

A Dynamic Analysis for Tuning Floating-point Precision

Cuong Nguyen



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/Eecs-2015-6

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/Eecs-2015-6.html>

February 13, 2015

Copyright © 2015, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A Dynamic Analysis for Tuning Floating-Point Precision

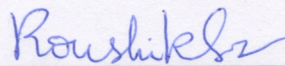
by Cuong Nguyen

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements for the
degree of **Master of Science, Plan II**.

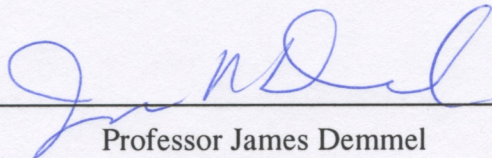
Approval for the Report and Comprehensive Examination:

Committee:



Professor Koushik Sen
Research Advisor

(02/10/2015)



Professor James Demmel
Second Reader

A Dynamic Analysis for Tuning Floating-Point Precision

Cuong Nguyen

Abstract

Floating-point numbers are widely used to approximate real number arithmetic in applications from domains such as scientific computing, graphics, and finance. However, squeezing one or more real numbers into a finite number of bits requires an approximate representation. As a consequence, the result of a floating-point computation typically contains numerical errors. To minimize the chance of problems, developers without an extensive background in numerical analysis are likely to use the highest available precision throughout the whole program. While more robust, this can increase the program execution time, memory traffic, and energy consumption.

In this work, we propose a dynamic analysis technique to assist developers in tuning precisions of floating-point programs with the objective of decreasing the program execution time. Our technique includes two phases. The first phase is a white-box analysis to reduce the precision search space, entitled Blame Analysis. The second phase is a black-box Delta Debugging based search algorithm that finds a type configuration that minimizes the set of variables that are required to be in higher precision, such that the new program, when transformed according to type configuration, will produce an accurate enough output and runs at least as fast as the original program. Our preliminary evaluation using ten programs from the GSL library and NAS benchmarks shows that our algorithm can infer type configurations that result in 4.56% execution time speed up on average, and up to 40% execution time speed up. In addition, Blame Analysis helps to speed up the analysis time of the second phase 9x on average.

Contents

1	Introduction	3
2	Overview	6
2.1	A Motivating Example	6
2.2	Overview of Our Approach	6
3	Reducing the Precision Search Space using Blame Analysis	10
3.1	Blame by Example	10
3.2	Shadow Execution	12
3.3	Online Blame Analysis	14
3.4	Handling Branch Divergence	16
3.5	Heuristic and Optimization	17
4	Minimizing Precision Usage using Delta-Debugging Based Search	19
4.1	Creating Search Space	19
4.2	Search Algorithm	20
4.3	Validating Configuration	22
5	Experimental Evaluation	23
5.1	Experiment Setup	23
5.2	Experiment Results	24
5.3	Discussion	27
6	Related Work	30
7	Conclusion	32

1 Introduction

Floating-point numbers are widely used in applications from domains such as scientific computing, graphics, and finance [19]. However, programs that use floating-point numbers are usually hard to reason about because of a number of numerical errors the program might have. Indeed, floating-point arithmetic most likely contain *roundoff* errors because floating-point numbers are inherently approximations of real values, The following two examples illustrate how computing with floating-point numbers might be frustrated.

Example 1.1. [19] Consider depositing \$100 every day into a bank account that earns an annual interest rate of 6%, compounded daily. If $n = 365$ and $i = .06$, the amount of money accumulated at the end of one year is $100 \frac{(1+i/n)^n - 1}{i/n}$. Computing this result using `float` on an IEEE standard machine yields the result \$37615.45 compared to the exact answer of \$37614.05, a discrepancy of \$1.40.

Example 1.2. [26] Consider the recurrence $u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}}$ where $u_0 = 2$ and $u_1 = -4$. Computing u_{30} with `float`, `double`, or `long double` on an IEEE standard machine yields the result 99.99999999998948 which is wrong in every single digit compared with the exact value 6.0067860930312057585. The following table shows that the computed value diverges from the exact value when n gets bigger.

n	Computed Value	Exact Value
3	18.5	18.5
4	9.378378378378379	9.3783783783783784
...		
30	99.99999999998948	6.0067860930312057585
31	99.99999999998943	6.0056486887714202679

To minimize the chance of problems, developers without an extensive background in numerical analysis are likely to use the highest precision available throughout the whole program. While more robust, this approach can increase program execution time, memory usage or energy assumption. For example, consider the program written by D. Bailey [4], which we will discuss in greater detail in Section 2.1, as an example. This program computes the arc length of an irregular function, written in `Fortran`. An implementation using `double` precision computes a result whose error is about 2×10^{-13} , compared to a second more accurate implementation using `double double` precision, but is about 20x slower¹. On the other hand, if double double precision is used only for one carefully chosen variable in the program, then the result computed is as accurate as if double double precision had been used throughout the whole computation while being almost as fast as the all-double implementation. Aside from the Bailey experiment [4], many other efforts [3, 11, 23, 25] have shown that programs implemented in mixed precision can sometimes compute a result of the same accuracy and be faster than when using solely the highest precision arithmetic.

¹ The author performed the experiment on an Intel-based Macintosh system, using the `gfortran` compiler and the QD package [20] to implement the double double type (approximately 31 digits).

In this work, we propose a dynamic analysis to assist developers in finding parts of the program that can be performed in lower precision. In addition, the program, when transformed according to our suggestion, will still produce an accurate enough answer and run at least as fast as the original program. Specifically, our analysis produces a type configuration that maps each floating-point variable to be tuned to its desired floating-point type. Our analysis consists of two phases. In the first phase, we employ a novel white-box dynamic analysis to reduce the size of the type configuration search space. We call this analysis Blame Analysis. Essentially, Blame Analysis computes the precision requirements of every instruction in the execution trace. The precision requirement specifies which precisions are required for the operand variables and the variable that holds the end result, so that the end result can be *accurate enough* with respect to a given error threshold. We will visit our definition of being accurate enough in Section 2. Essentially, it is accurate enough compared to the result as if the highest precision was used throughout the whole program. Blame Analysis then propagates and accumulates the precision requirements from the instruction that computes the end result to the beginning of the execution trace, and outputs all variables that are required to be computed in higher precision. We collect this set of variables as the search space for the second phase, while changing the type of other variables to lower precision. The analysis time is bounded by the number of instructions in the execution trace. When heuristics are applied, we observe that the overhead of the analysis is about 50 times the execution time of the program under analysis.

In the second phase of the analysis, we employ a black-box Delta Debugging [36] based search on the set of double precision variables inferred using Blame Analysis. The second analysis outputs a *local minimal* configuration that, when applied to the original program, results in a program that is accurate enough with respect to the given error threshold, and at least as fast as the original program, while using less precision. The analysis time of the second phase is bounded by the number of floating-point variables under analysis, and exhibits an $O(\log(n))$ best-case complexity and an $O(n^2)$ worst-case complexity, where n is the number of variables to be tuned. We note that both phases of our analysis require a representative set of program inputs. We do not guarantee that the analysis results will remain correct for other inputs.

We evaluate our analysis on a set of 10 programs, including 8 programs from the GSL library [13] and 2 programs from the NAS parallel benchmarks [31]. Our experiment shows encouraging results: the analysis was able to infer type configurations that translate to as high as 40% program execution speed up. In addition, Blame Analysis helps to speed up the analysis time of the second phase 9 times on average.

Our main contributions are as follows:

1. We introduce a novel dynamic analysis to assist developers in tuning precision of floating-point programs. The programs, when tuned according to our suggestion, are guaranteed to be accurate enough within the given error threshold and as least as fast as the original program, with respect to a set of representative inputs.
2. Our analysis consists of two novel algorithms: a black-box Delta Debugging based search algorithm to tune floating-point precisions, and a white-box analysis to reduce the search space for the previous algorithm, entitled Blame Analysis.

3. We implement our analysis and demonstrate its effectiveness on 10 numeric programs. Our implementation and benchmarks are publicly available at <https://github.com/corvette-berkeley>.

The rest of this paper is structured as follows. In Section 2, we present a motivating example and give an overview of our analysis. We then provide a detailed presentation of two algorithms used in our approach in Section 3 and Section 4 respectively. We present our experimental evaluation in Section 5, which is followed by related work in Section 6. We finally conclude in Section 7.

2 Overview

2.1 A Motivating Example

We begin with an example to motivate how changing the floating-point precision can affect the program execution time. Consider the program introduced by Bailey [4], which estimates the arc length of the following function over the interval $(0, \pi)$.

$$g(x) = x + \sum_{1 \leq k \leq 5} 2^{-k} \sin(2^k x)$$

The corresponding program sums $\sqrt{h^2 + (g(x_k + h) - g(x_k))^2}$ for $x_k \in [0, \pi)$ divided into n subintervals, where $n = 1,000,000$, so that $h = \frac{\pi}{n}$ and $x_k = kh$. An example implementation in C using `long double` precision is shown in Figure 2a. When compiled with `gcc` with optimization option `O2` and run on an Intel x86 system², this program produces the answer 5.795776322412856 (stored in variable `s1` on line 28). If the program uses `double` precision instead, the resulting value would be 5.79577632241**311**, which is only correct up to 11 or 12 digits after the decimal point (compared to the result produced by the `long double` precision program).

Figure 2b shows an optimized implementation written in C by an expert numerical analyst. Compared to the original program, the optimized program uses `long double` precision only for the variable `s1` on line 17. Six other variables have been lowered to `double` precision and one variable to `float`. The program produces the answer 5.795776322412856, which agrees with the answer when computed using all `long double` to 16 digits, and runs 10% faster than the `long double` precision version. When replacing `long double` with even higher `double double` precision from the QD package [20], the same transformation still produces the answer that agrees with the answer when computed using `call double double` to 16 digits, [4] and the performance improvements can be as high as 20×. We note that `double double` arithmetic is considerably slower than `long double` arithmetic because it is done in software, and each operation requires a sequence of roughly 20 instructions.

2.2 Overview of Our Approach

In this paper, we develop a dynamic analysis approach to assist developers in identifying parts of the program that can be performed in lower precision, with as little programming effort or mathematical analysis on their part as possible. Figure 1 depicts the overview of our approach. Inputs to our approach are the floating-point program under analysis, represented in the format of LLVM bitcode [24], a set of representative inputs, and an analysis input parameter file which specifies the program point of interest and the error threshold(s). LLVM bitcode representation of a program can be obtained from its source code using the CLANG compiler [24]. The source languages that we support include C, C++ and Fortran. Output of our approach is a type configuration that maps each floating-point

² All floating-point types used in this paper conform to the IEEE 754-2008 standard as implemented on various x86 architectures. In particular, `float` is implemented as the 32-bit precision type with 23 significant bits, `double` is implemented as the 64-bit precision type with 52 significant bits and `long double` is implemented as the 80-bit extended precision type with 64 significant bits. `double double` is implemented using pairs of double precision values.

variable to be tuned to its desired precision. This configuration, when applied to the original program, will result in another program that produces an answer that is accurate enough compared to the answer produced by the original program, within a given error threshold specified by the user, which may for example use a problem-dependent metric to compare the (trusted) computed answer from the all-double-precision version to the answer from the version with lower precision. We will refer to such an answer as an *accurate enough* answer. In addition, the new program is also guaranteed to run at least as fast as the original program.

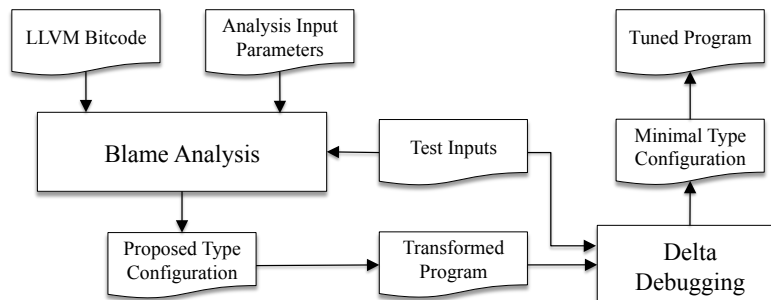


Fig. 1: Overview of Our Approach

Our approach comprises two core algorithms. The first algorithm divides the floating point variables into two categories: (1) those which may safely be converted to single precision, and (2) those which may or may not have to be in double precision; category (2) is further analyzed by the second algorithm. The second algorithm is a Delta-Debugging based search that infers a minimal set of variables that are required to be in double precision, such that the new program when changed accordingly will run at least as fast as the original program, while still being accurate enough.

The main challenge of our analysis is to devise an efficient search strategy through the type configuration space, to find a type configuration that (1) uses less precision, (2) produces an accurate enough answer, and (3) results in a program with better performance than the original program. We note here that our success metric is not to minimize the number of double precision variables alone, as doing that might increase the program execution time due to the overhead of type conversion between single and double precision. Instead, we minimize the number of double precision variables, while making sure that the transformed program gets better performance. We achieve this in two ways. First, we employ Blame Analysis to effectively reduce the type configuration search space. The overhead of Blame Analysis is about 50 times the program execution time. This number is comparable with the overhead of widely-used dynamic analysis tools such as VALGRIND or JALANGI [33]. The second phase of our approach looks for the *minimal* type configuration that satisfies the three conditions mentioned earlier. This second phase exhibits an $O(n^2)$ worst-case complexity, which requires $O(n^2)$ program transformations and re-executions, where n is the number of variables under analysis.

We consider two definitions of minimum in this work.

Finding a Global Minimum. Finding a global minimal set of floating-point variables

```

1 long double fun( long double x ) {
2   int k, n = 5;
3   long double t1;
4   long double d1 = 1.0L;
5
6   t1 = x;
7   for( k = 1; k <= n; k++ ) {
8     d1 = 2.0 * d1;
9     t1 = t1 + sin (d1 * x) / d1;
10  }
11  return t1;
12 }
13
14 int main( int argc, char **argv) {
15  int i, n = 1000000;
16  long double h, t1, t2, dppi;
17  long double s1;
18
19  t1 = -1.0;
20  dppi = acos(t1);
21  1 = 0.0;
22  t1 = 0.0;
23  h = dppi / n;
24
25  for( i = 1; i <= n; i++ ) {
26    t2 = fun (i * h);
27    s1 = s1 + sqrt (h*h +
28                  (t2 - t1)*(t2 - t1));
29    t1 = t2;
30  }
31  // final answer is stored
32  // in variable s1
33  return 0;
34 }

```

(a) Original program

```

1 double fun( double x ) {
2   int k, n = 5;
3   double t1;
4   float d1 = 1.0f;
5
6   t1 = x;
7   for( k = 1; k <= n; k++ ) {
8     d1 = 2.0 * d1;
9     t1 = t1 + sin (d1 * x) / d1;
10  }
11  return t1;
12 }
13
14 int main( int argc, char **argv) {
15  int i, n = 1000000;
16  double h, t1, t2, dppi;
17  long double s1;
18
19  t1 = -1.0;
20  dppi = acos(t1);
21  s1 = 0.0;
22  t1 = 0.0;
23  h = dppi / n;
24
25  for( i = 1; i <= n; i++ ) {
26    t2 = fun (i * h);
27    s1 = s1 + sqrt (h*h +
28                  (t2 - t1)*(t2 - t1));
29    t1 = t2;
30  }
31  // final answer is stored
32  // in variable s1
33  return 0;
34 }

```

(b) Tuned program

Fig. 2: Two implementations of the `arclength` program using different type configurations. The programs differ on the precision of all floating-point variables except for variable `s1`.

that are required to be in double precision to satisfy the three conditions mentioned earlier may require evaluation of an exponential number of type configurations. To be precise, we may be required to evaluate as many as 2^n configurations, where n is the total number

of variables under analysis. This naïve approach evaluates all possible type configurations by changing the type of one variable at a time. The shortcoming of this approach is that it does not scale to programs that have hundreds to thousands of variables, as typically observed in real world applications.

Finding a Local 1-Minimum. We denote Δ to be the set of floating-point variables that are required to be in double precision, so that the program satisfies the three conditions mentioned earlier. This means that all variables in Δ are in double precision, while other elements are in single precision. We say Δ is a local 1-minimum if when we remove any variable from Δ , which means that element removed can be in a lower precision, while all other elements remained in Δ are in the higher precision, the corresponding program will either (1) compute an end result that is inaccurate or (2) run slower than the original program. We note that a local 1-minimum solution might not be as good as a global minimum solution, which means that a local 1-minimum solution might result in a program that runs slower than the program transformed according to the global minimum solution. However, a local 1-minimum solution can be effectively found using a Delta Debugging based search algorithm, as we will show in Section 4. We therefore target local 1-minimum in this work.

```

1  /* computing factor*a^n */
2  double mpow(double a, double factor, int n) {
3    double res = factor;
4    int i;
5    for (i = 0; i < n; i++) {
6      res = res * a;
7    }
8    return res;
9  }
10
11 int main() {
12  double a = 1.84089642; /* stored as 1.8408964199999... in double precision */
13  double res, t1, t2, t3, t4;
14  double r1, r2, r3;
15
16  t1 = 4*a;
17  t2 = mpow(a, 6, 2);
18  t3 = mpow(a, 4, 3);
19  t4 = mpow(a, 1, 4);
20
21  /* res = a^4 - 4*a^3 + 6*a^2 - 4*a + 1
22    * = (a-1)^4 */
23  r1 = t4 - t3;
24  r2 = r1 + t2;
25  r3 = r2 - t1;
26  res = r3 + 1;
27
28  printf("res = %.10f\n", res);
29  return 0;
30 }

```

Fig. 3: Blame by Example

3 Reducing the Precision Search Space using Blame Analysis

We discuss the realization of Blame Analysis as a practical and effective tool for dynamically reducing the precision search space. We will start with an example to illustrate how the algorithm works.

3.1 Blame by Example

Consider the example program in fig. 3 which produces a result on line 24. When written using only double precision variables the result is (`res = 0.5000000113`). When executed solely in single precision, the result will be (`res = 0.4999980927`). Assuming that we are only interested in 8 significant digits of the result, then the required result would be (`res = 0.50000001xy`), where x and y can be any decimal digits. For each instruction in the program, Blame Analysis determines the precision that the corresponding operands are required to carry in order for its result to be accurate to a given precision. In this example,

Tab. 1: The `r3 = r2 - t1` statement executed when operands have different precisions. The column `Prec` shows the precisions used for the operands (`f1` corresponds to float, `db` to double, and `db8` is a value accurate up to 8 digits). Columns `r2` and `t1` show the values for the operands in the corresponding precisions. Column `r3` shows the result for the subtraction. Finally, column `S?` shows whether the result satisfies the given precision requirement.

Prec	r2	t1	r3	S?
(f1,f1)	6.8635854721	7.3635854721	-0.5000000000	No
(f1,db ₈)	6.8635854721	7.3635856000	-0.5000001279	No
(f1,db)	6.8635854721	7.3635856800	-0.5000002079	No
(db ₈ ,f1)	6.8635856000	7.3635854721	-0.4999998721	No
(db ₈ ,db ₈)	6.8635856000	7.3635856000	-0.5000000000	No
...
(db,db)	6.8635856913	7.3635856800	-0.4999999887	Yes

we consider three precisions: `f1` (float), `db` (double) and `d8` (accurate up to 8 significant digits compared to the double precision value). More specifically, the value in precision `d8` should represent a value that agrees with the value obtained when double precision is used throughout the entire program in 8 significant digits. Formally, such a value can be obtained from the following procedure. Let v be the value obtained when double precision is used throughout the entire program, and v_8 is the value of v in precision `db8`. According to the IEEE 754-2008 standard, the binary representation of v has 52 mantissa bits. We first find the number of bits that corresponds to 8 significant decimal digits in these 52 mantissa bits. The number of bits can be computed as $\lg(10^8) = 26.57$ bits. We therefore keep the 27 significant bits in the 52 mantissa bits, and set other bits in the mantissa to 0 to obtain the value v_8 . Similarly, if we are interested in 4, 6 or 10 significant decimal digits, we can keep 13, 19 and 33 significant bits in the mantissa respectively, and set other bits to 0.

Consider the statement on line 23: `r3 = r2 - t1`. Since the double value of `r3` is `-0.4999999887`, this means that we require `r3` to be `-0.49999998` (i.e., the value with precision `db8` that can be computed using the procedure described earlier, printed up to 8 significant digits). In order to determine the precision requirement for the two operands (`r2` and `t1`), we perform the subtraction operation with operands in all considered precisions. Table 1 shows some of the precision combinations we use for the operands. We note here that (1) the precision of the operator is single precision if both operands are in single precision; otherwise the precision of the operator is double precision, and (2) we only show value to 11 significant digits because we are only interested in up to 8 significant digits. For example, `(f1,db8)` means that `r2` has *float* precision, and `t1` has `db8` precision. For this particular statement, all but one operand precision combinations fails. Only when we try `(db,db)`, do we obtain a result that satisfies the precision requirement for the result (see last row of table 1). Blame Analysis will record that the precision requirement for the operands in the statement on line 23 is `(db,db)`, when the result is required to have precision `db8`.

Statements that occur in loops are executed more than once, such as line 5: `res = res * a`. Assume we also require precision `db8` for the result of this operation. The first time

we encounter the statement, the analysis records the double values for the operands and the result (6.0000000000, 1.8408964199, 11.0453785199), which is a tuple of input value of `res`, input value of `a` and output value of `res`. The algorithm tries different precision combinations for the operands, and determines that precision $(f1, db_8)$ suffices. The second time the statement is encountered, the analysis records new double values (11.0453785199, 1.8408964199, 20.3333977737). After trying all precision combinations for the operands, it is determined that this time the precision required is (db, db_8) , which is different from the requirement set the first time the statement was examined. At this point, it is necessary to *merge* both of these precision requirements to obtain a unified requirement. In Blame Analysis, the merge operation over-approximates the precision requirements. In this example, merging $(f1, db_8)$ and (db, db_8) would result in the precision requirement (db, db_8) .

Finally, after computing the precision requirements for every instruction in the program, the analysis performs a backward pass starting from the target statement on line 24. The pass finds the program dependencies, and collects all variables that are determined to be in single precision. Concretely, if we require the final result computed on line 24 to be accurate to 8 digits db_8 , the backward pass finds that the statement on line 24 depends on statement on line 23, which depends on statements on lines 22 and 15, and so on. The analysis collects the variables that cannot be allocated in single precision based on the program dependencies. In this example, all except for the variable `factor` in function `mpow` are collected because `factor` is the only variable that can be single precision (it always stores integer constants which do not require double precision).

In the rest of this section, we formally describe Blame Analysis algorithm and its implementation. Figure 4 depicts the architecture of our implementation of Blame Analysis, which is built on top of the LLVM compiler infrastructure [24]. Inputs to the tool are the LLVM bitcode of the program under analysis, a set of test inputs and an analysis input parameters. The analysis input parameters include (i) the program point(s) of interest and (ii) the number of significant digits that are required to be accurate. The output of the tool is a proposed type configuration that maps each floating-point variable to its desired precision.

Our implementation of Blame Analysis consists of two main components: a shadow execution engine for performing single and double precision computation *side-by-side* with the concrete computation (Section 3.2), and an online blame analysis algorithm integrated inside the shadow execution runtime (Section 3.3). In Section 3.4, we discuss a method for handling *branch divergence* during shadow execution, which sometimes presents in our benchmark programs. Finally, we present some heuristics and optimizations in Section 3.5. In Section 3.5, we also discuss some alternative ways to implement Blame Analysis, and present their strengths and weaknesses.

3.2 Shadow Execution

We introduce a kernel language (Figure 5) for formal discussion. The language includes standard arithmetic and boolean operation instructions. It also includes an assignment statement which assigns a constant value to a variable. Other instructions include `if-goto` and native function call instructions such as `sin`, `cos` and `fabs`.

In our shadow execution engine, each concrete floating-point value in the program has

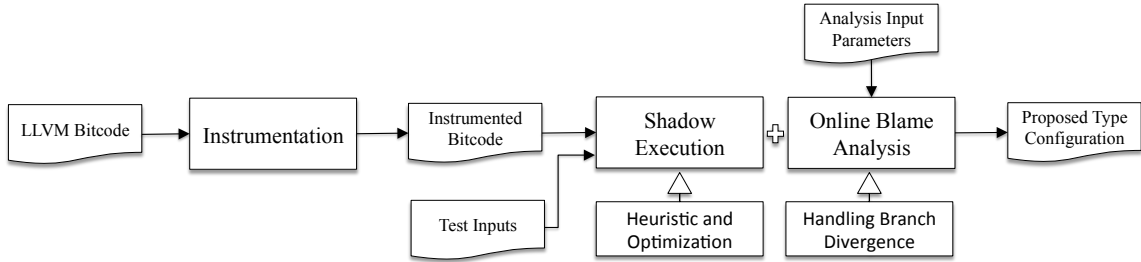


Fig. 4: Architecture of an Implementation of Blame Analysis

$$\begin{aligned}
 Pgm & ::= (L : Instr)* \\
 Instr & ::= x = y \ aop \ z \mid x = y \ bop \ z \mid \mathbf{if} \ x \ \mathbf{goto} \ L \mid \\
 & \quad x = \mathit{nativefun}(y) \mid x = c \\
 aop & ::= + \mid - \mid * \mid / \\
 bop & ::= = \mid \neq \mid < \mid \leq \\
 \mathit{nativefun} & ::= \mathbf{sin} \mid \mathbf{cos} \mid \mathbf{fabs} \\
 L \in Labels \quad x, y, z \in Vars \quad c \in Consts
 \end{aligned}$$

Fig. 5: Kernel Language

an associated *shadow value*. A shadow value associated with a concrete value carries two values corresponding to the concrete value when the program is computed entirely in *single* or *double* precision. We will represent a shadow value of a value v as $\{single : v_{single}, double : v_{double}\}$, where v_{single} and v_{double} are the values corresponding to v when the program is computed entirely in *single* or *double* precision.

In our implementation, the shadow execution is performed side-by-side with the concrete execution. Our implementation of shadow execution is based on instrumentation. We instrument *callbacks* for all instructions in the program. The shadow execution runtime interprets the callbacks following the same semantics of the corresponding instructions, however, it computes shadow values rather than concrete values.

Let A be the set of all memory addresses used by the program, S be the set of all shadow values associated with the concrete values computed by the program, and L be the set of labels of all instructions in the program. Shadow execution maintains two data-structures:

- a shadow memory M that maps a memory address to a shadow value, i.e. $M : A \rightarrow S$. If $M(a) = s$ for some memory address a , then it denotes that the value stored at address a has the associated shadow value s ,
- a label map LM that maps a memory address to an instruction label, i.e. $LM : A \rightarrow L$. If $LM(a) = l$ for some memory address a , then it denotes that the value stored at address a was last updated by the instruction labeled l .

As an example, Figure 6 shows how M and LM are updated when an add instruction $l : x = y + z$ is executed. In this example, x, y, z are variables and l is an instruction label. We also denote $\&x, \&y, \&z$ as the addresses of the variable x, y, z , respectively, in that state. In this example, the procedure `AddShadow` is the callback associated with the

Procedure AddShadow**Inputs**

$l : x = y + z$: instruction

Outputs

Updating the shadow memory M and the label map LM

Method

- 1 $\{\text{single: } y_{\text{single}}, \text{double: } y_{\text{double}}\} = M[\&y]$
- 2 $\{\text{single: } z_{\text{single}}, \text{double: } z_{\text{double}}\} = M[\&z]$
- 3 $M[\&x] = \{\text{single: } y_{\text{single}} + z_{\text{single}}, \text{double: } y_{\text{double}} + z_{\text{double}}\}$
- 4 $LM[\&x] = l$

Fig. 6: Shadow Execution of `fadd` Instruction

add instruction. Line 3 of the procedure re-interprets the semantic of an add instruction, but uses the shadow values of the left and right operands (obtained from line 1 and 2 of the same procedure) as operands. Line 3 performs the additions and returns the results in the same precision as the two operands. Line 4 of the procedure updates the label map LM to save the fact that x was last updated at the instruction labeled l .

Modeling C standard numeric library functions. We commonly encounter functions from *C standard numeric library* [1] such as `cos`, `sin` or `fabs` in our benchmark programs. Each of these functions has an associated single precision version, e.g. `sinf` is the single precision version of `sin`. Therefore, to compute the single precision results of these functions, the callbacks invoke the single precision versions of these functions using single precision arguments. Similarly, to compute the double precision results, the callbacks invoke the double precision versions of these functions using double precision arguments.

3.3 Online Blame Analysis

Our implementation of Blame Analysis receives the program point(s) of interest and the error threshold(s) as input. The error thresholds correspond to whether we want the results computed by the instructions at the program points of interest to be accurate to 4, 6, 8, or 10 significant digits. It outputs a type configuration that maps each floating-point variable to the desired precision, so that the results produced from the program points of interest are accurate within the error threshold. If more than one error threshold is given, it produces one type configuration for each error threshold.

We will need a set of precisions

$$\mathbf{P} = \{\mathbf{f1}, \mathbf{db}_4, \mathbf{db}_6, \mathbf{db}_8, \mathbf{db}_{10}, \mathbf{db}\}$$

which represents the precision requirement we will infer for each variable in the program. Precisions `f1` and `db` stands for single and double precision, respectively. Precisions `db4`, `db6`, `db8`, `db10` means to be accurate to 4, 6, 8 and 10 significant digits in double precision, respectively. Concretely, the values that correspond to these precisions are computed as follows. Assume that v is a value computed when double precision is used throughout the

entire the program, and v_i is the value of v in precision db_i , where i can be 4, 6, 8 or 10. We first compute the number of bits in the binary representation of v that corresponds to i significant decimal digits using the formula $\lceil \lg(10^i) \rceil$. We then keep $\lceil \lg(10^i) \rceil$ significant bits in the 52 mantissa bits of v , and set other bits in the mantissa to 0 to obtain the value v_i . Specifically, we keep 13, 19, 27 and 33 significant bits in the mantissa and set other bits to 0 to compute values of precisions db_4 , db_6 , db_8 , db_{10} respectively. We also define a total order of precisions as follows: $\text{fl} < \text{db}_4 < \text{db}_6 < \text{db}_8 < \text{db}_{10} < \text{db}$.

We also define a data structure

$$B : L \times P \rightarrow \mathcal{P}(L \times P)$$

which maps a pair of instruction label and precision to a set of pairs of instruction labels and precisions, where $\mathcal{P}(S)$ denotes the power set of S . If $B(l, p) = \{(l_1, p_1), (l_2, p_2)\}$, then it means that during an execution, if the instruction labelled l produces the value that is accurate to precision p , then instructions at label l_1 and l_2 must produce values that are accurate to precision p_1 and p_2 , respectively.

In shadow execution, we update data structure B on the execution of every instruction. We initialize B to the empty map at the beginning of the execution. We illustrate how B is updated for arithmetic operation and native function call instructions using a generic instruction of the form $l : x = f(y_1, y_2, \dots, y_n)$, where x, y_1, \dots, y_n are variables and f is an operator, which could be $+$, $-$, $*$, sin , cos , etc. We also denote $\&x, \&y_1, \dots, \&y_n$ as the addresses of x, y_2, \dots, y_n , respectively, in that state. We update $B(l, p)$ for each $p \in \mathbf{Prec}$ when this instruction is executed as follows. We use two functions, `BuildBlame` and `merge` \sqcup , to update $B(l, p)$.

The function `BuildBlame` receives an instruction and a precision requirement as input, and returns the precision requirements of the instructions that compute the operands. Figure 7 shows the pseudo-code of the function `BuildBlame`. The procedure first computes the correct result by obtaining the shadow value corresponding to the input instruction, and truncating the shadow value to precision p (line 1). The function `truncs(s, p)` computes the floating-point value corresponding to the precision p using the shadow value \mathbf{s} . Specifically, if the input precision is `fl`, `truncs` returns the single precision value from the shadow value. If the input precision is `db`, `truncs` returns the double precision value from the shadow value. If the input precision is `db4`, `db6`, `db8` or `db10`, `truncs` first obtains the double precision value from the shadow value, and truncates it according to the procedure described earlier. Line 2 obtains the shadow values corresponding to all operand variables. Then, the procedure find the minimal precisions p_1, p_2, \dots, p_n such that if we apply `f` to $\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_n$ truncated to precisions p_1, p_2, \dots, p_n , respectively, then the result truncated to precision p is equal to the correct result computed in line 1. The function `trunc(v, p)` returns the value v truncated to precision p . We note here that we require the two values at line 5 to be equal because we already truncate the two results to keep only the significant bits that correspond to precision p . Also, we order two tuples of precisions (p_1, p_2, \dots, p_n) and $(p'_1, p'_2, \dots, p'_n)$ in the following way. We say that $(p_1, p_2, \dots, p_n) < (p'_1, p'_2, \dots, p'_n)$ if and only if there exist i where $1 \leq i \leq n$ such that $p_i < p'_i$ and $\forall j, 1 \leq j < i : p_j = p'_j$.

The merge function \sqcup is defined as

$$\sqcup : \mathcal{P}(L \times P) \times \mathcal{P}(L \times P) \rightarrow \mathcal{P}(L \times P)$$

Procedure BuildBlame**Inputs**

$l : x = f(y_1, \dots, y_n)$: instruction
 p : precision requirement

Outputs

$\{(l_1, p_1), \dots, (l_n, p_n)\}$: precision requirement
of the instructions that computed the operands

Method

```

1 correct_res = truncs(M[&x],p)
2 (s1, ..., sn) = (M[&y1], ..., M[&yn])
3 find minimal precision p1, ..., pn such that the following holds:
4 (v1, ..., vn) = (truncs(s1, p1), ..., truncs(sn, pn))
5 trunc(f(v1, ..., vn), p) == correct_res
6 return {(LM(&y1), p1), ..., (LM(&yn), pn)}
```

Fig. 7: BuildBlame Procedure

If $(l, p_1), (l, p_2), \dots, (l, p_n)$ are all the pairs involving the label l present in LP_1 and LP_2 , then $(l, \max(p_1, p_2, \dots, p_n))$ is the only pair involving l present in $(LP_1 \sqcup LP_2)$.

Given the functions `BuildBlame` and \sqcup , we compute $B(l, p) \sqcup \text{BuildBlame}(l : x = f(y_1, \dots, y_n), p)$ and use the resulting set to update $B(l, p)$. In the special case of the instruction $l : x = C$, for every precision p , B always maps the instruction label l to an empty set, i.e. $B(l, p) = \emptyset$, because this instruction does not have any operand variables. Instructions that compute boolean values do not have any corresponding mappings in B , because they do not compute floating-point variables. However, they play an important role in preventing branch divergence, which we will discuss in Section 3.4.

At the end of the execution, we get a non-empty map B . Suppose we want to make sure the result computed by a given instruction labeled l_{out} is accurate to precision p . Then we want to know what would be the accuracy of the results computed by other instructions so that the accuracy of the result of the instruction labeled l_{out} is p . We compute this using the function $\text{Accuracy}(l_{out}, p, B)$ which returns a set of pair of instruction labels and precisions, such that if l', p' is present in $\text{Accuracy}(l_{out}, p, B)$ then the results of executing the instruction labeled l' must have a precision at least p' . $\text{Accuracy}(l_{out}, p, B)$ can be defined recursively as follows.

$$\text{Accuracy}(l, p, B) : \{(l, p)\} \sqcup \bigsqcup_{(l', p') \in B(l, p)} \text{Accuracy}(l', p', B)$$

3.4 Handling Branch Divergence

We must make sure that the configuration proposed by Blame Analysis, when applied to the original program, will not result in a new program that has different execution paths from the original program. To guarantee that, we exploit the concept of *discrete factor* introduced in [5]. Essentially, discrete factors are operations that have floating-point values as operands and produce discrete values as results. In our kernel language, boolean operation instructions are discrete factors. Similar to [5], we also want to guarantee that

none of the discrete factors induce different discrete values due to precision changes to the program.

For every boolean operation instruction $x = y \text{ bop } z$, we infer a pair of minimal precisions p_1 and p_2 , such that $\text{trunc}(y, p_1) \text{ bop } \text{trunc}(z, p_2) == x$. This means that the value of x does not change when we change the precisions of y and z to p_1 and p_2 , respectively. We then find the precision requirements for other instructions in the program such that values computed and stored in y and z are accurate to precisions p_1 and p_2 , respectively. Formula-wise, if l_1, \dots, l_n are labels of instructions that compute the values of variables involved in all boolean operation instructions, and p_1, \dots, p_n are the corresponding minimal precisions required for the values, respectively, such that the values computed by each boolean operation instruction are the same as the values computed when double precision is used throughout the program, we have:

$$\text{Accuracy}(l_{out}, p, B) : (l, p) \sqcup \bigsqcup_{(l', p') \in B(l_{out}, p)} \text{Accuracy}(l', p', B) \\ \sqcup \bigsqcup_{i=1}^n \text{Accuracy}(l_i, p_i, B)$$

Once we have computed $\text{Accuracy}(l_{out}, p, B)$, we know that if (l', p') is present in $\text{Accuracy}(l_{out}, p, B)$, then the instruction labeled l' must be executed with precision at least p' if we want (i) the program not to have different execution paths due to precision changes, and (ii) the result of executing instruction labeled l_{out} to have a precision p .

3.5 Heuristic and Optimization

Our implementation of online Blame Analysis is memory efficient because the size of the B mapping is bounded by the number of *static* instructions in the program. An alternative way to implement Blame Analysis is to implement *offline Blame Analysis*. Offline Blame Analysis first collects the complete execution trace, and builds the blame set of each executed instruction. As each instruction is examined only once, merging of operand precision is not required. Thus, when compared to online Blame Analysis, the offline Blame Analysis exhibits lower overhead per instruction, and sometimes produces better quality solutions. However, offline Blame Analysis is not scalable because the size of B explodes for large inputs and for long running programs. In particular, when running offline Blame Analysis on the `ep` program from the NAS benchmarks with input A [31], offline Blame Analysis returned an out of memory error code after using all 256GB memory of our system. On the other hand, online Blame Analysis was able to complete the analysis for the same program, using only 81MB memory.

Another alternative way to implement Blame Analysis is to implement Blame Analysis *without shadow execution*. Without shadow execution, we do not have the values corresponding to the precision `f1`, which are the values computed when single precision is used throughout the entire program. Instead, we define another precision `dbf1` to replace precision `f1`. Values in `dbf1` precision can be obtained by converting the values computed in double precision to single precision. The strength of this approach is that Blame Analysis runs faster without shadow execution. However, we note that the values in `dbf1` precision are often more accurate than the values in `f1` precision. Therefore, obtaining the values that are accurate to `dbf1` precision requires higher precisions throughout the program. As a consequence, Blame Analysis without shadow execution is less effective in finding variables

that can be allocated in single precision than Blame Analysis with shadow execution. For example, when running Blame Analysis on the `root` program from the GSL library [13], if we want the end result to be accurate to precision `db6`, Blame Analysis with shadow execution determines that 15 out of 16 variables can be in single precision, while Blame Analysis without shadow execution can only determine that 3 out of 16 variables can be in single precision. On the other hand, running Blame Analysis on long running programs, such that the `ep` and `cg` programs from NAS benchmarks [31], shows that shadow execution introduces only 30% slowdown on average. We note that the main bottleneck in Blame Analysis is the function `BuildBlame` (Section 3.3), which is required for both Blame Analysis with and without shadow execution.

In our implementation, we allow developers to specify what parts of the program they are interested in analyzing. For short running programs, such as functions within the GSL library [13], examining all instructions is feasible. Many long running scientific programs fortunately use iterative solvers. In this case, analyzing the last few iterations is likely to lead to a good solution, given that precision requirements are increased towards the end of the execution. This is the case in the NAS benchmarks we have analyzed. If no options are specified, Blame Analysis by default will be performed throughout the entire program execution.

4 Minimizing Precision Usage using Delta-Debugging Based Search

In Section 3, we proposed Blame Analysis, a method to infer a type configuration that, when applied to the original program, resulted in a program that (i) had the same execution paths as the original program, and (ii) produced an accurate enough answer within a given error threshold. In this section, we propose a black-box search algorithm that takes the type configuration produced by Blame Analysis as input, and produces a type configuration that (i) uses minimal precision, and (ii), when applied to the original program, will result in a new program that runs at least as fast as the original program. Our search algorithm is based on Delta Debugging search [36].

Figure 8 depicts the architecture of our Delta-Debugging based type configuration search algorithm, which is built on top of the LLVM compiler infrastructure [24]. Inputs to the tool are the LLVM bitcode of the program under analysis, a set of test inputs and a search configuration file. The search configuration file contains a map from each variable under tuning to a set of candidate precisions. Section 4.1 describes how we can infer this mapping given the program under analysis, and the type configuration produced by Blame Analysis. Output of the tool is the type configuration that, when applied to the original program, will result in a new program that runs at least as fast as the original program, while still producing an accurate enough answer.

Our tool consists of three main components: creating the search configuration file (Section 4.1), a feedback loop to create candidate type configurations based on delta-debugging search (Section 4.2) and a validator to validate the candidate configuration integrated within the feedback loop (Section 4.3).

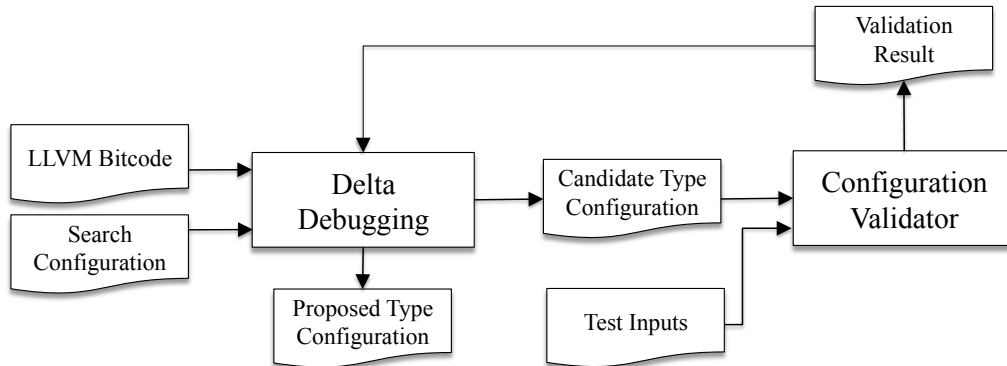


Fig. 8: Architecture of the Delta-Debugging Based Type Configuration Search

4.1 Creating Search Space

We start by creating the *search configuration file*, which consists of all variables whose precisions need to be tuned. The search file associates each floating-point variable with the set of floating-point types to be explored (e.g., `float`, `double`). The input to this process is the program under analysis in the format of LLVM bitcode. In the current implementation, the search file includes the local variables of all functions statically reached from `main`. We include only those global variables that are accessed by these functions. We include both

scalars and arrays. We note that for arrays, all entries of an array need to have the same precision. We currently do not support structure. The set of floating-point types to be explored for each variable is `{float, double}`.

We use the type configuration proposed by Blame Analysis to reduce the search space for Delta Debugging search in the following way. Firstly, if the variable x is determined to be of `float` precision in the type configuration proposed by Blame Analysis, we change the type of x in the (LLVM bytecode) program to `float` accordingly. In addition, we also remove the entry for x from the search configuration file.

4.2 Search Algorithm

We devise a modified version of the delta-debugging algorithm [36] to find a type assignment for floating-point variables so that the resulting program uses less precision while producing results that satisfy both accuracy and performance constraints.

Given a configuration search file, our goal is to find a type configuration that maps each variable to a type and that exhibits lower cost (performance improvement). Initially, each variable in the search file is associated with a set of types. Our algorithm iteratively refines each of these sets of types until it consists of only one type. Because our algorithm is based on the delta-debugging search algorithm, with heuristic pruning, it is efficient in practice. To balance between the searching cost and the quality of the selected configuration, we choose to search for a local 1-minimum configuration instead of a global minimum. As shown in Section 5, our algorithm finds type configurations for most of the analyzed programs (for the workloads under consideration), leading to performance speedup.

Figure 9 shows our lower cost configuration search algorithm, entitled LCCSEARCH. In this algorithm, a *change set* is a set of variables. The variables in the change set must have higher precision. The algorithm outputs a minimal change set, which consists of a set of variables that must be allocated in the higher precision (all other variables of interest can be in lower precision) so that the transformed program produces an accurate enough result and satisfies the performance goal. We note that for mixed-precision computations, the precision of the operation is the maximum precision of the two operands.

The algorithm starts by dividing the change set Δ into two subsets of equal or almost equal size Δ_1 and Δ_2 . It also creates the complement set of these subsets $\nabla_1 = \Delta \setminus \Delta_1$ and $\nabla_2 = \Delta \setminus \Delta_2$ (lines 4-5). For each of these subsets, the algorithm creates the corresponding program variant, checks whether the program variant produces an accurate enough result and is faster (4.3), and records the one that has the lowest cost (lines 6-11). The function `accurate(P, Δ)` transforms the program P according to Δ and returns a boolean value indicating whether the transformed program is accurate enough. Similarly, the function `cost(P, Δ)` transforms the program P according to Δ and returns the cost of the transformed program. We will discuss the realization of functions `accurate` and `cost` in greater detail in Section 4.3. If a change set with a lower cost exists, the algorithm recurses with that smaller change set (lines 15-16 and 24); otherwise it restarts the algorithm with a finer-grained partition (lines 21-24). In the special case where the granularity can no longer be increased, the algorithm returns the current Δ , which is a local minimum type configuration (lines 18-19).

Finally, we transform the program according to the change set Δ computed using LCC-

Procedure LCCSearch**Inputs**

P : target program
 Δ : change set

Outputs

A minimal change set

Algorithm

```

1  div = 2
2   $\Delta_{LC} = \Delta$ 
3   $div_{LC} = div$ 
4  for i in [1..div]:
5     $\Delta_i = \Delta[\frac{(i-1)|\Delta|}{div} \dots \frac{i|\Delta|}{div}]$ 
6     $\nabla_i = \Delta \setminus \Delta_i$ 
7    if accurate(P,  $\Delta_i$ ) and  $cost(P, \Delta_i) < cost(P, \Delta_{LC})$ :
8       $\Delta_{LC} = \Delta_i$ 
9       $div_{LC} = 2$ 
10   if accurate(P,  $\nabla_i$ ) and  $cost(P, \nabla_i) < cost(P, \Delta_{LC})$ :
11      $\Delta_{LC} = \nabla_i$ 
12      $div_{LC} = div - 1$ 
13 end for
14 if  $\Delta_{LC} \neq \Delta$ :
15    $\Delta = \Delta_{LC}$ 
16    $div = div_{LC}$ 
17 else:
18   if  $div > |\Delta|$ :
19     return  $\Delta$ 
20   else:
21      $div = 2 * div$ 
22   end if
23 end if
24 goto 3

```

Fig. 9: Lower Cost Configuration Search Algorithm

Search. We then measure the running time for the transformed program. If the running time of the transformed program is at least as fast as the running time of the original program, we return Δ as the proposed configuration. Otherwise, we return the configuration that corresponds to the original program as the output of the analysis. This final step guarantees that the configuration we produce will result in a program that runs at least as fast as the original program.

4.3 Validating Configuration

We validate a candidate configuration by first transforming the program according to the type configuration. We then run the transformed program and check for two criteria: correctness and performance.

We perform the transformation at the bitcode level, and by modifying the original program. First, we replace the `alloca` instructions to allocate the correct amount of memory for the variables whose type is to be changed. Second, we iterate through the use of each of these variables to identify instructions that may need to be transformed. We define a set of program transformation rules to be applied depending on the instruction to be transformed. For each transformed instruction, we iterate through its uses to continue to propagate the changes. Finally, in the case of arithmetic operation instructions, we run a final pass to make sure each operation is performed in the precision of its highest-precision operand.

To check for correctness criteria (function `accurate` in Figure 9), we compare the result produced by the transformed program against the *expected result*. The expected result is the value (or values) obtained by running the original program on a given set of inputs. We take into account the error threshold provided by the programmer when comparing results. To check for performance criteria (function `cost` in Figure 9), we measure the running times for the transformed programs. Note that the developer can specify the compiler to be used and the level of optimizations to be applied. In our experiments, we use the `clang` compiler with optimization level `O2`.

5 Experimental Evaluation

As noted earlier, we developed Blame Analysis as a method to speed up the analysis time of the LCCSearch algorithm (Section 4.2), by removing variables from the algorithm search space. This section first evaluates whether Blame Analysis helps to speed up LCCSearch analysis time. We then evaluate how Blame Analysis affects the type configurations proposed by LCCSearch overall. To facilitate our representation, we use the term *Blame LCCSearch* to refer to the combination approach of Blame Analysis and LCCSearch that is described in this paper. We use the term LCCSearch to refer to the approach that uses merely LCCSearch algorithm, without Blame Analysis. In our evaluation, we will compare Blame LCCSearch and LCCSearch approaches in term of analysis time, and the quality of the configurations proposed, measured by the speedup of the final configuration compared to the original code.

We present results for eight programs from the GSL library [13] and two programs from the NAS parallel benchmarks [31]. We use `clang` with no-optimization³ and a Python-wrapper [30] to build whole-program (or whole-library) LLVM bitcode, and `clang` with O2 optimization to compile programs for performance measurement. In particular, we use `clang 3.0` to compile programs for LCCSearch, as our implementation of LCCSearch does not support the newer version of LLVM currently. We use a newer version of `clang`, which is `clang 3.4` to compile programs for Blame Analysis, as well as for measuring performance measurement. We run our experiments on an Intel(R) Xeon(R) CPU E5-4640 0 @ 2.40GHz 8 core machine running Linux with 252GB RAM.

5.1 Experiment Setup

We expect that users of our approach have access to the source code of the program under analysis, and know which statements compute the end result of the program. For Blame Analysis, we specify the statements of interest in the format of lines of code, and the error threshold as 10^{-4} , 10^{-6} , 10^{-8} or 10^{-10} in the analysis input parameter file. These error thresholds correspond to the number of decimal significant digits we are interested in, which are 4, 6, 8 and 10 respectively.

Blame Analysis then transforms the set of lines of code to the set of LLVM instructions, and use that as the input to the analysis. Blame Analysis outputs the mapping of LLVM instructions to their desirable precisions. We use that to create a type configuration that maps each floating-point variable to its desired precision in the following way. We first map each LLVM instruction to the program line of code at the source code level. If that LLVM instruction is determined to be of precision `db4`, `db6`, `db8`, `db10` or `db` then we map every variables appeared in the corresponding line of code to double precision. Other variables in the program are mapped to single precision. We could interpret the results of Blame Analysis in a more fine-grained way, which would be a subject for future work.

For LCCSearch, we manually annotate the input program to specify the error threshold, log and check the results produced. LCCSearch explicitly outputs which variables need to be allocated in double precision, and which variables can be allocated in single precision.

³ Optimization sometimes removes variables, consequently causes the set of variables at the bitcode level to differ from the source code level.

We note that in an operation where one or more operands are in single precision and one or more operands are in double precision, the single precision operands are converted to double precision and the operation is performed in double precision. Type conversions in this case are handled automatically by the compiler.

Tab. 2: Average analysis time speedup of BLAME LCCSEARCH compared to LCCSEARCH

Program	Speedup	Program	Speedup
bessel	22.48×	sum	1.85×
gaussian	1.45×	fft	1.54×
roots	18.32×	blas	2.11×
polyroots	1.54×	ep	1.23×
rootnewt	38.42×	cg	0.99×

For the NAS parallel benchmarks (program `ep` and `cg`), we use the provided input Class A. For other programs, we generate at random 1000 floating-point inputs and classify them based on code coverage. We then pick one representative from each group to create test inputs, thus the goal is to maximize code coverage. We log and read the inputs in hexadecimal format to ensure that the inputs generated and the inputs used match at the bit level. This typically requires a few changes to the programs under analysis to read and use the inputs. In our experiments, we use four error thresholds that correspond to the number of digits of accuracy requirement, in particular, 4, 6, 8 and 10 digits. Finally, when comparing the performance, we run each program thousands of times (or millions depending on its size) to ensure that the program runs long enough to obtain a reliable timing measurement. The minimum total run time is 2 seconds, over all programs tested. We run the programs `ep` and `cg` only once because their running time is already long enough.

In addition, to measure the analysis time of Blame LCCSearch, for each error threshold, we sum up the analysis time of Blame Analysis and search time of LCCSearch. Furthermore, for `ep` and `cg` programs, we configure Blame Analysis to analyze only the last 10% of the executed instructions. For the rest of the programs, we configure Blame Analysis to analyze all executed instructions. For both approaches, if the configuration proposed results in a program that runs slower than the original program, we produce the original program as the output.

5.2 Experiment Results

Evaluation of Analysis Time. Figure 10 shows the analysis time of Blame LCCSearch (B+L) and LCCSearch (L) for ten benchmark programs. It also shows the analysis time for each error threshold (10^{-4} , 10^{-6} , 10^{-8} and 10^{-10})⁴. As noted earlier, the error threshold indicates the number of accurate digit required. For example, 10^{-4} roughly means that the result is required to be accurate to 4 digits.

Our result shows that Blame LCCSearch is faster than LCCSearch in 33 out of 39 (which is 85%) experiments. In general, we would expect that as variables are removed from the

⁴ The program `ep` only supports error threshold down to 10^{-8} , therefore we do not consider the error threshold 10^{-10} .

Tab. 3: Configurations found by Blame Analysis (B), BLAME LCCSEARCH (B+L), and LCCSEARCH alone (L). The column Initial gives the number of floating-point variables (double D, and float F) declared in the programs. For each selected error threshold, we show the type configuration found by each of the three analyses B, B+L, and L (number of variables per precision).

Program	Initial		Error Threshold 10^{-4}						Error Threshold 10^{-6}					
			B		B+L		L		B		B+L		L	
	D	F	D	F	D	F	D	F	D	F	D	F	D	F
bessel	26	0	1	25	26	0	26	0	1	25	26	0	26	0
gaussian	56	0	54	2	56	0	56	0	54	2	56	0	56	0
roots	16	0	1	15	16	0	16	0	1	15	16	0	16	0
polyroots	31	0	10	21	10	21	31	0	10	21	10	21	31	0
rootnewt	14	0	1	13	14	0	14	0	1	13	14	0	14	0
sum	34	0	24	10	11	23	11	23	24	10	11	23	34	0
fft	22	0	16	6	0	22	0	22	16	6	0	22	0	22
blas	17	0	1	16	0	17	0	17	1	16	0	17	0	17
ep	45	0	42	3	42	3	45	0	42	3	42	3	45	0
cg	32	0	26	6	2	30	2	30	28	4	13	19	13	19

Program	Initial		Error Threshold 10^{-8}						Error Threshold 10^{-10}					
			B		B+L		L		B		B+L		L	
	D	F	D	F	D	F	D	F	D	F	D	F	D	F
bessel	26	0	25	1	26	0	26	0	25	1	26	0	26	0
gaussian	56	0	54	2	56	0	56	0	54	2	56	0	56	0
roots	16	0	5	11	16	0	16	0	5	11	16	0	16	0
polyroots	31	0	10	21	10	21	31	0	10	21	10	21	31	0
rootnewt	14	0	5	9	14	0	14	0	5	9	14	0	14	0
sum	34	0	24	10	11	23	34	0	24	10	24	10	34	0
fft	22	0	16	6	22	0	22	0	16	6	22	0	22	0
blas	17	0	10	7	17	0	17	0	10	7	17	0	17	0
ep	45	0	42	3	42	3	45	0	-	-	-	-	-	-
cg	32	0	28	4	16	16	12	20	28	4	16	16	16	16

search space, the overall analysis time would be reduced. However, this is not necessarily true, especially when the number of variables removed is small. In some cases, removing variables from the search space can alter the search path of LCCSEARCH, which might result in a slower analysis time. For example, in the experiment with error threshold 10^{-4} for the `gaussian` program, Blame Analysis removes only 2 variables from the search space (Table 3), a small reduction that alters the search path and actually slows down the analysis. For `ep` and `cg` programs, the search space reduction results in analysis speed up for LCCSearch. However, the overhead of Blame Analysis causes Blame LCCSearch’s analysis time to be slower than LCCSearch when using error threshold 10^{-4} for `ep` and error thresholds 10^{-4} and 10^{-6} for `cg` program (Figure 11).

Table 2 shows the average analysis time speedup per program for all error thresholds. We observe analysis time speedups for 9 out of 10 programs. The largest speedup observed

Tab. 4: Speedup observed after precision tuning using configurations produced by BLAME LCCSEARCH and LCCSEARCH alone (L)

Program	Threshold 10^{-4}		Threshold 10^{-6}		Threshold 10^{-8}		Threshold 10^{-10}	
	B+L	L	B+L	L	B+L	L		
bessel	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
gaussian	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
roots	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
polyroots	0.4%	0.0%	0.4%	0.0%	0.4%	0.0%	0.4%	0.0%
rootnewt	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
sum	39.9%	39.9%	39.9%	0.0%	39.9%	0.0%	0.0%	0.0%
fft	8.3%	8.3%	8.3%	8.3%	0.0%	0.0%	0.0%	0.0%
blas	5.1%	5.1%	5.1%	5.1%	0.0%	0.0%	0.0%	0.0%
ep	0.6%	0.0%	0.6%	0.0%	0.6%	0.0%	-	-
cg	7.7%	7.7%	7.9%	7.9%	7.9%	7.4%	7.9%	7.9%

is 38.42 times and corresponds to the analysis time of program `rootnewt` program. Whenever we observe a large speedup, Blame Analysis removes a large number of variables from the search space of LCCSearch, at least for error thresholds 10^{-4} and 10^{-6} (Table 3). The only experiment in which BLAME LCCSEARCH is slower than LCCSEARCH is when analyzing the `cg` program, however the slowdown is only 1%. We note that in Table 3, there are programs for which B+L and L return the configurations where all variables are in double precision as the output. We remind that this is because when the configuration proposed by LCCSearch results in a program that runs slower than the original program, we produce the original configuration, which is the configuration where all variables are in double precision, as the output.

Evaluation of Proposed Configurations. Table 3 shows the configurations found by Blame Analysis (B), Blame LCCSearch (B+L) and LCCSearch (L). The configurations show the numbers of variables in double precision (D) and single precision (F). Our evaluation shows that Blame Analysis is effective in removing variables from LCCSEARCH search space. In particular, in all 39 experiments (4 error thresholds for 9 programs, and 3 error threshold for 1 program), Blame Analysis successfully removes at least one variable from the search space. If we consider the number of variables removed in all 39 experiments, in average, Blame Analysis removes 39.85% variables from the search space, and in median, it removes 28.34% variables.

The configurations proposed by Blame LCCSearch and LCCSearch agree in 28 out of 39 experiments, and differ in 11 out of 39 experiments. Table 4 shows the speedup observed when we apply the configurations proposed by Blame LCCSearch and LCCSEARCH to the original program. In all 11 cases in which the two configurations differ, the configuration proposed by Blame LCCSearch translates into a program that runs faster. In particular, in 3 cases, the speedup is significant, which is 39.9% compared to 0%.

Among 31 out of 39 (79.48%) experiments, Blame LCCSearch finds configurations that differ from the configurations suggested by Blame Analysis. Among those, 9 experiments

Tab. 5: Overhead of Blame Analysis

Program	Execution (s)	Analysis (s)	Overhead
cg	3.52	185.45	52.55x
ep	34.7	1699.74	48.98x

(29.03%) produces configurations that are different from the original program. This shows that LCCSearch is still useful in tuning the configurations found by Blame Analysis. And even though, in about 70% cases, LCCSearch does not find any configurations that differ from the original program, it is still useful in confirming that the original program is already an optimized one in term of accuracy and performance.

Overhead of Blame Analysis. Table 5 shows the overhead of Blame Analysis for `cg` and `ep` program. By itself, Blame Analysis introduces up to 53x slowdown, which is comparable to the run-time overhead reported in other widely-used instrumentation-based tools such as JALANGI or VALGRIND [33, 27]. For the rest of our benchmarks, the overhead is relatively negligible (< 1 second).

Figure 11 shows the impact of Blame Analysis time on the Blame LCCSearch analysis time. Overall, Blame Analysis causes Blame LCCSearch analysis time to be slower than LCCSearch analysis time in 3 out of 7 cases, which are error threshold 10^{-4} for `ep` program, and error thresholds 10^{-4} and 10^{-6} for `cg` program.

5.3 Discussion

We note that our tool does not guarantee that the transformed program produces an accurate enough answer for all inputs. These are different and more difficult problems. Our tool relies on the user to provide a set of representative program inputs to be used within the tuning process. Work on generating test cases specializing for floating-point programs is available, which can be used as a complement to our work [12, 6].

We currently require the developers to map the type configuration we produced to the source code manually. This process might be error prone and require a significant amount of work, especially for large programs. We hope to automate this process in the future.

For this study, we used four intermediate precisions, `db4`, `db6`, `db8`, and `db10` to track precision requirements during the analysis. This proved a good trade-off between the quality of the solution and runtime overhead. For some programs, increasing the granularity of intermediate precisions may lead to more variables kept in low precision. This would be another subject for future work.

Another direction is to use Blame Analysis as an intra-procedural analysis, rather than an inter-procedural analysis as presented in this paper. Concretely, we can apply it on each procedure and use the configurations inferred for each procedure to infer the configuration for the entire program. Doing so will enable the opportunity for parallelism and might greatly improve the analysis time in modular programs.

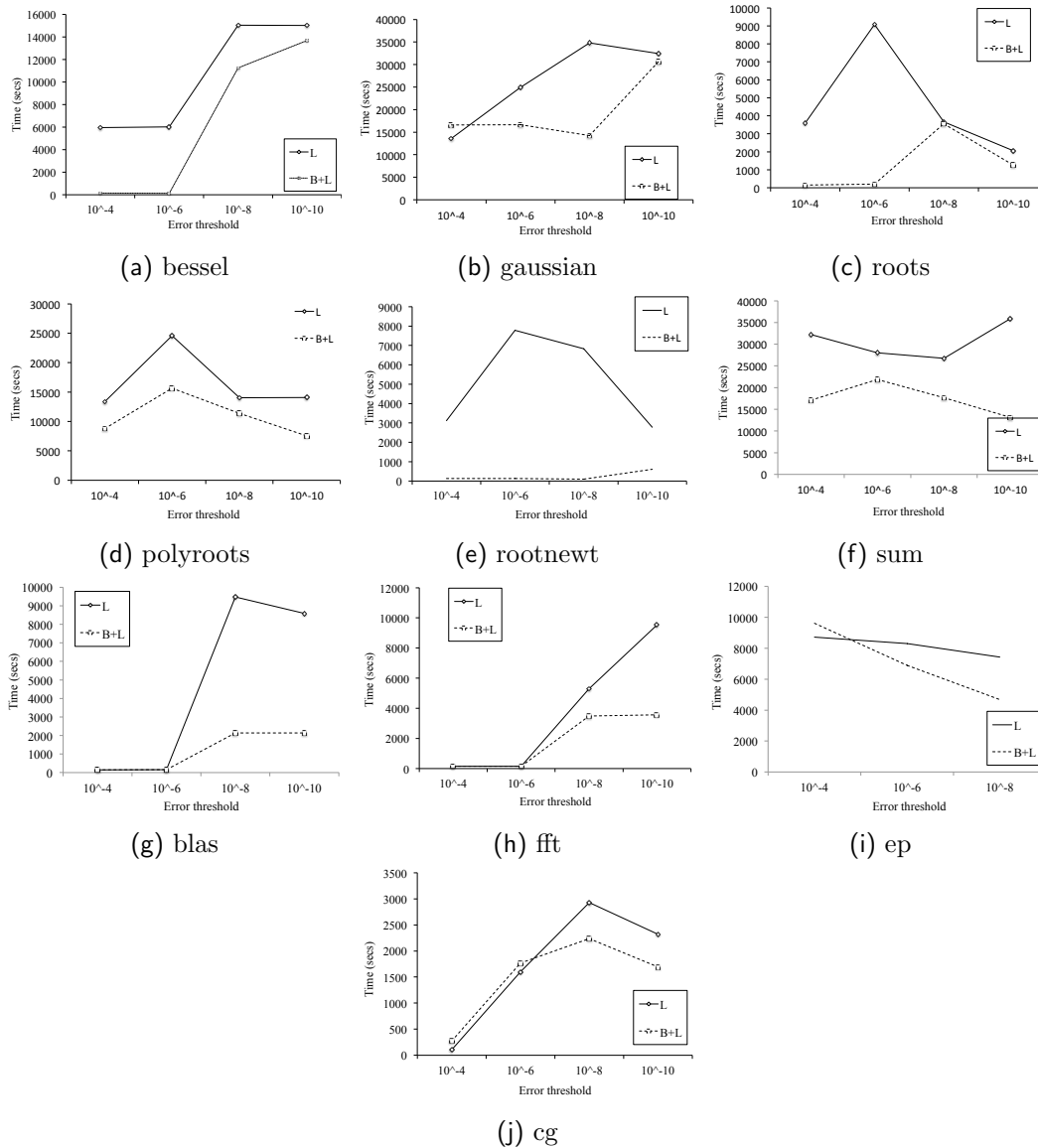


Fig. 10: Analysis time comparison between LCCSearch (L) and Blame LCCSearch (B+L). The vertical axis shows the analysis time. The horizontal axis shows the error thresholds used in each experiment. In these graphs, lower curve means to be more efficient.

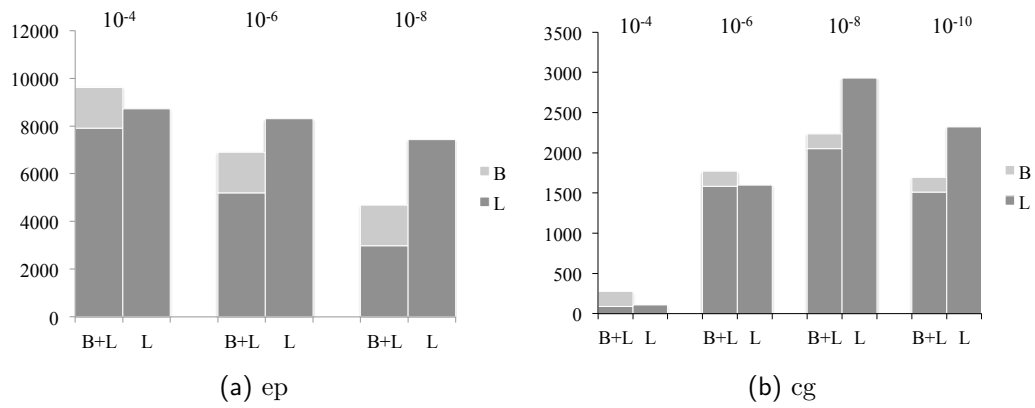


Fig. 11: Analysis time breakdown for Blame LCCSearch and LCCSearch for two NAS benchmark programs.

6 Related Work

Our approach considers the problem of automatically finding a type configuration for floating-point variables so that the resulting program still produces an accurate enough answer while running at least as fast as the original program. In recent work developed independently of our own, Lam et al. propose a framework for automatically finding mixed-precision floating-point computation [21]. This work appears to be most similar to ours. Their approach finds double precision instructions that can be replaced by single precision. They propose a brute-force search using heuristics. Their goal is to maximize the number of instructions that can be changed to single precision, and they operate on the binary code. The configurations proposed might not produce accurate results for tested inputs, and speedup is not explicitly considered as a goal. Furthermore, precision changes are described at the instruction level, thus it might be more difficult to map them to the source code.

Darulova et. al [15] develop a method for compiling a real-valued implementation program into a finite-precision implementation program, such that the finite-precision implementation program meets all desired precision with respect to the real numbers. Schkufza et. al [32] develop a method for optimization of floating-point programs using stochastic search by randomly applying a variety of program transformations, which sacrifice bit-wise precision in favor of performance. *FloatWatch* is a dynamic execution profiling tool for floating point programs which is designed to identify instructions that can be computed in a lower precision [9]. It works by first computing the overall range of values for each instruction of interest. Using this information, the tool recommends to use less precision if possible. Darulova and Kuncak also implemented a dynamic range analysis feature for the `Scala` language [14]. The approach uses interval and affine forms to represent the input, and examine how errors are magnified by each operation during execution. Their work might also be used for precision tuning purposes, by first computing a dynamic range for each instruction of interest and then tuning the precision based on the computed range, similar to *FloatWatch*. However, range analysis often incurs overestimates too large to be useful for precision tuning analysis. *Gappa* is another tool that uses range analysis to verify and prove formal properties of floating-point programs [16]. In the context of precision tuning, one can use *Gappa* to verify ranges for certain variables and expressions in a program, and then choose the appropriate precision for these variables and expressions. Nevertheless, *Gappa* scales only to small programs with simple structures and several hundreds of operations, and thus is used mostly for verifying elementary functions.

Our work is also related to a large body of work on accuracy analysis. [7] presented a dynamic analysis approach for finding accuracy problems. Their approach computes every floating-point instruction side by side in higher precision. The higher precision computation is stored in a *shadow value*. If the differences between the original value and the shadow value become too large, their tool reports a potential accuracy problem. *FPIinst* is another tool that computes floating point errors for the purpose for detecting accuracy problem [2]. It also computes a shadow value side by side, but it stores an absolute error in double precision instead. [22] proposed a tool for detecting cancellation. Cancellation is detected by first computing the exponent of the result and the operands. If the exponent of the result is less than the maximum of those of the two operands, a cancellation has occurred. *Fluctuat* is

another static analysis tool based on abstract interpretation and affine arithmetic to analyze the propagation of rounding errors in C programs. It can detect potential errors like runtime errors or unstable tests due to numerical problems [17]. Our tool can complement accuracy analysis for debugging purposes in the following way. It can attempt to tune the set of potentially error-generating floating-point instructions to have higher precision until the accuracy problem goes away. The cost model could be changed to favor a configuration that requires the fewest changes.

In addition, our work is related to code autotuning to improve performance [18, 34, 35, 29, 8]. That previous work has however not tried to tune floating-point precision in the way this work does. Finally, our work on Blame Analysis is related to other dynamic analysis tools that employ shadow execution and instrumentation [33, 28, 10, 27]. These tools, however, are designed as a general dynamic analysis framework rather than specializing in analyzing floating-point programs like ours.

7 Conclusion

In this paper, we have presented an automated dynamic technique to assist developers in tuning floating-point precision. Our technique comprises two novel algorithms, entitled Blame Analysis and LCCSearch, which effectively search through the type configuration search space, to infer a local minimum configuration that, when applied to the original program, results in a new program that uses less precision, is accurate enough and at least as fast as the original program. We implement our technique as an open source tool, which is available at <https://github.com/corvette-berkeley>. Initial evaluation on a set of seven GSL programs and two NAS benchmark programs show encouraging results: we are able to find configurations that translate to as high as 40% execution time speed up. In addition, Blame Analysis helps to speed up the analysis time of LCCSearch 9 times on average.

In the future, we would like to apply our analysis to a wider range of applications to gain better statistical results. We also want to explore different strategies to scale up our approach to larger programs. One of the directions is to first employ our technique as an intra-procedural analysis, then use the configuration inferred for each procedure to infer the configuration for the entire program. This will be a subject for future work.

References

- [1] C numerics library. <http://http://www.cplusplus.com/reference/cmath/>.
- [2] D. An, R. Blue, M. Lam, S. Piper, and G. Stoker. Fpinst: Floating point error analysis using dyninst, 2008.
- [3] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.
- [4] D. H. Bailey. Resolving numerical anomalies in scientific computation. <http://www.davidhbailey.com/dhbpapers/numerical-bugs.pdf>, 2008.
- [5] T. Bao and X. Zhang. On-the-fly detection of instability problems in floating-point program execution. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 817–832. ACM, 2013.
- [6] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 549–560. ACM, 2013.
- [7] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In J. Vitek, H. Lin, and F. Tip, editors, *PLDI*, pages 453–462. ACM, 2012.
- [8] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC. see <http://www.icsi.berkeley.edu/~bilmes/hipac>.
- [9] A. W. Brown, P. H. J. Kelly, and W. Luk. Profiling floating point value ranges for reconfigurable implementation. In *Proceedings of the 1st HiPEAC Workshop on Reconfigurable Computing*, pages 6–16, 2007.
- [10] D. Bruening, T. Garnett, and S. P. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, pages 265–275, 2003.
- [11] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. 2008.
- [12] W. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 43–52, 2014.

-
- [13] G. P. Contributors. GSL - GNU scientific library - GNU project - free software foundation (FSF). <http://www.gnu.org/software/gsl/>, 2010.
- [14] E. Darulova and V. Kuncak. Trustworthy numerical computation in scala. In C. V. Lopes and K. Fisher, editors, *OOPSLA*, pages 325–344. ACM, 2011.
- [15] E. Darulova and V. Kuncak. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 235–248, New York, NY, USA, 2014. ACM.
- [16] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Trans. Comput.*, 60(2):242–253, Feb. 2011.
- [17] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védryne. Towards an industrial use of fluctuat on safety-critical avionics software. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '09, pages 53–69, Berlin, Heidelberg, 2009. Springer-Verlag.
- [18] M. Frigo. A Fast Fourier Transform compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.
- [19] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.
- [20] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, ARITH '01, pages 155–, Washington, DC, USA, 2001. IEEE Computer Society.
- [21] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In A. D. Malony, M. Nemirovsky, and S. P. Midkiff, editors, *ICS*, pages 369–378. ACM, 2013.
- [22] M. O. Lam, J. K. Hollingsworth, and G. W. Stewart. Dynamic floating-point cancellation detection. In *1st International Workshop on High-performance Infrastructure for Scalable Tools*, 2011.
- [23] M. O. Lam, J. K. Hollingsworth, and G. W. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39(3):146–155, 2013.
- [24] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004)*, 20-24 March 2004, San Jose, CA, USA, pages 75–88. IEEE Computer Society, 2004.
- [25] X. S. Li, J. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. Thompson, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002.

- [26] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkh Basel, 1st edition, 2009.
- [27] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07*, pages 89–100, New York, NY, USA, 2007. ACM.
- [28] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In A. Moshovos, J. G. Steffan, K. M. Hazelwood, and D. R. Kaeli, editors, *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pages 2–11. ACM, 2010.
- [29] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.
- [30] T. Ravitch. LLVM Whole-Program Wrapper @ONLINE, Mar. 2011.
- [31] W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow. New implementations and results for the nas parallel benchmarks 2. In *In 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.
- [32] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 53–64, New York, NY, USA, 2014. ACM.
- [33] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In B. Meyer, L. Baresi, and M. Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 488–498. ACM, 2013.
- [34] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series*. Institute of Physics Publishing, June 2005.
- [35] C. Whaley. Automatically Tuned Linear Algebra Software (ATLAS). math-atlas.sourceforge.net, 2012.
- [36] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.