to find individual program components that can be reconfigured to use single-precision arithmetic without causing the overall program to fail a verification test. This search process can focus on either the arithmetic operations themselves or the memory locations accessed by the operations. We have demonstrated this technique by running it on a variety of applications, highlighting the insights provided by the analysis. In one case, we obtained a speedup of nearly 2X by reconfiguring a microkernel to use single-precision arithmetic.

3. We have developed a technique for performing generalized reduced-precision analysis. This technique truncates the results of floating-point operations, simulating the use of lower-precision arithmetic. We have also described an

automated search process that uses this truncation technique to determine the precision level requirements of each program component. This search can be run incrementally to gain overview insights more quickly, with further iterations of the search leading to more detailed results. The result is a profile of a program's precision level sensitivity, reported in histograms and program structure graphs. We have shown that this technique provides similar results to mixed-precision analysis with much lower overhead.

All of these analyses prove the ability of automated runtime techniques to inform developers regarding floating-point behavior, and the latter two analyses (mixed and reduced precision) specifically fulfill the goal of providing insights towards reducing precision with a minimal impact on accuracy.

In conclusion, this dissertation has shown that automated techniques can provide insights regarding floating-point behavior as well as guidance towards acceptable precision level reduction. This work provides immediately useful, practical tools as well as a basis for further research. These techniques represent novel contributions to the fields of high performance computing and program analysis, and serve as the first major step towards the larger vision of automated extreme-scale floating-point precision and performance tuning.

Appendices

Appendix A

Sample Application: Sum2Pi_X

A.1 Overview

This appendix demonstrates the techniques presented in this dissertation by applying the full CRAFT analysis tool suite to a simple example target program. The program calculates $\pi \cdot x$ using a computational-heavy summation method that magnifies rounding error over the sequence of many operations. The program then compares the result with the correct answer as determined by directly calculating $\pi \cdot x$ using a single multiplication, and uses an epsilon value of 10^{-7} to determine whether a particular run passes or fails. For the example tests presented here, $x = 2000$.

Section A.7 lists the program's source code, and Section A.8 lists its makefile. The program's source code contains two main floating-point type designations: `real` and `sum_type`. The former designates a standard-precision floating-point data type, while the latter designates an important accumulator value that requires higher precision. These types are bound to either single- or double-precision types at compile time, allowing the project makefile to build multiple versions of the program. By default, the makefile builds three versions of the original program: 1) Double, where both types are double precision, 2) Single, where both types are single precision,

		real	
		32	64
sum_type	32	Single (fail)	(fail)
	64	Mixed (pass)	Double (pass)

Figure A.1: Sum2Pi_X versions and results

and 3) Mixed, where `real` is single precision and `sum_type` is double precision. The Double version is considered to be the "original" version.

Figure A.1 shows the four combinations of data types that can be selected at compile time. The diagonal elements represent full-single-precision and full-double-precision versions; the single-precision version fails verification and the double-precision version passes verification. The lower-left corner represents an intuitive mixed-precision implementation: the important summation variables use double precision and the rest of the variables use single precision. This version passes verification, and therefore represents the goal of a mixed-precision version of the original program that passes verification. Later sections show how this configuration might be discovered using insights from the analyses described in this dissertation. The upper-right corner represents a low-precision summation type and a high-precision standard type; this variant fails verification and is of little interest. Figure A.2 shows the output for running the three versions of the program built by the make-file, showing that the Double and Mixed versions pass verification while the Single version fails.

```
$ ./sum2pi_x
===  Sum2PI_X  ===
sizeof(real)=8
sizeof(sum_type)=8
  RESULT:  6.2831852954768319e+03
  CORRECT: 6.2831853071800006e+03
  ERROR:   1.8626171386763082e-09
  THRESH:  9.9999999999999995e-08
SUM2PI_X - SUCCESSFUL!


$ ./sum2pi_x-float
===  Sum2PI_X  ===
sizeof(real)=4
sizeof(sum_type)=4
  RESULT:  6.2832021484375000e+03
  CORRECT: 6.2831855468750000e+03
  ERROR:   2.6422205792187015e-06
  THRESH:  9.9999999999999995e-08
SUM2PI_X - FAILED!!!


$ ./sum2pi_x-mixed
===  Sum2PI_X  ===
sizeof(real)=4
sizeof(sum_type)=8
  RESULT:  6.2831850051879883e+03
  CORRECT: 6.2831855468750000e+03
  ERROR:   8.6212160965715157e-08
  THRESH:  9.9999999999999995e-08
SUM2PI_X - SUCCESSFUL!
```

Figure A.2: Sum2Pi_X correctness test results

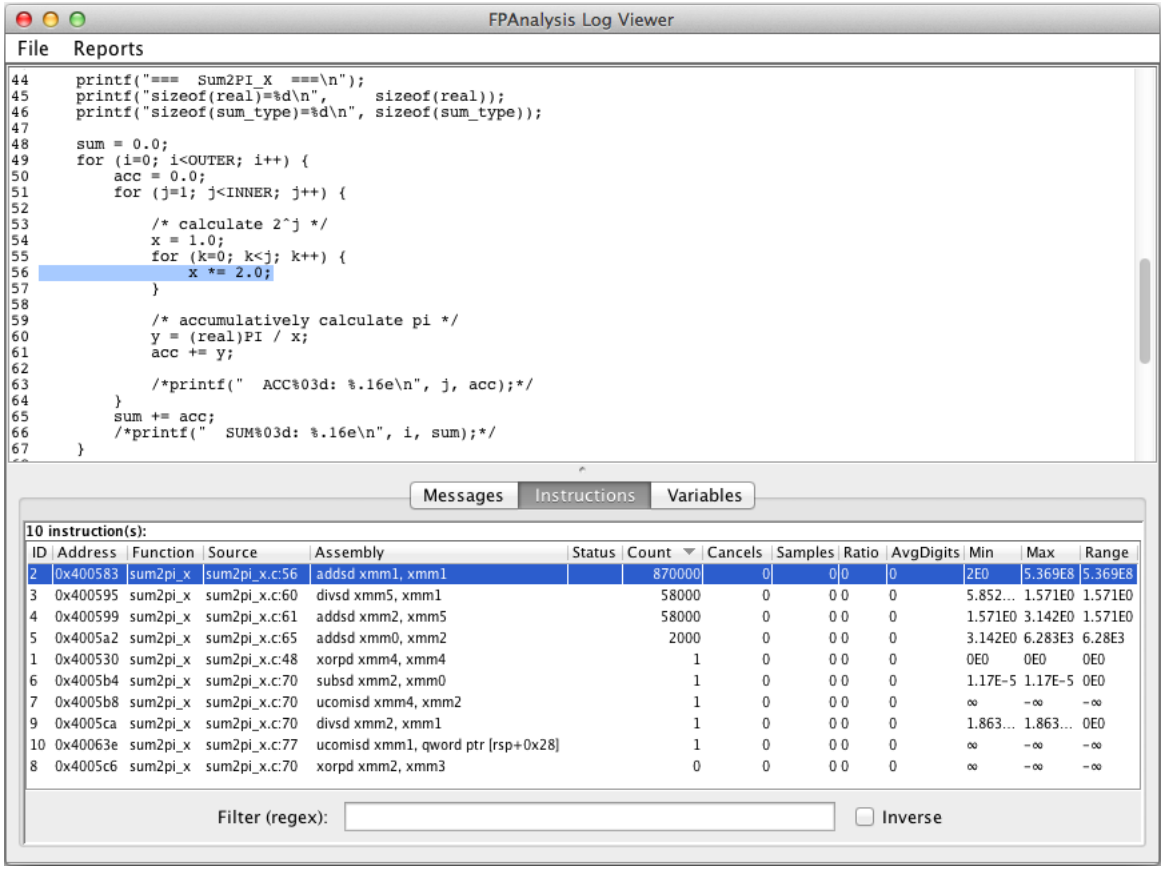


Figure A.3: Sum2Pi_X count and range results

A.2 Preliminary analyses

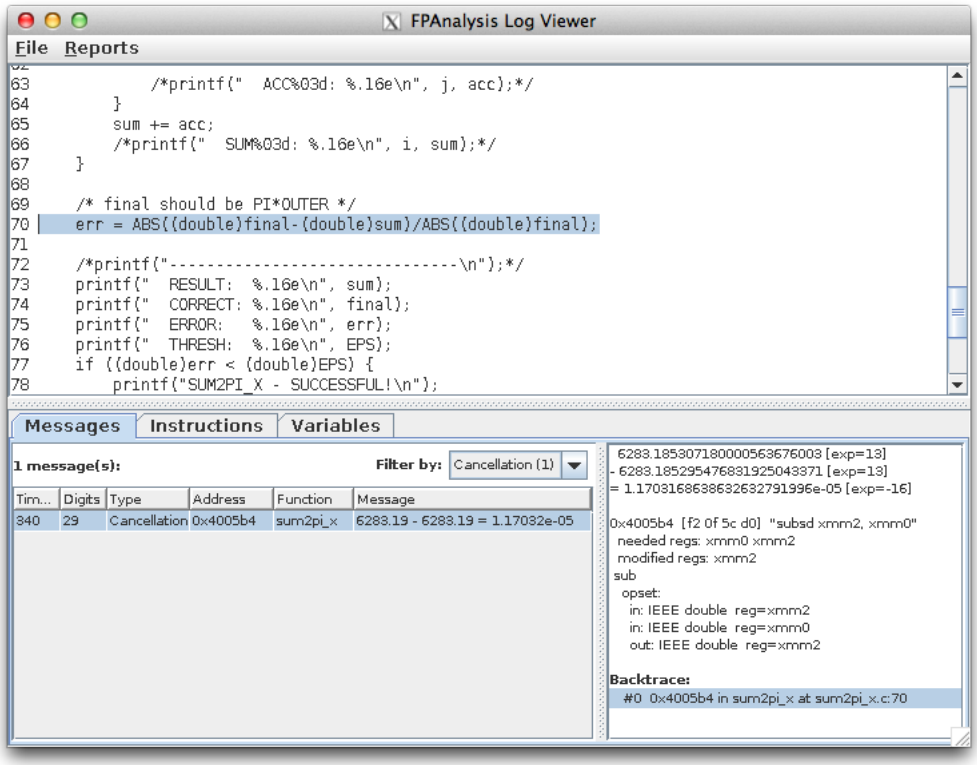
We first apply our instruction count, NaN detection, and range tracking analyses to the example program. Examining the code, we expect that the outer loop (calculating $\pi \cdot x$) to be executed 2,000 times, given that OUTER is set to 2,000. We expect the inner loop (calculating π) to be executed $29 \cdot 2{,}000 = 58{,}000$ times, given that INNER is arbitrarily set to 30, and the loop iterates from 1 to INNER (exclusive). In addition, we expect the inner-most (2^j) loop to be executed $2{,}000 \cdot \sum_{i=1}^{29} i = 2{,}000 \cdot 435 = 87{,}000$ times. Further, we expect the inner accumulation

to see a maximum value of $\pi \approx 3.142$ and the outer accumulation to see a maximum value of $\pi \cdot x \approx 6{,}283$. Finally, we would expect not to see any NaN values during a normal run.

Figure A.3 shows the results of running instruction count and range tracking analyses on the Double version of the program. The count results ("Count" column) match our expected execution counts exactly, demonstrating the execution differences between the three loops. The range results ("Min", "Max", and "Range" columns) show the ranges of individual instructions and also match our expectations. We also ran our NaN detection analysis, which verified that no NaN values are encountered during a normal run.

A.3 Cancellation detection

The Sum2Pi_X has few subtraction operations, so the program's execution yields little cancellation. Figure A.4 shows cancellation detection results obtained by analyzing the Double version of the program, using the default detection threshold. The first screenshot highlights the single case of cancellation detected during a successful run. The lower portion of the screen contains information about the cancellation itself, while the upper portion shows the corresponding source code location. The second screenshot shows the contents of the "Instructions" tab with the single instruction. If the analysis had found more cancellations, this tab would show aggregate data, such as the total number of cancellations per instruction and the average



1 instruction(s):

ID	Address	Function	Source	Assembly	Status	Count	Cancels	Samples	Ratio	AvgDigits
1	0x4005b4	sum2pi_x	sum2pi_x.c:70	subsd xmm2, xmm0		1	1	1	1	29

Figure A.4: Sum2Pi_X cancellation results

Version	Significand Size (Bits)	Canceled Bits
Single	23	18
Mixed	23/52	23
Double	52	29

Figure A.5: Sum2Pi_X cancellation

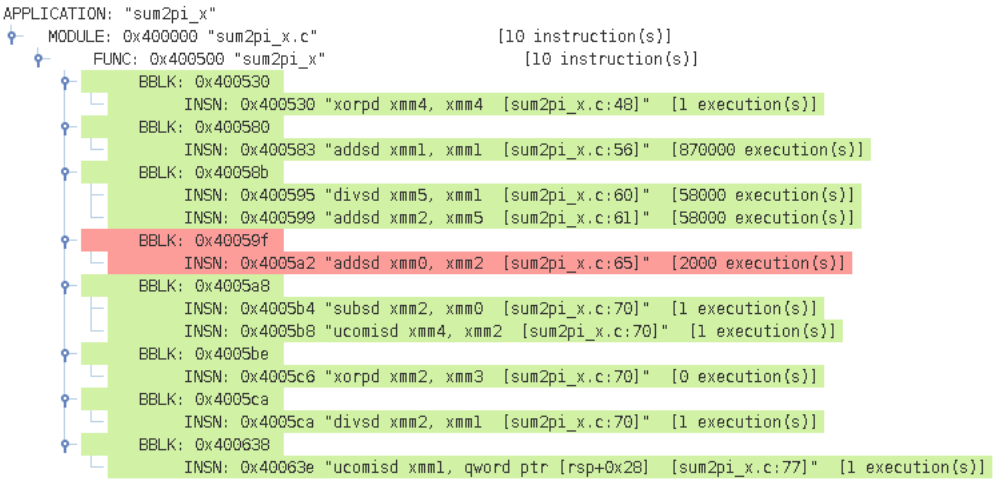
number of bits canceled.

The single cancellation detected takes place during the verification comparison, which subtracts the computed sum from the reference final value. In this context, the cancellation indicates that the result has twenty-nine binary digits (bits) of precision (roughly nine or ten decimal digits), because there were twenty-nine bits that canceled during the subtraction. The single-precision version yields a single cancellation of eighteen bits, which indicates a much lower level of accuracy (around six decimal digits) for the answer. Because this level of accuracy is close to the epsilon value used for verification, we might guess that we could do most of the computation in single precision. Indeed, the Mixed version of the program yields a single cancellation of twenty-three bits, which lies between the two whole-program precision levels, and is slightly closer to the single-precision version. Figure A.5 shows these values alongside the number of significand bits that were present in each version of the program. Thus, even though the application does little subtraction, the cancellation results still provide insight in the overall program analysis.

A.4 Mixed-precision analysis

The program contains four major candidate instructions for single-precision replacement, as indicated by comments in the source code in Section A.7. We ran mixed-precision analysis (as described in Chapter 6), which determined that all except one of the instructions could be individually replaced with single-precision while still passing verification. The final configuration (with all three replaced) also passed verification. Figure A.6 shows the results. These results are confirmed by the Mixed version of the program, which represents a specially-compiled version of the final configuration obtained using our analysis. While this configuration was known *a priori* for this simple example, this exercise demonstrates how such a configuration might be constructed using insights provided by our analysis.

This particular program does not experience a speedup in its mixed-precision configuration. This lack of speedup is because the program does not use packed SSE arithmetic or specialized hardware that could lead to a computational improvement in single precision, nor does it have high memory requirements that could lead to a storage or bandwidth improvement. These characteristics were necessary to keep the example simple; we show in other parts of the dissertation that performance improvements can be obtained using these techniques.




```
44          printf("===  Sum2PI_X  ===\n");
45          printf("sizeof(real)=%d\n",     sizeof(real));
46          printf("sizeof(sum_type)=%d\n", sizeof(sum_type));
47
48 0        sum = 0.0;
49          for (i=0; i<OUTER; i++) {
50              acc = 0.0;
51              for (j=1; j<INNER; j++) {
52
53                  /* calculate 2^j */
54                  x = 1.0;
55                  for (k=0; k<j; k++) {
56 0                    x *= 2.0;
57                  }
58
59                  /* accumulatively calculate pi */
60 0                y = (real)PI / x;
61 0                acc += y;
62
63                  /*printf("  ACC%03d: %.16e\n", j, acc);*/
64              }
65 0            sum += acc;
66              /*printf("  SUM%03d: %.16e\n", i, sum);*/
67          }
68
69          /* final should be PI*OUTER */
70 0000     err = ABS((double)final-(double)sum)/ABS((double)final);
71
72          /*printf("------------------------------\n");*/
73          printf("  RESULT:  %.16e\n", sum);
74          printf("  CORRECT: %.16e\n", final);
75          printf("  ERROR:   %.16e\n", err);
76          printf("  THRESH:  %.16e\n", EPS);
77 0        if ((double)err < (double)EPS) {
78              printf("SUM2PI_X - SUCCESSFUL!\n");
79          } else {
80              printf("SUM2PI_X - FAILED!!!\n");
81          }
82          return 0;
```

Figure A.6: Sum2Pi_X mixed-precision results

A.5 Reduced-precision analysis

We also ran our generalized reduced-precision sensitivity analysis (as described in Chapter 7). As in the previous section, the analysis identified four primary candidates for analysis. Figure A.7 shows the results. The analysis determined that the four candidates required 0, 22, 27, and 32 bits of precision.

First, the analysis reveals that the 2^j calculation uses zero bits of precision. This result indicates that the addition operation (which doubles the value each time) can be completely wiped of its significand, while still allowing the computation to proceed and pass final verification. This insight may seem surprising at first, but on closer examination, is entirely correct. The nature of floating-point representation allows powers of two to be represented purely using the exponent field and the implicit significand bit that stores a value of one. Thus, this instruction does not need a significand, and could potentially be replaced by a bit-shift operation that would be considerably faster. Of course, such a computation could be easily avoided altogether in this simple case, but in a larger program such computations could be missed during manual inspections. Our analysis provides an automated technique leading to the insight that an alternate computation method might be beneficial in such cases.

Second, the analysis reveals a sensitivity requirement of 32 bits for the outer accumulator. This result confirms the results obtained using the mixed-precision analysis in the previous section, and could have provided similar insight leading to

a mixed-precision configuration.

Third, the analysis reveals a sensitivity requirement of 27 bits for one of the instructions that the mixed-precision analysis determined could be replaced. Because single precision provides 23 bits for the significand, the reported value of 27 bits is overly cautious. This result demonstrates the conservative nature of the reduced-precision analysis, because it uses truncation rather than rounding. Such differences highlight the value of using multiple analyses on a single target program.

A.6 Conclusion

Taken together, the analyses described in this dissertation provide a comprehensive examination of the floating-point behavior of the simple example described in this appendix. The tools provided by CRAFT allow us to learn about the execution profile, including instruction count and dynamic ranges as well as instruction-level precision sensitivity. The analysis also yielded a mixed-precision configuration based on insights from mixed-precision and reduced-precision analysis; this configuration is verified using a specially-compiled version of the original program. In conclusion, this exercise demonstrates the efficacy and ease of use of the techniques described in this dissertation.

```
APPLICATION: "sum2pi_x"  Prec=32
  MODULE: 0x400000 "sum2pi_x.c"  Prec=32          [6 instruction(s)]
    FUNC: 0x400500 "sum2pi_x"  Prec=32            [6 instruction(s)]
      BBLK: 0x400580   Prec=0
        INSN: 0x400583 "addsd xmm1, xmm1  [sum2pi_x.c:56]"  Prec=0  [870000 execution(s)]
      BBLK: 0x40058b   Prec=27
        INSN: 0x400595 "divsd xmm5, xmm1  [sum2pi_x.c:60]"  Prec=22  [58000 execution(s)]
        INSN: 0x400599 "addsd xmm2, xmm5  [sum2pi_x.c:61]"  Prec=27  [58000 execution(s)]
      BBLK: 0x40059f   Prec=32
        INSN: 0x4005a2 "addsd xmm0, xmm2  [sum2pi_x.c:65]"  Prec=32  [2000 execution(s)]
      BBLK: 0x4005a8   Prec=0
        INSN: 0x4005b4 "subsd xmm2, xmm0  [sum2pi_x.c:70]"  Prec=0  [1 execution(s)]
      BBLK: 0x4005ca   Prec=0
        INSN: 0x4005ca "divsd xmm2, xmm1  [sum2pi_x.c:70]"  Prec=0  [1 execution(s)]
```

```
44        printf("===  Sum2PI_X  ===\n");
45        printf("sizeof(real)=%d\n",     sizeof(real));
46        printf("sizeof(sum_type)=%d\n", sizeof(sum_type));
47
48        sum = 0.0;
49        for (i=0; i<OUTER; i++) {
50            acc = 0.0;
51            for (j=1; j<INNER; j++) {
52
53                /* calculate 2^j */
54                x = 1.0;
55                for (k=0; k<j; k++) {
56                    x *= 2.0;
57                }
58
59                /* accumulatively calculate pi */
60                y = (real)PI / x;
61                acc += y;
62
63                /*printf("  ACC%03d: %.16e\n", j, acc);*/
64            }
65            sum += acc;
66            /*printf("  SUM%03d: %.16e\n", i, sum);*/
67        }
68
69        /* final should be PI*OUTER */
70        err = ABS((double)final-(double)sum)/ABS((double)final);
71
72        /*printf("------------------------------\n");*/
73        printf("  RESULT:  %.16e\n", sum);
74        printf("  CORRECT: %.16e\n", final);
75        printf("  ERROR:   %.16e\n", err);
76        printf("  THRESH:  %.16e\n", EPS);
77        if ((double)err < (double)EPS) {
78            printf("SUM2PI_X - SUCCESSFUL!\n");
79        } else {
80            printf("SUM2PI_X - FAILED!!!\n");
81        }
82        return 0;
```

Figure A.7: Sum2Pi_X reduced-precision results

A.7 Source: sum2pi_x.c

```
/**
 * sum2pi_x.c
 *
 * CRAFT demo app. Calculates pi*x in a computationally-heavy way that
 * demonstrates how to use CRAFT.
 *
 * September 2013
 */

#include <stdio.h>
#include <stdlib.h>

/* macros */
#define ABS(x) ( ((x) < 0.0) ? (-(x)) : (x) )

/* constants */
#define PI     3.14159265359
#define EPS    1e-7

/* loop  iterations; OUTER is X */
#define INNER    30
#define OUTER    2000

/* 'real' is double-precision if not pre-defined */
#ifndef real
#define real double
#endif

/* sum type is the same as 'real' if not pre-defined */
#ifndef sum_type
#define sum_type real
#endif

int sum2pi_x() {

    int i, j, k;
    real x, y, z;
    real err;
    real acc;
    sum_type sum;
```

```
    real final = (real)OUTER * PI;  /* correct answer */

    printf("===  Sum2PI_X  ===\n");
    printf("sizeof(real)=%d\n",     sizeof(real));
    printf("sizeof(sum_type)=%d\n", sizeof(sum_type));

    sum = 0.0;
    for (i=0; i<OUTER; i++) {
        acc = 0.0;
        for (j=1; j<INNER; j++) {

            /* calculate 2^j */
            x = 1.0;
            for (k=0; k<j; k++) {
                x *= 2.0;               /* CANDIDATE #1 */
            }

            /* accumulatively calculate pi */
            y = (real)PI / x;           /* CANDIDATE #2 */
            acc += y;                   /* CANDIDATE #3 */

        }
        sum += acc;                     /* CANDIDATE #4 */
        /*printf("  SUM%03d: %.16e\n", i, sum);*/
    }

    /* final should be PI*OUTER */
    err = ABS((double)final-(double)sum)/ABS((double)final);

    /*printf("------------------------------\n");*/
    printf("  RESULT:  %.16e\n", sum);
    printf("  CORRECT: %.16e\n", final);
    printf("  ERROR:   %.16e\n", err);
    printf("  THRESH:  %.16e\n", EPS);
    if ((double)err < (double)EPS) {
        printf("SUM2PI_X - SUCCESSFUL!\n");
    } else {
        printf("SUM2PI_X - FAILED!!!\n");
    }
    return 0;
}

int main() {
    return sum2pi_x();
}
```

A.8 Source: Makefile

```
all: sum2pi_x sum2pi_x-float sum2pi_x-mixed

test: all
./sum2pi_x
./sum2pi_x-float
./sum2pi_x-mixed

sum2pi_x: sum2pi_x.c
gcc -g -O2 -o sum2pi_x sum2pi_x.c

sum2pi_x-float: sum2pi_x.c
gcc -g -O2 -Dreal=float -o sum2pi_x-float sum2pi_x.c

sum2pi_x-mixed: sum2pi_x.c
gcc -g -O2 -Dreal=float -Dsum_type=double -o sum2pi_x-mixed sum2pi_x.c

clean:
rm -f sum2pi_x sum2pi_x-float sum2pi_x-mixed
```

Bibliography

[1] ASC Sequoia Benchmark Codes. https://asc.llnl.gov/sequoia/benchmarks/. Accessed 21 September 2011.

[2] CRAFT: Configurable Runtime Analysis for Floating-point Tuning. http://sourceforge.net/projects/crafthpc/. Accessed 14 January 2014.

[3] Flap: A Matlab Package for Adjustable Precision Floating-Point Arithmetic. http://www.cs.umd.edu/~stewart/flap/flap.html. Accessed 10 April 2010.

[4] GNU Multiple Precision Arithmetic Library. http://www.gmplib.org/. Accessed 23 February 2010.

[5] MPFR Library. http://www.mpfr.org/. Accessed 23 February 2010.

[6] SPEC CPU2006 Benchmark. http://www.spec.org/cpu2006/. Accessed 23 February 2010.

[7] Aberth, Oliver. A Precise Numerical Analysis Program. *Communications of the ACM*, 17(9):509–513, September 1974. ISSN 00010782. doi:10.1145/361147.361107.

[8] AMD. AMD64 Manual Volume 4: 128-Bit and 256-Bit SSE. 2011.

[9] Andrade, Marcus Vinicius Alvim, João Luiz Dihl Comba, and Jorge Stolfi. Affine Arithmetic. In *INTERVAL'94*, pages 1–10. St. Petersburg, Russia, 1994.

[10] Anzt, Hartwig, Piotr Luszczek, Jack Dongarra, and Vincent Heuveline. GPU-Accelerated Asynchronous Error Correction for Mixed Precision Iterative Refinement. In *Euro-Par 2012 Parallel Processing*, pages 908–919. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-32820-6_89.

[11] Anzt, Hartwig, Björn Rocker, 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. ISSN 18652034. doi:10.1007/s00450-010-0124-2.

[12] Arya, Sunil, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[13] Baboulin, Marc, 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*, 180:2526–2533, 2009.

[14] Bailey, D. H., et al. THE NAS PARALLEL BENCHMARKS. *International Journal of Supercomputer Applications*, 5(3):63–73, September 1991. doi:10.1177/109434209100500306.

[15] Bailey, David H. Multiprecision Translation and Execution of Fortran Programs. *ACM Transactions on Mathematical Software*, 19(3):288–319, 1993.

[16] Bailey, David H., Yozo Hida, Xiaoye S. Li, and Brandon Thompson. ARPREC: An Arbitrary Precision Computation Package. Technical report, 2002.

[17] Bao, Tao and Xiangyu Zhang. On-the-fly Detection of Instability Problems in Floating-Point Program Execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '13*, pages 817–832. ACM Press, New York, New York, USA, October 2013. ISBN 9781450323741. doi:10.1145/2509136.2509526.

[18] Beazley, D.M. SWIG Users Manual. Technical report, University of Utah Technical Report UUCS-98-012 (1998), 1998.

[19] Benz, Florian, Sebastian Hack, and Andreas Hildebrandt. A Dynamic Program Analysis to find Floating-Point Accuracy Problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2012. ISBN 9781450312059.

[20] Boisvert, Ronald F., Roldan Pozo, Karin Remington, Richard Barrett, and Jack J. Dongarra. The Matrix Market: a web resource for test matrix collections. In Ronald F. Boisvert, editor, *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137. Chapman & Hall, London, 1997.

[21] Brown, Ashley W, Paul H J Kelly, and Wayne Luk. Profiling floating point value ranges for reconfigurable implementation. In *Workshop on Reconfigurable Computing, HiPEAC 2007*. 2007.

[22] Brown, Ashley W, Paul 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. 2008.

[23] Buck, Bryan and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *Journal of Supercomputing Applications and High Performance Computing*, 14:317–329, 2000.

[24] Buttari, Alfredo, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Exploiting Mixed Precision Floating Point Hardware in Scientific Computation. Technical report, 2007.

[25] Buttari, Alfredo, 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. *ACM Transactions on Mathematical Software*, 34(4):1–22, 2008.

[26] Carlone, Ralph V. PATRIOT MISSILE DEFENSE: Software Problem Led to System Failure at Dhahran, Saudi Arabia (GAO/IMTEC-92-26). Technical report, U.S. Government Accountability Office, 1992.

[27] Carr, John W. Error Analysis in Floating Point Arithmetic. *Communications of the ACM*, 2(5):10–16, May 1959. ISSN 00010782. doi:10.1145/368325.368329.

[28] Clark, M. A., 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), 2010.

[29] Demmel, James W., Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A SUPERNODAL APPROACH TO SPARSE PARTIAL PIVOTING. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999. ISSN 08954798. doi:10.1137/S0895479895291765.

[30] Dongarra, Jack, et al. The International Exascale Software Project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011. ISSN 10943420. doi:10.1177/1094342010391989.

[31] Fang, Claire Fang, Rob A. Rutenbar, Markus Püschel, and Tsuhan Chen. Toward Efficient Static Analysis of Finite-Precision Effects in DSP Applications via Affine Arithmetic Modeling. In *Proceedings of the 40th Conference on Design Automation (DAC 2003)*, pages 496–501. ACM Press, 2003. ISBN 1581136889. doi:10.1145/775832.775960.

[32] Furuichi, Mikito, 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. ISSN 00219991. doi:10.1016/j.jcp.2011.09.007.

[33] Göddeke, Dominik 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. ISSN 10459219. doi:10.1109/TPDS.2010.61.

[34] Göddeke, Dominik, 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. ISSN 1744-5760. doi:10.1080/17445760601122076.

[35] Goldberg, David. What Every Computer Scientist Should Know About Floating Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[36] Goubault, Eric. Static Analyses of the Precision of Floating-Point Operations. *Static Analysis*, pages 234–259, 2001. doi:DOI-10.1007/3-540-47764-0_14.

[37] Goubault, Eric, Matthieu Martel, and Sylvie Putot. Asserting the Precision of Floating-Point Computations: a Simple Abstract Interpreter. *Programming Languages and Systems*, pages 287–306, 2002. doi:10.1007/3-540-45927-8_15.

[38] Hao, Xuejun and Amitabh Varshney. Variable-Precision Rendering. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics (SI3D'01)*, pages 149–158. ACM Press, 2001. ISBN 1581132921. doi:10.1145/364338.364384.

[39] He, Yun and Chris H.Q. Ding. Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications. *The Journal of Supercomputing*, 18(3):259–277, 2001. ISSN 09208542. doi:10.1023/A:1008153532043.

[40] Higham, Nicholas J. *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM Philadelphia, 2002.

[41] Hogg, J. D. 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. ISSN 00983500. doi:10.1145/1731022.1731027.

[42] IEEE. *IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008)*. IEEE, New York, August 2008.

[43] Jenkins, John, Eric R. Schendel, Sriram Lakshminarasimhan, David A. Boyuka II, Terry Rogers, Stephane Ethier, Robert Ross, Scott Klasky, and Nagiza F. Samatova. Byte-precision Level of Detail Processing for Variable Precision Analytics. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, SC '12. IEEE Computer Society Press, 2012. ISBN 978-1-4673-0804-5.

[44] Jézéquel, Fabienne and Jean-Marie Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933–955, June 2008. ISSN 00104655. doi:10.1016/j.cpc.2008.02.003.

[45] Kahan, William. The Improbability of PROBABILISTIC ERROR ANALYSES for Numerical Computations. Technical Report July 1995, University of California, Berkeley, 1996.

[46] Kaneko, Toyohisa and Bede Liu. On Local Roundoff Errors in Floating-Point Arithmetic. *Journal of the ACM*, 20(3):391–398, 1973. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321765.321771.

[47] Krämer, Walter. A priori Worst Case Error Bounds for Floating-Point Computations. *IEEE transactions on computers*, 47(7):750–756, 1998.

[48] Krämer, Walter and Armin Bantle. Automatic Forward Error Analysis for Floating Point Algorithms. *Reliable Computing*, 7(4):321–340, August 2001. ISSN 1385-3139. doi:10.1023/A:1011463324243.

[49] Laakso, Timo I. and Leland B. Jackson. Bounds for Floating-Point Roundoff Noise. *IEEE Transactions on Circuits and Systems*, 41(6):424–426, 1994.

[50] Larus, James R 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. ISBN 0897916972. ISSN 03621340. doi:10.1145/207110.207163.

[51] Lattner, Chris 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.

[52] Laurenzano, Michael A., Mustafa M. Tikir, Laura Carrington, and Allan Snavely. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 175–183. Performance Modeling and CHaracterization Laboratory, Ieee, 2010. ISBN 9781424460236. doi: 10.1109/ISPASS.2010.5452024.

[53] Le Grand, Scott, Andreas W. Götz, and Ross C. Walker. SPFP: Speed without compromise–A mixed precision model for GPU accelerated molecular dynamics simulations. *Computer Physics Communications*, 184(2):374–380, February 2012. ISSN 00104655. doi:10.1016/j.cpc.2012.09.022.

[54] Li, Xiaoye S., et al. Design, Implementation and Testing of Extended and Mixed Precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, June 2002. ISSN 00983500. doi:10.1145/567806.567808.

[55] Liu, Bede and Toyohisa Kaneko. Error Analysis of Digital Filters Realized with Floating-point Arithmetic. *Proceedings of the IEEE*, 57(10):1735–1747, 1969. ISSN 00189219. doi:10.1109/PROC.1969.7388.

[56] Luk, Chi-Keung, 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200. ACM, New York, NY, USA, 2005. ISBN 1-59593-056-6. doi:http://doi.acm.org/10.1145/1065010.1065034.

[57] Martel, Matthieu. Propagation of Roundoff Errors in Finite Precision Computations: A Semantics Approach. *Programming Languages and Systems*, pages 159–186, 2002. doi:10.1007/3-540-45927-8_14.

[58] Martel, Matthieu. Semantics-Based Transformation of Arithmetic Expressions. *Static Analysis*, pages 298–314, 2007. doi:10.1007/978-3-540-74061-2_19.

[59] Martel, Matthieu. Program Transformation for Numerical Precision. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*, pages 101–110. ACM Press, New York, NY, USA, January 2009. ISBN 978-1-60558-327-3. doi:http://doi.acm.org/10.1145/1480945.1480960.

[60] Meredith, Jeremy S., Gonzalo Alvarez, Thomas A. Maier, Thomas C. Schulthess, and Jeffrey S. Vetter. Accuracy and performance of graphics processors: A Quantum Monte Carlo application case study. *Parallel Computing*, 35(3):151–163, March 2009. ISSN 01678191. doi:10.1016/j.parco.2008.12.004.

[61] Nelder, J. A. and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965. ISSN 0010-4620. doi:10.1093/comjnl/7.4.308.

[62] Nethercote, Nicholas. *Dynamic Binary Analysis and Instrumentation*. Ph.D. thesis, University of Cambridge, November 2004.

[63] Nethercote, Nicholas and Julian Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, volume 42, pages 89–100. PLDI, ACM, 2007.

[64] Quinlan, Dan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000. ISSN 0129-6264. doi:10.1142/S0129626400000214.

[65] Richman, Paul L. Automatic Error Analysis for Determining Precision. *Communications of the ACM*, 15(9):813–820, September 1972. ISSN 00010782. doi: 10.1145/361573.361581.

[66] Rinard, Martin. Probabilistic Accuracy Bounds for Fault-Tolerant Computations that Discard Tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS'06)*, ICS '06, pages 324–334. ACM Press, 2006. ISBN 1595932828. doi:10.1145/1183401.1183447.

[67] Rubio-González, Cindy, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning Assistant for Floating-Point Precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on (SC'13)*, pages 1–12. ACM Press, New York, New York, USA, November 2013. ISBN 9781450323789. doi: 10.1145/2503210.2503296.

[68] Schreppers, Walter 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. ISSN 00983500. doi: 10.1145/1322436.1322441.

[69] Strzodka, Robert and Dominik Göddeke. Mixed Precision Methods for Convergent Iterative Schemes. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, pages 23–24. 2006.

[70] Strzodka, Robert and Dominik Göddeke. Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. In *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 259–270. 2006. doi:10.1.1.71.1466.

[71] Tabatabaee, Vahid, 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)*, SC '05. IEEE Computer Society, 2005. ISBN 1595930612. doi:10.1109/SC.2005.52.

[72] Tiwari, Ananta, Vahid Tabatabaee, and Jeffrey K. Hollingsworth. Tuning parallel applications in parallel. *Parallel Computing*, 35(8-9):475–492, 2009. ISSN 01678191. doi:10.1016/j.parco.2009.07.001.

[73] Wilkinson, J. H. Error Analysis of Floating-point Computation. *Numerische Mathematik*, 2(1):319–340, December 1960. ISSN 0029-599X. doi:10.1007/BF01386233.

[74] Wilkinson, J. H. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Inc., 1964.

[75] Wilkinson, J. H. Error analysis revisited. *IMA Bulletin*, 22(11/12):192–200, 1986.