Floating-Point Precision Tuning Using Blame Analysis

Cindy Rubio-González1, Cuong Nguyen2, Benjamin Mehne2, Koushik Sen2, James Demmel2, William Kahan2, Costin Iancu3, Wim Lavrijsen3, David H. Bailey3, and David Hough4

1University of California, Davis, crubio@ucdavis.edu
2University of California, Berkeley, {nacuong, bmehne, ksen, demmel, wkahan}@cs.berkeley.edu
3Lawrence Berkeley National Laboratory, {cciancu, wlavrijsen, dhbailey}@lbl.gov
4Oracle 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 their operands. The analysis determines the precision of all operands such that a given precision is achieved in the final result of the program. Our evaluation on ten scientific programs shows that Blame Analysis is successful in lowering operand precision. As it executes the program only once, the analysis is particularly useful when targeting reductions in execution time. In such cases, the analysis needs to be combined with search-based tools such as Precimonious. Our experiments show that combining Blame Analysis with Precimonious leads to obtaining better results with significant reduction in analysis time: the optimized programs execute faster (in three cases, we observe as high as 39.9% program speedup) and up to 38× faster than Precimonious alone.

CCS Concepts
•Software and its engineering → Dynamic analysis;
Software performance; Formal software verification; Software testing and debugging; •Mathematics of computing → Numerical analysis;

Keywords
floating point, mixed precision, program optimization

1. INTRODUCTION
Algorithmic [44, 2, 14] or automated program transformation techniques [35, 25] to tune the precision of floating-point variables in scientific programs have been shown to significantly improve execution time. Developers prescribe the accuracy required for the program result and the tools attempt to maximize the volume of data stored in the lowest native precision. Generally, this 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 have been shown to hold great promise. State-of-the-art techniques employ dynamic analyses that search through the program instruction space [25] or through the program variable/data space [35]. Due to the empirical nature, a quadratic or worse (in instructions or variables) number of independent searches (program executions) with different precision constraints are required to find a solution that improves the program execution time for a given set of inputs. While some search-based tools [35] attempt to only provide solutions that lead to faster execution time, others [25] provide solutions with no performance guarantees.

In this paper we present a novel method to perform floating-point precision tuning that combines concrete and shadow program execution, and it is able to find a solution after only a single execution. The main insight of Blame Analysis is that given a target instruction and a precision requirement, one can build a blame set that contains all other program instructions with their operands in minimum precision. In other words, given an instruction and a precision requirement, a blame set contains the precision requirements for the instructions that define the values of its operands. As the execution proceeds, each instruction is executed with multiple floating-point precisions for each operand and its blame set is updated. The solution associated with a program point is computed using a merge operation over all blame sets. This can be used to infer the set of program variables that can be declared as float instead of double while satisfying the precision requirements for a provided test input set. Note that, similar to [25], Blame Analysis can only reduce precision with no performance guarantees.

We have implemented Blame Analysis using the LLVM compiler infrastructure [27] and evaluated it on eight programs from the GSL library [17] and two programs from the NAS parallel benchmarks [37]. To provide more context, we also evaluated it against the Precimonious [35] search-
based tool. We have implemented both an offline analysis that executes on program execution traces, as well as an online analysis that executes together with the program.

Blame Analysis was always successful in lowering the precision of all test programs for the given test input sets: it identified on average that 40% of the program variables can be declared as float, 28% variables in median. The transformed programs did not only exhibit improved execution time. The offline analysis is able to lower the precision of a larger number of variables than the online version, but this comes with decreased scalability. For the online version, we bound the overhead we had to restrict the analysis in some cases to consider only the last stages of execution. Even with this restriction, the online analysis produced solutions while imposing running time overhead as high as 50×, comparable to other commercial dynamic analysis tools.

If reduction in execution time is desired, Blame Analysis can be combined with feedback-directed search tools such as Precimonious, which systematically searches for a type assignment for floating-point variables so that the resulting program executes faster. When using Blame Analysis to determine an initial solution for Precimonious, we always find better type assignments. The total analysis time is 9× faster on average, and up to 48× faster in comparison to Precimonious alone. In all cases in which the resulting type assignment differs from Precimonious alone, the type assignment produced by the combined analyses translates into a program that runs faster.

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 empirical analysis that executes together with the program.

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.
- We provide 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.

The rest of this paper is organized as follows. Section 2 presents an overview of precision tuning and current challenges. We describe Blame Analysis in Section 3, and present its experimental evaluation in Section 4. We then discuss limitations and future work in Section 5. Related work is discussed in Section 6. We conclude in Section 7.

2. TUNING FLOATING-POINT PRECISION

Programming languages provide support for multiple floating-point data types: float (single-precision 32-bit IEEE 754), double (double-precision 64-bit IEEE 754) and long double (80-bit extended precision). Software packages such as QD [24] 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 to use conservatively the highest available precision. While more robust, this can significantly degrade program performance. Many efforts [3, 4, 5, 13, 23, 28] have shown that using mixed precision can sometimes compute a result of the same accuracy faster than when using solely the highest precision arithmetic. Unfortunately, determining the appropriate precision combination requires domain-specific knowledge combined with advanced numerical analysis expertise.

Floating-point precision-tuning tools can help suggesting ways in which programs can be transformed to effectively use mixed precision. These tools serve multiple purposes. For a given test input set, one goal is to determine an optimal (minimal or maximal) set of program variables [35] or instructions [25], whose precision can be changed such that the “answer” is within a given error threshold. If the goal is to improve accuracy, expressions can be rewritten to reduce rounding errors [31]. Another goal is to reduce memory storage by maximizing the number of variables whose precision can be lowered. Finally, improving program performance is another important objective.

2.1 Design and Scalability Concerns

Our main interest is in tools that target scientific computing and use a dual objective by targeting both accuracy and performance. The state-of-the-art tools compute a solution by searching over global program state (variables or instructions). Thus the search maintains a “global” solution and it requires multiple executions. Due to the empirical nature and the heuristics to bound the search space and state, the solutions do not capture a global optimum.

From our perspective, particularly attractive are tools that operate on the program variable space, as they may suggest permanent changes to the application. The state-of-the-art is reflected by Precimonious [35], which systematically searches for a type assignment (also referred to as type configuration) for floating-point program variables. Its analysis time is determined by the execution time of the program under analysis, and by the number of variables in the program. The algorithm requires program re-compilation and re-execution for different type assignments. The search is based on the Delta-Debugging algorithm [45], which exhibits a worst-case complexity of $O(n^2)$, where $n$ is the number of variables in the program. To our knowledge, Precimonious and other automated floating point precision tuners [35, 25] use empirical search and exhibit scalability problems with program size or program runtime.

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., $10^{-4}$). Given a target precision, it is also
often the case that the solution determined independently for a higher precision provides better performance than the solution determined directly for the lower precision.

In this work, we set to develop a method that alleviates the scalability challenges of existing search-based floating-point precision tuning approaches by: (1) reducing the number of required program analyses/transformations/executions, and (2) performing only local, fine-grained transformations, without considering their impact on the global solution.

Our BLAME ANALYSIS is designed 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. When evaluating the approach we are interested in several factors: (1) quality of solution, i.e., how much data is affected, (2) scalability of the analysis, and (3) impact on the performance of the tuned program.

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). The analysis focuses on the impact in accuracy, but does not consider the impact in the running time. Because of this, the solutions produced are not guaranteed to improve program performance and a triage by programmers is required.

Even when triaged, solutions do not necessarily improve execution time. As performance is a main concern, we also consider combining BLAME ANALYSIS with dual objective search-based tools, as a pre-processing stage to reduce the search space. In the case of PRECIMONIOUS, this approach can potentially 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 [17], for the target precision $10^{-10}$. The blas program performs matrix multiplication. As shown at the rightmost point in Figure 1, knowing a priori that 7 out of 17 floating-point variables can be safely allocated as float reduces PRECIMONIOUS analysis time from 2.3 hours to only 35 minutes. This simple filtering accounts for a $4 \times$ speedup in analysis time.

In the rest of this paper, we present BLAME ANALYSIS and evaluate its efficacy in terms of analysis running time and quality of solution in two settings: (1) applied by itself, and (2) as a pre-processing stage for PRECIMONIOUS. We refer to quality of solution as whether the resulting type assignments lead to programs with faster execution time.

3. BLAME ANALYSIS

BLAME ANALYSIS consists of two main components: a shadow execution engine, and an integrated online analysis. The analysis is performed side-by-side with the program execution through instrumentation. For each instruction, e.g., fadd (floating-point addition), BLAME ANALYSIS executes the instruction multiple times, each time using different precisions for the operands. Examples of precision include float, double, and double truncated to 8 digits.

Consider the sample program shown in Figure 2, which computes and saves the final result in variable res on line 24. In this example, we consider three precisions: fl (float), db (double) and db8 (accurate up to 8 digits compared to the double precision value). More specifically, the value in precision db8 represents a value that agrees with the value obtained when double precision is used throughout the en-

![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 faster the analysis.](image)

![Figure 2: Sample Program](image)
Table 1: Statement $r_3 = r_2 - t_1$ is executed using different precisions for $r_2$ and $t_1$. The column $OP\ Prec$ shows the precisions used for the operands ($fl$ corresponds to float, $db$ to double, and $dib$ to double accurate up to 8 digits). Columns $r_2$ and $t_1$ show the values for the operands in the corresponding precisions. Column $r_3$ shows the result for the subtraction. Precision $(db, db)$ produces a result that satisfies the precision requirement $dib$.

<table>
<thead>
<tr>
<th>Op Prec</th>
<th>$r_2$</th>
<th>$t_1$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(fl, fl)$</td>
<td>6.8635854721</td>
<td>7.3635854721</td>
<td>-0.5000000000</td>
</tr>
<tr>
<td>$(fl, db)$</td>
<td>6.8635854721</td>
<td>7.3635856000</td>
<td>-0.5000000127</td>
</tr>
<tr>
<td>$(fl, db)$</td>
<td>6.8635854721</td>
<td>7.3635856800</td>
<td>-0.5000002079</td>
</tr>
<tr>
<td>$(db, fl)$</td>
<td>6.8635856000</td>
<td>7.3635854721</td>
<td>-0.4999998721</td>
</tr>
<tr>
<td>$(db, db)$</td>
<td>6.8635856000</td>
<td>7.3635856000</td>
<td>-0.5000000000</td>
</tr>
</tbody>
</table>

... (omitted) ...

| $(db, db)$ | 6.8635856913 | 7.3635856800 | -0.4999999887 |

tire 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 $dib$. According to the IEEE 754-2008 standard, the binary representation of $v$ has 52 explicitly stored bits in the significand. We first find the number of bits that corresponds to 8 significant decimal digits. The number of bits can be computed as $lg(10^8) = 26.57$ bits. We therefore keep 27 bits, and set the remaining bits in the significant to 0 to obtain the value $v_8$.

Back to our example, when the final result is computed in double precision, the result is $res = 0.5000000013$. When the computation is performed solely in single precision (all variables in the program are declared as float instead of double), the result is $res = 0.4999998027$. Assuming that we require the result to have precision $dib$, the result would be $res = 0.500000001xy$, where $x$ and $y$ can be any decimal digits. Precision tuning is based on the precision requirement set for the final result(s) of the program.

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. For example, let us consider the statement on line 23: $r_3 = r_2 - t_1$, and precision $dib$ for its result. Since the double value of $r_3$ is $-0.4999999887$, this means that we require $r_3$ to be $-0.49999998$ (i.e., the value matches to 8 significant digits). In order to determine the precision requirement for the two operands ($r_2$ and $t_1$), 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, $(fl, db)$ means that $r_2$ has fl precision, and $t_1$ has $dib$ 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 $(db, db)$ when the result is required to have precision $dib$. Similarly, operand precision requirements will also be recorded when the result is required to have other precisions under consideration ($fl$, and $db$ in this example).

The precision requirement $dib$ for the result of this operation, the first time this statement is executed the analysis records the double values for the operands and the result (6.0000000000, 1.8408964200, 11.045375200). The algorithm tries different precision combinations for the operands, and determines that precision $(fl, db)$ suffices. The second time the statement is executed, the analysis records new double values (11.045375200, 1.8408964200, 20.333333750). After trying all precision combinations for the operands, it is determined that this time the precision required is $(db, db)$, 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 $(fl, db)$ and $(db, db)$ would result in the precision requirement $(db, db)$.

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, and considering the precision requirement for the final result. The pass finds the program dependencies and required precisions, 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 $dib$, 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, along with the corresponding precision requirements. The analysis then collects the variables that can be allocated in single precision. In this example, only variable $factor$ in function $mpow$ can be allocated in 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 analysis heuristics and optimizations (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 expressions. 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. Each shadow value carries two values corresponding to the
In this example, $x, y, z$ be the set of all memory addresses used by the program, with $x$ of variables and $y$, $z$ respectively. Precisions $P$ be the set of all precisions, i.e., $P = \{\text{single, double}\}$.

3.3 Building the Blame Sets

In this section, we formally describe our 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., $P = \{\text{single, double}\}$. Precisions $\{f, \text{db}\}$ stand for single and double precisions, respectively. Precisions $\{\text{db}, \text{db6, db8, db10}\}$ denote values 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: $f < \text{db6} < \text{db8} < \text{db10} < \text{db}$. 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., $B : \mathcal{L} \times P \rightarrow \mathcal{L} \times P$, where $\mathcal{L}$ denotes the power set of $L$. If $B(\ell, p) = \{(\ell_1, p_1), (\ell_2, p_2)\}$, then it means that during an execution if instruction labeled $\ell$ produces a value that is accurate up to precision $p$, then instructions labeled $\ell_1$ and $\ell_2$ must produce values that are accurate up to precision $p_1$ and $p_2$, respectively.

The blame map $B$ is updated 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 a program run consider a state where this instruction is 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 $\ell : x = f(y_1, \ldots, y_n)$ 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 the instructions that define the values of its operands. Figure 5 shows the pseudocode of the function BlameSet. The function first computes the accurate result by retrieving the shadow value corresponding to the input instruction, and truncating the shadow value to precision $p$ (line 1). Function trunc_shadow($s, p$) returns the floating-point value corresponding to the precision $p$ given the shadow value $s$. Specifically, if $p$ is single precision, then the single value of $s$ is returned, otherwise trunc_shadow returns the double value of $s$ truncated to $p$. The shadow values corresponding to all operand variables are retrieved on line 2. Then, the procedure finds the minimal precisions $p_1, \ldots, p_n$ such that if we apply $f$ on $s_1, \ldots, s_n$ truncated to precisions $p_1, \ldots, p_n$, respectively, then the result truncated to precision $p$ is equal to the accurate result computed on line 1. Function trunc($x, p$) returns $x$ truncated to precision $p$. We then pair each $p$, with $LM[\text{key}^i]$, 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

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

**Procedure FAddShadow**

**Inputs**

$\ell : x = y + z : \text{instruction}$

**Outputs**

Updates the shadow memory $M$ and the label map $LM$

**Method**

1. $\{\text{single, double}\} = M[\&y]$
2. $\{\text{single, double}\} = M[\&z]$
3. $M[\&x] = \{\text{single, double}\}$
4. $LM[\&x] = \ell$

Figure 4: Shadow Execution of fadd Instruction
If \((\ell, p_1), (\ell, p_2), \ldots, (\ell, p_n)\) are all the pairs involving the label \(\ell\) present in \(LP_1\) or \(LP_2\), then \((\ell, \text{max}(p_1, p_2, \ldots, p_n))\) is the only pair involving \(\ell\) present in \((LP_1 \cup LP_2)\).

Given the functions \(\text{BlameSet}\) and merge \(\sqcup\), we compute \(B(\ell, p) \sqcup \text{BlameSet}(\ell : x = f(y_1, \ldots, y_n), p)\) 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 result computed by a given instruction labeled \(\ell_{out}\) is accurate up to precision \(p\). 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 \(\ell_{out}\) is \(p\). We compute this using the function \(\text{Accuracy}(\ell_{out}, p, B)\) which returns a set of pairs instruction labels and precisions, such that if \((\ell', p')\) is present in \(\text{Accuracy}(\ell_{out}, p, B)\), then the result of executing the instruction labeled \(\ell'\) must have a precision of at least \(p'\). \(\text{Accuracy}(\ell, p, B)\) can then be defined recursively as follows.

\[
\text{Accuracy}(\ell, p, B) = \{(\ell, p)\} \sqcup \bigcup_{(\ell', p') \in B(\ell, p)} \text{Accuracy}(\ell', p', B)
\]

After computing \(\text{Accuracy}(\ell_{out}, p, B)\), we know that if \((\ell', p')\) is present in \(\text{Accuracy}(\ell_{out}, p, B)\), then the instruction labeled \(\ell'\) must be executed with precision at least \(p'\) for the result of executing instruction \(\ell_{out}\) to have a precision \(p\).

### 3.4 Heuristics and Optimizations

To attain scalability for large or long running programs, the implementation of \(\text{Blame Analysis}\) must address memory usage and running time. We have experimented with both online and offline versions of our algorithm.

The offline \(\text{Blame Analysis}\) first collects the complete execution trace, and then builds the blame set for each dynamic instruction (i.e., if a static instruction is executed more than once, a blame set will be computed for each time the instruction was executed). As each instruction is examined only once, merging operand precisions is not required. Thus, when compared to online \(\text{Blame Analysis}\), the offline approach often produces better (lower precision) solutions. However, the size of the execution trace, and the blame set information explode for long running programs. For example, when running offline \(\text{Blame Analysis}\) on the ep NAS benchmark with input class\(^2\) S, the analysis terminates with an out of memory error, exhausting 256 GB of RAM.

The online \(\text{Blame Analysis}\) 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 for ep. On the other hand, the blame sets for each instruction have to be merged across all its dynamic invocations, making the analysis slower. In our implementation, we allow developers to specify what part of the program they are interested to analyze. For short running programs, such as functions within the GSL 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, \(\text{Blame Analysis}\) by default will be performed throughout the entire program execution.

In summary, our results show that offline \(\text{Blame Analysis}\) is fast (no merge operations) and produces better solutions for small programs, but it is expensive in terms of memory usage, which makes it impractical. In contrast, online \(\text{Blame Analysis}\) is memory efficient, produces good solutions, and it is not too expensive in terms of running time, thus it has the potential to perform better when analyzing larger programs. For brevity, the results reported in the rest of this paper are obtained using the online analysis.

### 4. EXPERIMENTAL EVALUATION

The \(\text{Blame Analysis}\) architecture is described in Figure 6. We build the analysis on top of the LLVM compiler infrastructure [27]. The analysis takes as input: (1) LLVM bitcode of the program under analysis, (2) a set of test inputs, and (3) analysis parameters that include the target instruction and the desired error threshold(s). Because the analysis is implemented using LLVM, it can be applied to programs written in languages that have a LLVM compiler frontend (e.g., C, C++, and Fortran). We use the original Precimonious benchmarks (written in C), which have been modified by experts to provide acceptability criteria for the result precision. For \(\text{Blame Analysis}\) we select the acceptability code developed for Precimonious as the target instruction set. Thus, the results provided by both analyses always satisfy the programmer 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 \(\text{Blame Analysis}\) by itself, as well as when used as a pre-processing stage for Precimonious. We refer to the latter as \(\text{Blame + Precimonious}\). 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. We validate all the results presented in this section by manually modifying the programs according to the type assignments suggested by the tools, and running them to verify that the corresponding final results are as accurate as required for all test inputs.

### 4.1 Experiment Setup

We present results for eight programs from the GSL library [17] and two programs from the NAS parallel benchmarks [37]. We use clang with no optimizations\(^3\) and a

---

\(^2\)Class S is a small input, designed for serial execution.

\(^3\)Optimizations sometimes remove floating-point variables, which causes the set of variables at the LLVM bitcode level to differ from the variables at the source code level.
Python wrapper [34] to build whole program (or whole library) LLVM bitcode. Note that we do apply optimization level -02 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 256 GB RAM.

We use the procedure described in [35] to select program inputs. For the NAS benchmarks (programs ep and cg), we use the provided input Class A. For the rest, 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. We are indeed using the same set of inputs used in the original evaluation of Precimonious.

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 Performance

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.

By itself, BLAME ANALYSIS introduces up to $50\times$ slowdown, which is comparable to the runtime overhead reported by widely-used instrumentation based tools such as Valgrind [30] and Jalangi [39]. Table 2 shows the overhead for programs cg and ep. For the rest of our benchmarks, the overhead is relatively negligible (less than one second).

To measure the analysis time of the combined analyses, we add the analysis time of BLAME ANALYSIS and the search 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 all benchmarks, except for program ep. The original version of this program uses error threshold $10^{-8}$, thus we do not consider error threshold $10^{-10}$.

Overall, we find that BLAME + Precimonious is faster than Precimonious in 31 out of 39 experiments (4 error thresholds for 9 programs, and 3 error thresholds for 1 program). 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 4), 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 for programs ep ($10^{-4}$ and $10^{-6}$), and cg ($10^{-8}$). Figure 8 shows the analysis time breakdown.

Table 3 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 4). 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%.

In terms of memory usage, the online version of BLAME ANALYSIS uses up to 81 MB of memory in our experiments. 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 4 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, BLAME ANALYSIS successfully identifies at least one variable as float. If we consider all 39 experiments, BLAME ANALYSIS removes from the search space 40% of the variables on average, with a median of 28%.

The type configurations proposed by BLAME + Precimonious and Precimonious agree in 28 out of 39 experiments, and differ in 11 experiments. Table 5 shows the

<table>
<thead>
<tr>
<th>Program</th>
<th>Execution (sec)</th>
<th>Analysis (sec)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>cg</td>
<td>3.52</td>
<td>185.45</td>
<td>52.55×</td>
</tr>
<tr>
<td>ep</td>
<td>34.70</td>
<td>1699.74</td>
<td>48.98×</td>
</tr>
</tbody>
</table>

Table 2: Overhead of BLAME ANALYSIS

<table>
<thead>
<tr>
<th>Program</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>bessel</td>
<td>22.48×</td>
</tr>
<tr>
<td>gaussian</td>
<td>1.45×</td>
</tr>
<tr>
<td>roots</td>
<td>18.32×</td>
</tr>
<tr>
<td>polyroots</td>
<td>1.54×</td>
</tr>
<tr>
<td>rootnewt</td>
<td>38.42×</td>
</tr>
</tbody>
</table>

Table 3: Average analysis time speedup of BLAME + Precimonious compared to Precimonious alone

![Figure 8: Analysis time breakdown for BLAME + Precimonious (B+P) and Precimonious (P) for two NAS benchmark programs](image)

In terms of memory usage, the online version of BLAME ANALYSIS uses up to 81 MB of memory in our experiments. 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).
Figure 7: Analysis time comparison between Precimonious (P) and Blame + Precimonious (B+P). The vertical axis shows the analysis time in seconds. The horizontal axis shows the error thresholds used in each experiment. In these graphs, a lower curve means the analysis is more efficient.

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

Blame Analysis has several limitations. First, similar to other state-of-the-art tools for precision tuning, our analysis cannot guarantee accurate outputs for all possible in-
puts, thus there is the need for representative test inputs. Although input generation has a significant impact on the type configurations recommended by our analysis, the problem of generating floating-point inputs is orthogonal to the problem addressed in this paper. In practice, we expect programmers will be able to provide meaningful inputs, or use complementary input-generation tools [16]. Still, we believe our tool is a powerful resource for the programmer, who will ultimately decide whether to apply the suggested configurations fully or partially.

Another limitation is that Blame Analysis does not take into account program performance; by itself, the suggested configurations might not lead to program speedup. Note that, in general, lowering precision does not necessarily result in a faster program. For example, consider the addition $v_1 + v_2$. Assume $v_1$ has type float and $v_2$ has type double. The addition will be performed in double precision, requiring to cast $v_1$ to double. When a large number of such casts is required, the tuned program might be slower than the original program. The analysis focuses on the impact in accuracy, but does not consider the impact in running time. Because of this, the solutions produced are not guaranteed to improve performance.

Last, the program transformations suggested by Blame Analysis are limited to changing variable declarations whose precision will remain the same throughout the execution of the program. We do not currently handle shift of precision during program execution, which could potentially contribute to improving program performance. Also, the analysis does not consider algorithmic changes that could also potentially improve running time. Note that both kinds of transformations would require additional efforts to express program changes.

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 efficiency of Blame Analysis as a standalone tool, as well as a filter for Precimonious.

A future direction is to use Blame Analysis as an intra-procedural analysis, rather than an interprocedural 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. Another future direction is to experiment with other intermediate precisions. In this paper, we used four intermediate precisions, db4, db8, db16, and db32, 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, further pruning the search space of Precimonious.

6. RELATED WORK

Precimonious [35] is a dynamic analysis tool for tuning floating-point precision, already detailed. Lam et al. [25] also propose a framework for finding mixed-precision floating-point computation. 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 DoubleCuts and Lam’s framework in that it performs a white-box analysis on the
Table 5: Speedup observed after precision tuning using configurations produced by Blame + Precimonious (B+P) and Precimonious alone (P)

<table>
<thead>
<tr>
<th>Program</th>
<th>Threshold $10^{-4}$</th>
<th>Threshold $10^{-6}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B+P</td>
<td>P</td>
</tr>
<tr>
<td>bessel</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>gaussian</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>roots</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>polyroots</td>
<td>0.4%</td>
<td>0.0%</td>
</tr>
<tr>
<td>rootnewt</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>sum</td>
<td>39.9%</td>
<td>39.9%</td>
</tr>
<tr>
<td>fft</td>
<td>8.3%</td>
<td>8.3%</td>
</tr>
<tr>
<td>blas</td>
<td>5.1%</td>
<td>5.1%</td>
</tr>
<tr>
<td>ep</td>
<td>0.6%</td>
<td>0.0%</td>
</tr>
<tr>
<td>cg</td>
<td>7.7%</td>
<td>7.7%</td>
</tr>
</tbody>
</table>

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 our analysis is to minimize the use of double precision in the program without considering performance.

Darulova et. al [19] 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 [38] 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 [11] 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 [18] 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 [20] 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 [8, 6, 7, 21, 26, 41, 46]. Benz et al. [8] 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 [1] is another tool that computes floating-point errors to detect accuracy problems. It computes a shadow value side-by-side, but it stores an absolute error in double precision instead. Herbie [31] estimates and localizes rounding errors, and then rewrites numerical expressions to improve accuracy. The above tools aim to find accuracy problems (and improve accuracy), not to find opportunities to reduce floating-point precision.

Other large areas of research that focus on improving performance are autotuning (e.g., [9, 22, 33, 42, 43]) and approximate computing (e.g., [10, 15, 29, 36, 40]). 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 [39, 30, 32, 12]. 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. The code is open source and available online.

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 the analysis as a filter to prune the inputs to Precimonious, a floating-point tuning tool that searches through the variable space. The combined analysis time is 9× faster on average, and up to 38× in comparison to Precimonious alone. The resulting type configurations 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.

8. ACKNOWLEDGMENTS

Support for this work was provided through the X-Stack program funded by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under collaborative agreement numbers DE-SC0008099 and DE-SC0010200. This work was also partially funded by DARPA award number HR0011-12-2-0016, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Hewlett-Packard, Huawei, LGE, NVIDIA, Oracle, and Samsung.

---

4https://github.com/corvette-berkeley/shadow-execution
9. REFERENCES


