Floating-Point Precision Tuning Using Blame Analysis

Cuong Nguyen[1], Cindy Rubio-González[1], Benjamin Mehne[1], Koushik Sen[1], James Demmel[1], William Kahan[1], Costin Iancu[2], Wim Lavrijsen[2], David H. Bailey[2], and David Hough[3]

[1]EECS Department, UC Berkeley, {nacuong, rubio, bmehne, ksen, demmel, wkahan}@cs.berkeley.edu
[2]Lawrence Berkeley National Laboratory, {cciancu, wlavrijsen, dhbailey}@lbl.gov
[3]Oracle Corporation, david.hough@oracle.com

ABSTRACT

While tremendously useful, automated techniques for tuning the precision of floating-point programs face important scalability challenges. We present BLAME ANALYSIS*, a novel dynamic approach that speeds up precision tuning.* BLAME ANALYSIS *performs floating-point instructions using different levels of accuracy for the operands. The analysis determines the precision of all operands such that a given precision is achieved in the final result. Our evaluation on ten scientific programs shows* BLAME ANALYSIS *is successful in lowering operand precision without sacrificing output precision. The best results are observed when using* BLAME ANALYSIS *to filter the inputs to the* PRECIMONIOUS *search-based tool. Our experiments show that combining* BLAME ANALYSIS *with* PRECIMONIOUS *leads to finding better results faster: the modified programs execute faster (in three cases, we observe as high as 39.9% program execution speedup) and the combined analysis time is up to 38× faster than* PRECIMONIOUS *alone.*

1. INTRODUCTION

Algorithmic [31, 3, 10] or automated program transformation techniques [25, 18] to tune the precision of floating-point variables in scientific programs have been shown to significantly improve execution time. Given a developer specified precision constraint, their goal is to maximize the volume of program data stored in the lowest native precision, which, generally, results in improved memory locality and faster arithmetic operations.

Since tuning floating-point precision is a black art that requires both application specific and numerical analysis expertise, automated program transformation tools are clearly desirable and they have been shown to hold great promise. State-of-the-art techniques employ dynamic analyses that search through the program instruction space [18] or through the program variable/data space [25] using algorithms that are quadratic or worse in the number of instructions or variables. Due to their empirical nature, multiple independent searches with different precision constraints are required to find a solution that improves the program execution time.

We attempt to improve the space and time scalability of precision tuning with the program size using a two pronged approach. We explore better search algorithms using a dynamic program analysis, refered to as BLAME ANALYSIS, designed to tune precision in a single execution pass. BLAME ANALYSIS does not attempt to improve execution time. We then explore combined approaches, using BLAME ANALYSIS to bootstrap our PRECIMONIOUS [25] search-based tool which has been designed to reduce precision and to improve execution time.

While search-based tools attempt to determine an optimal solution that leads to faster execution time, our BLAME ANALYSIS is designed to determine a solution fast, using a combination of concrete and shadow program execution. The main insight of the analysis is that given a target instruction together with a precision requirement, one can build a *blame* set that contains all other program instructions with minimum precision, type assignments for their operands that satisfy the precision criteria for the original instruction. As the execution proceeds, the analysis builds the *blame* sets for all instructions within the program. The solution associated with a program point is computed using a *merge* operation over all *blame* sets.

We have implemented BLAME ANALYSIS using the LLVM [20] compiler infrastructure and evaluate it on eight programs from the GSL library [11] and two programs from the NAS parallel benchmarks [26]. We have implemented both *offline* analyses on program execution traces, as well as *online* analyses that execute together with the program.

When considered standalone, BLAME ANALYSIS was always successful in lowering the precision of all test programs; it is effective in removing variables from the search space, reducing it in average by 39.85% of variables, and in median, by 28.34% of variables. As it is solely designed to lower precision, the solutions do not always improve the program execution time. The *offline analysis* is able to lower the precision of a larger number of variables than the *online* version. However, as the trace-based approach does not scale with the program size, only the *online* approach is practical. We have observed runtime overhead for the online analysis as high as 50×, comparable to other commercial dynamic analysis tools.

The biggest benefits of BLAME ANALYSIS are observed when using it as a pre-processing stage to reduce the search space for PRECIMONIOUS. When used as a filter, BLAME ANALYSIS always leads to finding better results faster. Total analysis time is 9× faster on average, and up to 38× faster in comparison to PRECIMONIOUS alone. In all cases in which the result differs from PRECIMONIOUS alone, the configuration produced by BLAME + PRECIMONIOUS translates into a program that runs faster. For three benchmarks, the additional speedup is significant, up to 39.9%.

We believe that our results are very encouraging and indicate that floating-point tuning of entire applications will become feasible in the near future. As we now understand the more subtle behavior of Blame Analysis, we believe we can improve both analysis speed and the quality of the solution. It remains to be seen if this approach to develop fast but conservative analyses can supplant the existing slow but powerful search-based methods. Nevertheless, our work proves that using a fast "imprecise" analysis to bootstrap another slow but precise analysis can provide a practical solution to tuning floating point in large code bases.

This work makes the following contributions:

- We present a single-pass dynamic program analysis for tuning floating-point precision, with overheads comparable to that of other commercial tools for dynamic program analysis, such as Valgrind [21].
- We present an empirical comparison between single-pass and search-based, dual-optimization purpose tools for floating-point precision tuning.
- We demonstrate powerful and fast precision tuning by combining the two approaches.

2. TUNING FLOATING-POINT PRECISION

Most programming languages provide support for the floating point data types `float` (single-precision 32-bit IEEE 754 floating point), and `double` (double-precision 64-bit IEEE 754 floating point). Some programming languages also offer support for `long double` (80-bit extended precision). Furthermore, software packages such as QD [17] provide support for even higher precision (data types `double-double` and `quad-double`). Because reasoning about floating-point programs is often difficult given the large variety of numerical errors that can occur, one common practice is using the highest available floating-point precision. While more robust, this can degrade program performance significantly.

The goal of floating point precision tuning approaches is to lower the precision of as many program variables and instructions as possible, while (1) producing an answer within a given error threshold, and (2) being faster with respect to the original program.

Tools that operate on the program variable space are particularly useful as they may suggest permanent changes to the application. For example, consider a program that declares three floating-point variables as `double x, y, z`. The analysis takes as input a required precision, e.g. 10^{-6}, and computes a type assignment over program variables. The tool may map the variables to the following data types: $x \mapsto$ `float`, $y \mapsto$ `double`, and $z \mapsto$ `float`. This means that if we rewrite the program to reflect the type changes, then it will produce a result within the error threshold while running faster.

One such state-of-the art tool is Precimonious [25], which systematically searches for a type assignment for floating-point program variables.

2.1 Scalability Challenges

Precimonious analysis time is determined by the execution time of the program under analysis, and by the number of variables in the program. The algorithm Precimonious uses requires program re-compilation and re-execution for different type assignments. The search is based on the Delta-Debugging algorithm [32], which exhibits a worst-case complexity of $O(n^2)$, where n is the number of variables in the program.

In practice, it is very difficult for programmers to predict how the type of a variable affects the overall precision of the program result and the Precimonious analysis has to consider *all* the variables within a program, both global and local. This clearly poses a scalability challenge to the overall approach. In our evaluation of Precimonious (Section 4), we have observed cases in which the analysis takes hours for programs that have fewer than 50 variables and native runtime less than 5 seconds. Furthermore, as the analysis is empirical, determining a good solution requires repeating it over multiple precision thresholds. A solution obtained for a given precision (e.g. 10^{-6}) will always satisfy lower thresholds, e.g. 10^{-4}. It is often the case that tuning for a higher precision results in a better solution than tuning directly for an original target lower precision.

To our knowledge, Precimonious and other automated floating point precision tuners [25, 18] use empirical search and exhibit scalability problems with program size or program runtime.

In this work, we develop Blame Analysis as a method to quickly identify program variables whose precision does not affect the final result, for *any* given target threshold. The analysis takes as input one or more precision requirements and executes the program *only once* while performing shadow execution. As output, it produces a listing specifying the precision requirements for different instructions in the program, which then can be used to infer which variables in the program can definitely be in single precision without affecting the required accuracy for the final result.

By itself, Blame Analysis can be used to lower program precision to a specified level. Note that, in general, lowering precision does not necessarily result in a faster program (e.g., cast instructions might be introduced, which could make the program slower than the higher-precision version). Blame Analysis focuses on the impact in accuracy, but does not consider the impact in the running time of the tuned program. Because of this, the solutions produced by Blame Analysis are not guaranteed to improve program performance, and thus a triage by programmers is required.

Blame Analysis can also be used in conjunction with search-based tools, as a pre-processing stage to reduce the search space. In the case of Precimonious, this approach has great potential to shorten the analysis time while obtaining a good solution. Figure 1 shows how removing variables from the search space affects the analysis time for the `blas` program from the `GSL` library [11], for the target precision 10^{-10}. The `blas` program performs matrix multiplication, and it declares 17 floating-point variables. As shown at the rightmost point in Figure 1, knowing a priori that seven out of 17 floating-point variables can be safely allocated as `float` reduces the Precimonious analysis time from 2.3 hours to only 35 minutes. This simple filtering accounts for a 4× speedup in analysis time.

In this paper, we evaluate the efficacy of Blame Analysis in terms of analysis overhead and quality of solution in both settings: 1) applied by itself; and 2) as a pre-processing stage for Precimonious.

3. BLAME ANALYSIS

Our implementation of Blame Analysis consists of two main components: an instrumentation component, and an integrated *online* Blame Analysis algorithm. Blame Analysis is performed side-by-side with the program execution

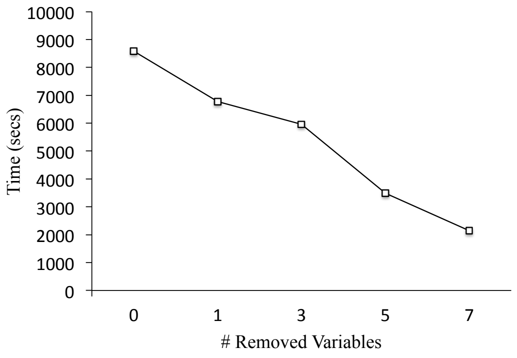


Figure 1: The effect of reducing PRECIMONIOUS search space on analysis time for the `blas` benchmark (error threshold 10^{-10}). The horizontal axis shows the number of variables removed from the search space. The vertical axis shows analysis time in seconds. In this graph, the lower the curve the more efficient the analysis becomes.

through the instrumentation. For each instruction, e.g. `add`, the BLAME ANALYSIS performs the instruction multiple times, each time using different precisions for the operands (e.g. double, float, double truncated to 10 digits, double truncated to 8 digits, etc.). The analysis checks the results to find which combinations of precisions for the operands satisfy a given precision requirement for the result. The satisfying precision combinations are recorded. The BLAME ANALYSIS algorithm maintains and updates a *blame set* for each instruction. The blame set associated with each instruction specifies the precision requirements for all operands such that the target instruction has the required precision. At the end of the analysis we collect all the variables that have been determined to be in single precision in all the relevant blame sets. These variables constitute the result of precision tuning and can also be removed from the PRECIMONIOUS search space.

3.1 Blame by Example

Consider the example program in Figure 2 which produces a result on line 24. When written using only `double` precision variables the result is `res = 0.`**`50000001`**`13`. When executed solely in `single` precision, the result is `res = 0.4999980927`. Assuming that we are interested in up to 8 digits of accuracy, the required result would be `res` = 0.**50000001**xy, 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, we consider three precisions: `fl` (float), `db` (double) and `db`$_8$ (accurate up to 8 digits compared to the double precision value). More specifically, the value in precision `db`$_8$ represents 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 `db`$_8$. According to the IEEE 754-2008 standard, the binary representation of v has 52 significand bits. We first find the number of bits that corresponds to 8 significant decimal digits in these 52 significand bits. The number of bits can be

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

Figure 2: Example

Table 1: The `r3 = r2 - t1` statement executed when operands have different precisions. The column Prec shows the precisions used for the operands (`fl` corresponds to float, `db` to double, and `db`$_8$ 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?
(fl,fl)	6.8635854721	7.3635854721	-0.5000000000	No
(fl,db$_8$)	6.8635854721	7.3635856000	-0.5000001279	No
(fl,db)	6.8635854721	7.3635856800	-0.5000002079	No
(db$_8$,fl)	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

computed as $lg(10^8) = 26.57$ bits. We therefore keep the 27 significant bits in the 52 significand bits, and set other bits in the significand 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, or 33 significant bits in the significand respectively, and set other bits to 0.

Consider the statement on line 23: `r3 = r2 - t1`. Since the double value of `r3` is `-0.`**`49999998`**`87`, this means that we require `r3` to be `-0.49999998` (i.e., the value matches 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. For example, (`fl`, `db`$_8$) means that `r2` has *float* precision, and `t1` has `db`$_8$ precision. For this particular statement, all but one operand precision combinations fail. Only until we try (`db`, `db`), then 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 $(\texttt{db}, \texttt{db})$, when the result is required to have precision $\texttt{db}_8$.

Statements that occur in loops are executed more than once, such as line 5: `res = res * a`. Assume we also require precision $\texttt{db}_8$ 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, 1.8408964200, 11.0453785200). The algorithm tries different precision combinations for the operands, and determines that precision $(\texttt{fl}, \texttt{db}_8)$ suffices. The second time the statement is encountered, the analysis records new double values (11.0453785200, 1.8408964200, 20.3333977750). After trying all precision combinations for the operands, it is determined that this time the precision required is $(\texttt{db}, \texttt{db}_8)$, 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 $(\texttt{fl}, \texttt{db}_8)$ and $(\texttt{db}, \texttt{db}_8)$ would result in the precision requirement $(\texttt{db}, \texttt{db}_8)$.

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 $\texttt{db}_8$, 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 can be allocated in single precision based on the program dependencies. In this example, only variable `factor` in function `mpow` is collected because it 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 our BLAME ANALYSIS algorithm and its implementation. 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 execution (Section 3.2), and an online BLAME ANALYSIS algorithm integrated inside the shadow execution runtime (Section 3.3). Finally, we present some heuristics and optimizations in Section 3.4.

3.2 Shadow Execution

Figure 3 introduces a kernel language used to formally describe our algorithm. 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 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.

$$\begin{array}{rcl}
Pgm & ::= & (L : Instr)* \\
Instr & ::= & x\ =\ y\ aop\ z \mid x\ =\ y\ bop\ z \mid \\
 & & \textbf{if}\ x\ \textbf{goto}\ L \mid \\
 & & x =\ nativefun(y) \mid x\ =\ c \\
aop & ::= & + \mid - \mid * \mid / \\
bop & ::= & = \mid \neq \mid < \mid \leq \\
nativefun & ::= & \textbf{sin} \mid \textbf{cos} \mid \textbf{fabs} \\
\multicolumn{3}{c}{L \in Labels \quad x, y, z \in Vars \quad c \in Consts}
\end{array}$$

Figure 3: Kernel Language

Procedure FAddShadow
Inputs
l : x = y + z : instruction
Outputs
Updating the shadow memory M and the label map LM
Method

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

Figure 4: Shadow Execution of `fadd` Instruction

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 floating-point 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 4 shows how M and LM are updated when an `fadd` 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 variables x, y, z, respectively, in that state. In this example, the procedure `FAddShadow` is the callback associated with the `fadd` instruction. The procedure re-interprets the semantics of the `fadd` instruction (line 3), but uses the shadow values for the corresponding operands (obtained on lines 1 and 2). Line 3 performs the additions and returns the results in the same precision as the operands. The label map LM is updated on line 4 to record that x was last updated at the instruction labeled l.

3.3 Building the Blame Sets

In this section, we formally describe Blame Analysis. Let A be the set of all memory addresses used by the program, L be the set of labels of all instructions in the program, P be the set of all precisions, i.e. $P = \{\texttt{fl}, \texttt{db}_4, \texttt{db}_6, \texttt{db}_8, \texttt{db}_{10}, \texttt{db}\}$. Precisions `fl` and `db` stand for single and double precisions, respectively. Precisions $\texttt{db}_4$, $\texttt{db}_6$, $\texttt{db}_8$, $\texttt{db}_{10}$ denote numbers that are accurate up to 4, 6, 8 and 10 digits in double precision, respectively. We also define a total order on precisions as follows: $\texttt{fl} < \texttt{db}_4 < \texttt{db}_6 < \texttt{db}_8 < \texttt{db}_{10} < \texttt{db}$. In Blame Analysis we also maintain a blame map B that maps a pair of instruction label and precision to a set of pairs of instruction labels and precisions, i.e., $B \colon L \times P \to \mathcal{P}(L \times P)$, where $\mathcal{P}(S)$ denotes the power set of S. If $B(\ell, p) = \{(\ell_1, p_1), (\ell_2, p_2)\}$, then it means that during an execution if instruction labeled ℓ produces a value that is accurate up to precision p, then instructions labeled ℓ_1 and ℓ_2 must produce values that are accurate up to precision p_1 and p_2, respectively.

During Blame Analysis we update the blame map on the execution of every instruction. We initialize B to the empty map at the beginning of an execution. We illustrate how we update B using a simple generic instruction of the form $\ell : x = f(y_1, \ldots, y_n)$, where x, $y_1, \ldots, y_n$ are variables and f is an operator, which could be $+$, $-$, $*$, `sin`, `log`, etc. In an execution consider a state where this instruction gets executed. Let us assume that $\&x$, $\&y_1$, $\ldots \&y_n$ denote the addresses of the variables x, $y_1, \ldots, y_n$, respectively, in that state.

When the instruction $\ell : \mathsf{x} = \mathsf{f}(\mathsf{y}_1, \ldots, \mathsf{y}_n)$ is executed during concrete execution, we also perform a side-by-side shadow execution of the instruction to update $B(\ell, p)$ for each $p \in P$ as follows. We use two functions, `BlameSet` and merge $\sqcup$, to update $B(\ell, p)$. The function `BlameSet` receives an instruction and a precision requirement as input, and returns the precision requirements for a set of instructions. Figure 5 shows the pseudo-code of the function `BlameSet`. The function 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). $\mathsf{truncs(s,p)}$ returns the floating-point value corresponding to the precision p given the shadow value s. $\mathsf{trunc(x,p)}$ returns x truncated to precision p. Line 2 obtains the shadow values corresponding to all operand variables. Then, the function finds the minimal precisions $\mathsf{p}_1, \ldots, \mathsf{p}_n$ such that if we apply f on $\mathsf{s}_1, \ldots, \mathsf{s}_n$ truncated to precisions $\mathsf{p}_1, \ldots, \mathsf{p}_n$, respectively, then the result truncated to precision p is equal to the correct result computed on line 1. We then pair each p_i with $LM[\&\mathsf{y}_i]$, the last instruction that computed the value y_i, and return the resulting set of pairs of instruction labels and precisions.

The merge function $\sqcup$ is defined as

$$\sqcup \colon \mathcal{P}(L \times P) \times \mathcal{P}(L \times P) \to \mathcal{P}(L \times P)$$

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

Given the functions `BlameSet` and merge $\sqcup$, we compute $B(\ell, p) \sqcup$ $\texttt{BlameSet}(\ell : \mathsf{x} = \mathsf{f}(\mathsf{y}_1, \ldots, \mathsf{y}_n), p)$ and use the resulting set to update $B(\ell, p)$.

At the end of an execution we get a non-empty map B. Suppose we want to make sure that the results computed by a given instruction labeled ℓ_{out} is accurate up to precision

```
function BlameSet
Inputs
  ℓ: x = f(y_1, ..., y_n)  :  instruction with label ℓ
                        p  :  precision requirement
Outputs
  {(ℓ_1, p_1), ..., (ℓ_n, p_n)}  :  precision requirements of the
                                    instructions that computed the operands
Method
  1 correct_res = truncs($M$[&x],p)
  2 (s_1, ..., s_n) = ($M$[&y_1], ..., $M$[&y_n])
  3 find minimal precisions p_1, ..., p_n such that the following holds:
  4     (v_1,...,v_n) = (truncs(s_1,p_1),...,truncs(s_n,p_n))
  5     trunc(f(v_1,...,v_n),p)==correct_res
  6 return {($LM$[&y_1],p_1), ..., ($LM$[&y_n],p_n)}
```

Figure 5: `BlameSet` Procedure

p. Then we want to know what should be the accuracy of the results computed by the other instructions so that the accuracy of the result of the instruction labeled ℓ_{out} is p. We compute this using the function $\texttt{Accuracy}(\ell_{out}, p, B)$ which returns a set of pairs instruction labels and precisions, such that if (ℓ', p') is present in $\texttt{Accuracy}(\ell_{out}, p, B)$, then the result of executing the instruction labeled ℓ' must have a precision of at least p'. $\texttt{Accuracy}(\ell, p, B)$ can then be defined recursively as follows.

$$\texttt{Accuracy}(\ell, p, B) = \{(\ell, p)\} \sqcup \bigsqcup_{(\ell', p') \in B(\ell, p)} \texttt{Accuracy}(\ell', p', B)$$

Once we have computed $\texttt{Accuracy}(\ell_{out}, p, B)$, we know that if (ℓ', p') is present in $\texttt{Accuracy}(\ell_{out}, p, B)$, then the instruction labeled ℓ' must be executed with precision at least p' if we want the result of executing instruction labeled ℓ_{out} to have a precision p.

3.4 Implementation Optimizations

To attain scalability for large or long running programs, the implementation of Blame Analysis has to address both space and time optimization concerns and we have experimented with *offline* and *online* analyses.

Offline Blame Analysis first collects the complete execution trace, and builds the blame set for each executed instruction. As each instruction is examined only once, merging of operand precision is not required. Thus, when compared to the online version, the offline approach exhibits lower analysis overhead per instruction, as well as being able to produce better quality solutions. However, the size of the execution trace and the *blame* function explodes for large inputs and long running programs. For example, when running offline Blame Analysis on the `ep` NAS [26] benchmark with input class[1] `S`, offline Blame Analysis returns an out of memory error code after using all 256 GB memory in our system.

Our implementation of online Blame Analysis is more memory efficient because the size of the blame sets is bounded by the number of *static* instructions in the program. As shown in Section 4.2, the maximum analysis working set size is 81 MB, in case of program `ep`. On the other hand, the information for each instruction has to be merged across all its dynamic invocations and the online analysis has to be further optimized for speed. In our implementation, we allow developers to specify what part of the program they

[1]Class S is a small input, designed for serial execution.

are interested to analyze. For short running programs, such as functions within the GSL [11] library, examining all instructions is feasible. Most long running scientific programs fortunately use iterative solvers, rather than direct solvers. In this case, analyzing the last few algorithmic 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 considered. If no options are specified, Blame Analysis by default will be performed throughout the entire program execution.

Overall, our results indicate that offline Blame Analysis is high in space usage but fast (lower overhead per instruction) and produces a better solution than the online version. Online Blame Analysis is low in space usage, and worse in running time and solution quality than the offline version. For brevity, all the results reported in the rest of this paper are obtained using the online analysis.

4. EXPERIMENTAL EVALUATION

The Blame Analysis architecture is described in Figure 6. We buid the analysis on top of the LLVM compiler infrastructure [20]. The analysis takes as input: (1) LLVM bitcode of the program under analysis, (2) a set of test inputs, and (3) analysis input parameters that include the target instruction and the desired error threshold(s). We use the original Precimonious benchmarks which have been modified by experts to provide acceptability criteria for the result precision. For Blame Analysis we select the acceptability code developed for Precimonious as the target instruction set. *Thus, the results provided by both analyses always satisfy the developer specified precision criteria.*

The analysis result consists of the set of variables that can be in single precision. In this section, we present the evaluation of Blame Analysis by itself, as well as when used as a pre-processing stage for Precimonious. We refer to the latter as Blame + Precimonious. We compare this combined approach with using Precimonious alone, and perform an evaluation in terms of the analysis running time, and the impact of the analysis results in improving program performance.

4.1 Experiment Setup

We present results for eight programs from the GSL library [11] and two programs from the NAS parallel benchmarks [26]. We use Clang with no-optimization [2] and a Python-wrapper [24] to build whole-program (or whole-library) LLVM bitcode. Note that we do apply optimization level `-O2` when performing final performance measurements on the tuned programs. We run our experiments on an Intel(R) Xeon(R) CPU E5-4640 0 @ 2.40GHz 8-core machine running Linux with 256GB RAM.

For the NAS parallel benchmarks (programs `ep` and `cg`), we use the provided input Class A. For other programs, we generate 1000 random floating-point inputs, which we classify into groups based on code coverage. We then pick one input from each group, i.e., we want to maximize code coverage while minimizing the number of inputs to consider. We log and read the inputs in hexadecimal format to ensure that the inputs generated and the inputs used match at the bit level.

[2]Optimization sometimes removes floating-point variables, which causes the set of variables at the bitcode level to differ from the variables at the source code level.

Table 3: Speedup observed after precision tuning using configurations produced by Blame + Precimonious (B+P) and Precimonious alone (P)

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

	Threshold 10^{-8}		Threshold 10^{-10}	
Program	B+P	P	B+P	P
bessel	0.0%	0.0%	0.0%	0.0%
gaussian	0.0%	0.0%	0.0%	0.0%
roots	0.0%	0.0%	0.0%	0.0%
polyroots	0.4%	0.0%	0.4%	0.0%
rootnewt	0.0%	0.0%	0.0%	0.0%
sum	39.9%	0.0%	0.0%	0.0%
fft	0.0%	0.0%	0.0%	0.0%
blas	0.0%	0.0%	0.0%	0.0%
ep	0.6%	0.0%	-	-
cg	7.9%	7.4%	7.9%	7.9%

Table 4: Average analysis time speedup of Blame Analysis compared to Precimonious

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×

In our experiments, we use error thresholds 10^{-4}, 10^{-6}, 10^{-8}, and 10^{-10}, which correspond to 4, 6, 8 and 10 digits of accuracy, respectively. Additionally, for NAS programs `ep` and `cg`, we configure Blame Analysis to consider only the last 10% of the executed instructions. For the rest of the programs, Blame Analysis considers all the instructions executed.

4.2 Analysis Running Time and Memory

This section compares the performance of Blame Analysis and its combination with Precimonious. We also compare the *online* and *offline* versions of Blame Analysis in terms of memory usage.

Blame Analysis

By itself, Blame Analysis introduces up to 50× slowdown, which is comparable to the run-time overhead reported by widely-used instrumentation-based tools such as Valgrind [21] and Jalangi [28]. Table 5 shows the overhead for programs `cg` and `ep`. For the rest of our benchmarks, the overhead is relatively negligible (< 1 second).

Blame + Precimonious

To measure the analysis time of the combined analyses, we add the analysis time of Blame Analysis and the search

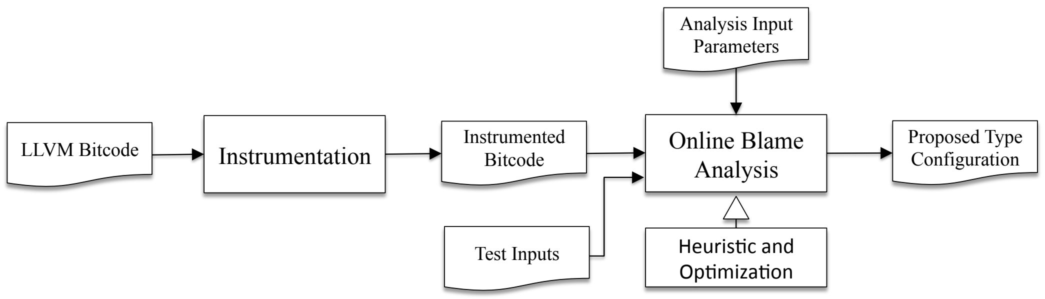


Figure 6: Architecture Blame Analysis implementation

Table 2: Configurations found by Blame Analysis (B), Blame + Precimonious (B+P), and Precimonious alone (P). 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+P, and P (number of variables per precision). × denotes the cases where the tools select the original program as fastest.

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

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

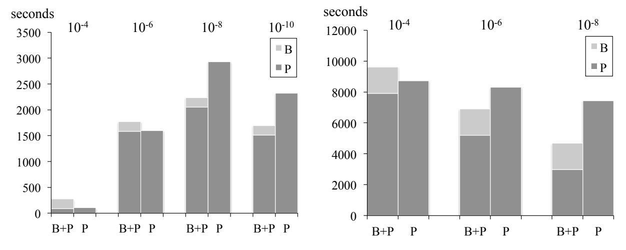
Table 5: Overhead of Blame Analysis

Program	Execution (sec)	Analysis (sec)	Overhead
cg	3.52	185.45	52.55×
ep	34.70	1699.74	48.98×

time of Precimonious for each error threshold. Figure 7 shows the analysis time of Blame + Precimonious (B+P) and Precimonious (P) for each of our benchmarks. We use all error thresholds for our benchmarks, except for program `ep`. The original version of this program uses error threshold 10^{-8}, thus we do not consider error threshold 10^{-10}.

Table 4 shows the average speedup per program for all error thresholds. We observe analysis time speedups for 9 out of 10 programs. The largest speedup observed is 38.42× and corresponds to the analysis of program `rootnewt`. Whenever we observe a large speedup, Blame Analysis removes a large number of variables from the search space of Precimonious, at least for error thresholds 10^{-4} and 10^{-6} (see Table 2). This translates into significantly shorter analysis time for Precimonious. The only experiment in which Blame + Precimonious is slower than Precimonious on average, is when analyzing the program `cg`, however the slowdown observed is only 1%.

Our results show that Blame + Precimonious is faster than Precimonious in 31 out of 36 experiments. In general, we would expect that as variables are removed from the search space, the overall analysis time will be reduced. However, this is not necessarily true, especially when very few variables are removed. In some cases, removing variables from the search space can alter the search path of Precimonious, which results in a slower analysis time. For example, in the experiment with error threshold 10^{-4} for `gaussian`, Blame Analysis removes only two variables from the search space (see Table 2), a small reduction that changes the search path and actually slows down the analysis. For programs `ep` and `cg`, the search space reduction results in analysis time speedup for Precimonious. However, the overhead of Blame Analysis causes the combined Blame + Precimonious running time to be slower than Precimonious when using error thresholds 10^{-4} and 10^{-6} for programs `ep` and `cg`, respectively (see Figure 8).



(a) ep (b) cg

Figure 8: Analysis time breakdown for Blame + Precimonious (B+P) and Precimonious (P) for two NAS benchmark programs

Memory

In our experiments, the *online* version of Blame Analysis uses up to 81 MB of memory. The most expensive benchmark in terms of analysis memory usage is program `ep`. For this program, the *offline* version of the analysis runs out memory (256 GB).

4.3 Analysis Results

Table 2 shows the type configurations found by Blame Analysis (B), Blame + Precimonious (B+P) and Precimonious (P), which consist of the numbers of variables in double precision (D) and single precision (F). It also shows the initial type configuration for the original program. Our evaluation shows that Blame Analysis is effective in lowering precision. In particular, in all 39 experiments (4 error thresholds for 9 programs, and 3 error thresholds for 1 program), Blame Analysis successfully identifies at least one variable as `float`. If we consider the number of variables removed in all 39 experiments, Blame Analysis removes from the search space 39.85% of the variables on average, with a median of 28.34%.

The type configurations proposed by Blame + Precimonious and Precimonious agree in 28 out of 39 experiments, and differ in 11 experiments. Table 3 shows the speedup observed when we tune the programs according to these type configurations. In all 11 cases in which the two configurations differ, the configuration proposed by Blame + Precimonious produces the best performance improvement. In particular, in three cases we observe 39.9% additional speedup.

In 31 out of 39 experiments, Blame + Precimonious finds configurations that differ from the configurations suggested by Blame Analysis alone. Among those, 9 experiments produce a configuration that is different from the original program. This shows that our analysis is conservative and Precimonious is still useful in further improving configurations found by Blame Analysis alone.

Note that for Blame Analysis, we have reported results only for the *online* version of the analysis. Our experiments indicate that the *offline* version has memory scalability problems and while its solutions sometimes are better in terms of the number of variables that can be lowered to single precision, it is not necessarily better at reducing analysis running time, or the running time of the tuned program.

5. DISCUSSION

While very useful, automated tools for floating-point precision tuning have to overcome scalability concerns. As it adds a constant overhead per instruction, the scalability of our single-pass Blame Analysis is determined solely by the program runtime. The scalability of Precimonious is determined by both program runtime and the number of variables in the program. We believe that our approach uncovers very exciting potential for the realization of tools able to handle large codes. There are several directions to improve the efficacy of Blame Analysis as a standalone tool, as well as a filter for Precimonious.

For this study, we used four intermediate precisions, $\mathtt{db}_4$, $\mathtt{db}_6$, $\mathtt{db}_8$, and $\mathtt{db}_{10}$ 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.

Another direction is to use Blame Analysis as an intraprocedural 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.

6. RELATED WORK

Precimonious [25] is a dynamic analysis tool for tuning floating-point precision. It employs an efficient Delta-Debugging based method to search through the program variable and function space to find a program that uses less precision and runs faster than the original program. Independently developed of Precimonious, Lam et al. also proposes a framework for finding mixed-precision floating-point computation [18]. Lam's approach uses a brute-force algorithm to find double precision instructions that can be replaced by single instructions. Their goal is to use as many single instructions in place of double instructions as possible, but not explicitly consider speedup as a goal. Blame Analysis differs from Precimonious and Lam's framework in that it performs a white-box analysis on the set of instructions executed by the program under analysis, rather than through searching. Thus, Blame Analysis is not bounded by the exponential size of the variable or instruction search space. Similar to Lam's framework, the goal of Blame Analysis is to minimize the use of double precision in the program under analysis.

Darulova et. al [13] 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, however the approach does not support mixed precision. Schkufza et. al [27] develops 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* [8] is a dynamic execution profiling tool for floating-point programs which is designed to identify instructions that can be computed in a lower precision by computing the overall range of values for each instruction of interest. As with other tools described in this paper, all the above also face scalability challenges.

Darulova and Kuncak [12] also implemented a dynamic range analysis feature for the Scala language that could 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 [14] is another tool that uses range analysis to verify and prove formal properties of floating-point programs. One could use Gappa to verify ranges for certain program variables and expressions, and then choose their appropriate precisions. Nevertheless, Gappa scales only to small programs with simple structures and several hundreds of operations, and thus is used mostly for verifying elementary functions.

A large body of work exists on accuracy analysis [6, 4, 5, 19, 15]. Benz et al. [6] present a dynamic approach that consists on computing every floating-point instructions side-by-side in higher precision, storing the higher precision values in shadow variables. FPInst [2] is another tool that computes floating point errors for the purpose for detecting accuracy problem. It also computes a shadow value side-by-side, but it stores an absolute error in double precision instead.

Another large area of research focused on improving performance is autotuning [7, 16, 23, 29, 30]. However, no previous work has tried to tune floating-point precision as discussed in this paper. Finally, our work on BLAME ANALYSIS is related to other dynamic analysis tools that employ shadow execution and instrumentation [28, 21, 22, 9]. These tools, however, are designed as general dynamic analysis frameworks rather than specializing in analyzing floating-point programs like ours.

7. CONCLUSION

We introduce a novel dynamic analysis designed to tune the precision of floating-point programs. Our implementation uses a shadow execution engine and when applied to a set of ten programs it is able to compute a solution with at most 50× runtime overhead. Our workload contains a combination of small to medium size programs, some that are long running. This is encouraging and similar to other widely-used dynamic analysis tools, such as Valgrind. The code is open source and available online[3].

When used by itself, BLAME ANALYSIS is able to lower the precision for all tests, but the results do not necessarily translate into execution time improvement. The largest impact is observed when using our analysis as a filter to prune the inputs to another floating-point tuning tool which searches through the variable space. In this case, BLAME ANALYSIS reduces the analysis time up to 38× and enables the tool to find new solutions that improve program execution time by as much as 39.9%.

We believe that our results are very encouraging and indicate that floating-point tuning of entire applications will become feasible in the near future. As we now understand the more subtle behavior of BLAME ANALYSIS, we believe we can improve both analysis speed and the quality of the solution. It remains to be seen if this approach to develop fast but conservative analyses can supplant the existing slow but powerful search-based methods. Nevertheless, our work proves that using a fast "imprecise" analysis to bootstrap another slow but precise analysis can provide a practical solution to tuning floating point in large code bases.

[3]URL witheld for blind review.

8. 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] H. Anzt, D. Lukarski, S. Tomov, and J. Dongarra. Self-adaptive multiprecision preconditioners on multicore and manycore architectures. *Proceedings of 11th International Meeting High Performance Computing for Computational Science, VECPAR*, 2014.

[4] T. Bao and X. 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*, pages 817–832, 2013.

[5] 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.

[6] 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, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 453–462. ACM, 2012.

[7] 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/phipac.

[8] 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.

[9] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2003.

[10] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perform. Comput. Appl.*, 21(4), Nov. 2007.

[11] G. P. Contributors. GSL - GNU scientific library - GNU project - free software foundation (FSF). http://www.gnu.org/software/gsl/, 2010.

[12] E. Darulova and V. Kuncak. Trustworthy numerical computation in scala. In C. V. Lopes and K. Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 325–344. ACM, 2011.

[13] E. Darulova and V. Kuncak. Sound compilation of reals. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 235–248. ACM, 2014.

[14] F. de Dinechin, C. Q. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using gappa. pages 242–253, 2011.

[15] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In M. Alpuente, B. Cook, and C. Joubert, editors, *Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.

[16] M. Frigo. A Fast Fourier Transform compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia*, May 1999.

[17] 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*, 2001.

[18] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *International Conference on Supercomputing, ICS'13,*, 2013.

[19] M. O. Lam, J. K. Hollingsworth, and G. W. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39(3):146–155, 2013.

[20] 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, 2004.

[21] 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, 2007.

[22] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization*, pages 2–11, 2010.

[23] 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.

[24] T. Ravitch. LLVM Whole-Program Wrapper @ONLINE, Mar. 2011.

[25] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13*, page 27, 2013.

[26] 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.

[27] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 9. ACM, 2014.

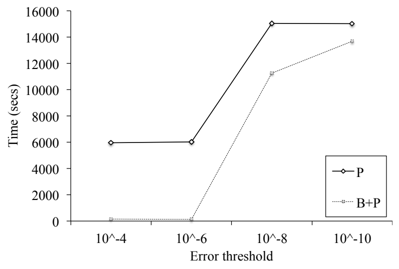
[28] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, pages 488–498, 2013.

[29] 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.

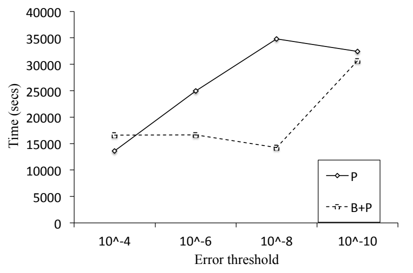
[30] C. Whaley. Automatically Tuned Linear Algebra Software (ATLAS). math-atlas.sourceforge.net, 2012.

[31] I. Yamazaki, S. Tomov, T. Dong, and J. Dongarra. Mixed-precision orthogonalization scheme and adaptive step size for ca-gmres on gpus. *Proceedings of 11th International Meeting High Performance Computing for Computational Science, VECPAR*, 2014.

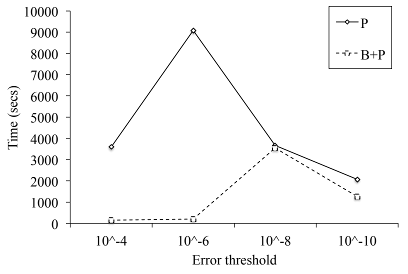
[32] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.



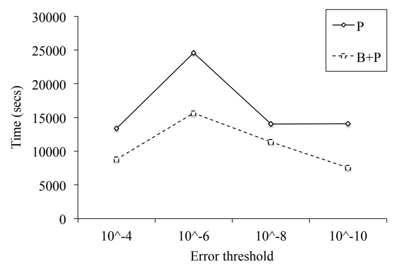
(a) bessel



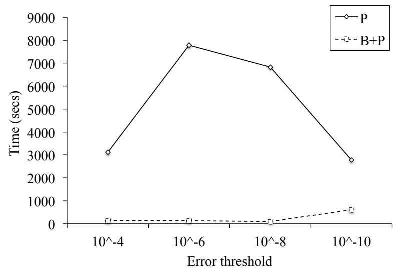
(b) gaussian



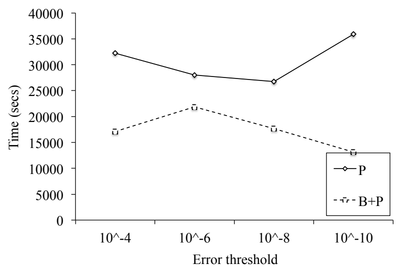
(c) roots



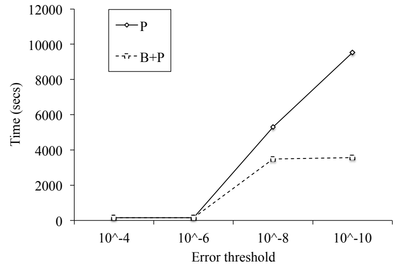
(d) polyroots



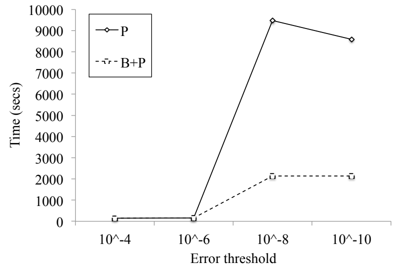
(e) rootnewt



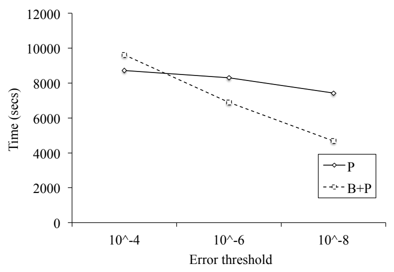
(f) sum



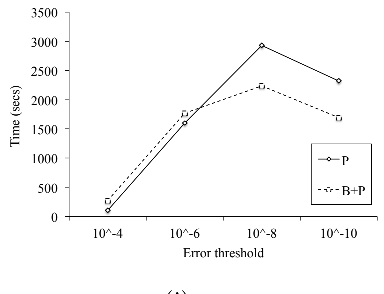
(g) blas



(h) fft



(i) ep



(j) cg

Figure 7: Analysis time comparison between PRECIMONIOUS (P) and BLAME + PRECIMONIOUS (B+P). The vertical axis shows the analysis time. The horizontal axis shows the error thresholds used in each experiment. In these graphs, a lower curve means more efficient.