Precimonious: Tuning Assistant for Floating-Point Precision

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

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

ABSTRACT

Given the variety of numerical errors that can occur, floating-point programs are difficult to write, test and debug. One common practice employed by developers without an advanced background in numerical analysis is using the highest available precision. While more robust, this can degrade program performance significantly. In this paper we present Precimonious, a dynamic program analysis tool to assist developers in tuning the precision of floating-point programs. Precimonious performs a search on the types of the floating-point program variables trying to lower their precision subject to accuracy constraints and performance goals. Our tool recommends a type instantiation that uses lower precision while producing an accurate enough answer without causing exceptions. We evaluate Precimonious on several widely used functions from the GNU Scientific Library, two NAS Parallel Benchmarks, and three other numerical programs. For most of the programs analyzed, Precimonious reduces precision, which results in performance improvements as high as 41%.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; G.1.6 [**Numerical Analysis**]: Optimization

General Terms

Algorithms, Languages, Performance

Keywords

floating-point arithmetic, mixed precision, program optimization, delta-debugging algorithm, dynamic program analysis

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SC'13, November 17–21 2013, Denver, Colorado, USA.

ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503296.

1. INTRODUCTION

Floating-point arithmetic [15, 26] is used in applications from a wide variety of domains such as high-performance computing, graphics and finance. To minimize the chance of problems, developers without an extensive background in numerical analysis are likely to use the highest available precision throughout the whole program. Even experts may find it difficult to understand how sensitive the execution is to rounding error and consequently default to using the highest precision available. While more robust, this can increase the program execution time, memory traffic, and energy consumption. Ideally, a programmer would use no more precision than needed in any part of an application.

To illustrate the problem, consider the experiment performed by Bailey [3] with a program that computes the arc length of an irregular function, written in `Fortran`. An implementation using `double` precision computes a result whose error is about $2 \cdot 10^{-13}$, compared to a second more accurate implementation using `double double` precision, but is about 20x faster[1]. On the other hand, if `double double` precision is used only for *one* of the variables in the program, then the result computed is as accurate as if `double double` precision had been used throughout the whole computation while being almost as fast as the all-double implementation. Aside from the Bailey experiment [3], many other efforts [2, 9, 20, 22] have shown that programs implemented in mixed precision can sometimes compute a result of the same accuracy faster than when using solely the highest precision arithmetic.

Given the complexity of existing numerical programs, it might be prohibitively expensive or downright impossible to manually tune their floating-point precision. In this paper, we present a tool called Precimonious (short for "parsimonious with precision") to help automate this process. Our tool has been implemented using the LLVM [21] compiler infrastructure and it recommends a type assignment for the floating-point variables in the program so that the resulting program uses lower floating-point precision while producing results that satisfy accuracy and performance constraints.

Precimonious exposes to application developers an interface to specify the accuracy acceptance criteria. It takes as input a program annotated with the developer's accuracy

[1] The author performed the experiment on an Intel-based Macintosh system, using the `gfortran` compiler and the QD package [16] to implement the `double double` type (approximately 31 digits).

expectation and it implements a dynamic program analysis to find a program variant that uses lower precision, subject to performance constraints. Given the set of program floating-point variables and their possible types, the analysis performs a search trying to reduce the precision while satisfying the constraints. To guarantee the performance constraints, the analysis uses either static performance models or dynamic instrumentation with hardware performance counters. The latter allows for easy extensibility to performance metrics such as energy or memory traffic. The search is based on the *delta-debugging* algorithm [29], which exhibits an $O(n \log n)$ average complexity and an $O(n^2)$ worst-case complexity, where n is the number of variables to be tuned. Precimonious also requires a representative set of program inputs. No guarantees are made for other inputs.

We evaluate Precimonious on eight widely used functions from the GNU Scientific Library (GSL) [10], two NAS Parallel Benchmarks [4], and three other numerical programs, two of which are described in [3]. We use running time as our performance metric. For most of the analyzed programs, Precimonious can find lower-precision type configurations when varying the degree of accuracy desired. A *type configuration* maps each floating-point program variable to a floating-point type suggested by our tool. When our tool does not find a type configuration, it is because no existing configuration leads to a speedup. By using the precision suggested by our tool, which is found automatically, we observe speedups as high as 41% when running the programs transformed according to the guidance of Precimonious. The main contributions of this paper are:

- We introduce a novel automated approach for recommending a type configuration for program variables that leads to performance improvement in floating-point programs. The resulting program uses lower precision and has better performance than the original program.

- We implement our algorithm in a tool called Precimonious and demonstrate its effectiveness by evaluating our tool on several numerical programs. Precimonious, and all the data and results presented in this paper, are publicly available under BSD license [2].

The rest of this paper is organized as follows. In Section 2, we present a motivating example and describe challenges in precision tuning. We then provide a detailed description of our approach in Section 3. We give implementation details and present our experimental evaluation in Section 4. The limitations of our approach are discussed in Section 5. Finally, we discuss related work in Section 6 before concluding in Section 7.

2. OVERVIEW

A motivating example to show the effect of floating-point precision on program performance has been introduced by Bailey in [3]. The task is to estimate the arc length of the following function over the interval $(0, \pi)$.

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

The corresponding program sums $\sqrt{h^2 + (g(x_k + h) - g(x_k))^2}$ for $x_k \in [0, \pi)$ divided into n subintervals, where $n = 1000000$, so that $h = \frac{\pi}{n}$ and $x_k = kh$. Allowing k and n to go to ∞ makes this computation much harder [18].

A straightforward implementation in `C` using `long double` precision is shown in Figure 1(a). When compiled with `gcc` and run on an Intel x86 system[3] this program produces the answer 5.795776322412856 (stored in variable `s1` on line 27).

If the program uses `double` precision instead, the resulting value would be 5.79577632241**311**, which is only correct up to 11 or 12 digits after the decimal point (compared to the result produced by the `long double` precision program).

Figure 1(b) shows an optimized implementation written in `C` by an expert numerical analyst. From the original eight floating-point variables appearing on the highlighted lines, the optimized program uses `long double` precision only for the variable `s1` on line 17. Six other variables have been lowered to `double` precision and one variable to `float`. The program produces the correct answer and runs 10% faster than the `long double` precision version. When replacing `long double` with even higher `double double` precision from the QD package [16], the same transformation remains valid [3] and the performance improvements can be as high as 20×. We note that `double double` arithmetic is considerably slower than `long double` arithmetic because it is done in software, and each operation requires a sequence of roughly 20 instructions.

For any program, manually finding such an optimized type configuration requires domain-specific knowledge combined with advanced numerical analysis expertise. While probably feasible for small programs, manual changes are prohibitive for large code bases.

Our proposed tool automates precision tuning of floating-point programs. We reduce the problem of precision tuning to determining which program variables, if any, can have their types changed to a lower precision while producing an answer within a given error threshold. We say such an answer is *accurate enough*. In addition to satisfying accuracy requirements, the changes also have to satisfy performance constraints, e.g., the transformed program should be at least as fast as the original.

The main challenge for any automated approach is devising an efficient search strategy through the problem space. In our case this is the set of program variables and their type assignments. For the code example from Figure 1(a), there are eight floating-point variables and we consider three different floating-point precisions. The search space contains 3^8 possible combinations.

2.1 Searching for Type Configurations

For a given program, there may exist one or more *valid* type configurations that (1) use less floating-point precision than the type configuration used by the original program, (2) produce an accurate enough answer without exceptions, and (3) result in a program with better performance than the original program.

Different search algorithms can be considered depending on what valid configuration we are interested in finding. A *global* minimum corresponds to the type configuration that

[2] `https://github.com/corvette-berkeley/precimonious`

[3] All floating-point types used in this paper conform to the IEEE 754-2008 standard as implemented on various x86 architectures. In particular, `long double` is implemented as the 80-bit extended precision type with 64 significant bits.

```c
long double fun( long double x ) {
  int k, n = 5;
  long double t1;
  long double d1 = 1.0L;

  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;
    t1 = t1 + sin (d1 * x) / d1;
  }
  return t1;
}

int main( int argc, char **argv) {
  int i, n = 1000000;
  long double h, t1, t2, dppi;
  long double s1;

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer is stored in variable s1
  return 0;
}
```

(a) Program in `long double` precision

```c
double fun( double x ) {
  int k, n = 5;
  double t1;
  float d1 = 1.0f;

  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;
    t1 = t1 + sin (d1 * x) / d1;
  }
  return t1;
}

int main( int argc, char **argv) {
  int i, n = 1000000;
  double h, t1, t2, dppi;
  long double s1;

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer is stored in variable s1
  return 0;
}
```

(b) Tuned program using mixed precision

Figure 1: Two implementations of the `arclength` program using different type configurations. The programs differ on the precision of all floating-point variables except for variable `s1`.

uses the least precision that results in the best performance improvement among all valid type configurations for a given program. A *local* minimum (called *1-minimal* below) is a valid type configuration for which lowering the precision of any one additional variable would cause the program to compute an insufficiently precise answer or violate the performance constraint (e.g., being slower than the original program). We define a *change set*, denoted by Δ, to be a set of variables initialized in higher precision, thought to be ineligible for their precision to be lowered. A change set Δ applied to the program under consideration results in a program variant where only the variables in Δ are in higher precision.

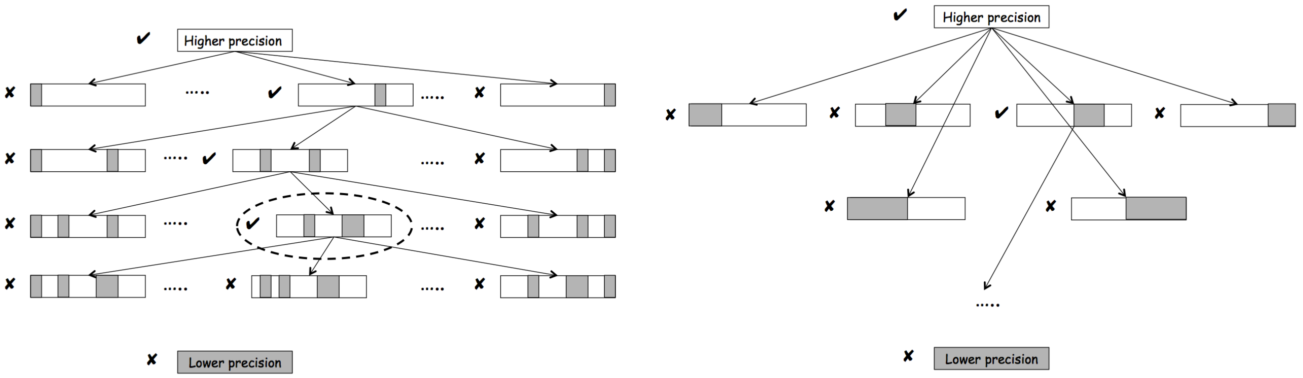
Finding a Global Minimum: Finding the global minimum may require evaluation of an exponential number of type configurations. To be precise, we may be required to evaluate P^n configurations, where n is the number of floating-point variables in the change set and P is the number of levels of floating-point precision. The naïve approach which evaluates all possible type configurations by changing one variable at a time is infeasible for large code bases.

Finding a Local Minimum: If we are interested in finding a 1-minimal valid configuration, a naïve approach would consist of removing from the change set, say Δ, one element at a time, where Δ initially consists of all floating-point variables in the program. This means the element removed can be in a lower precision, while all other elements are in the higher precision. If any of these change sets translates to a program that passes the accuracy and performance test, we recur with this smaller set. Otherwise Δ is 1-minimal, and we can stop the search.

This algorithm is illustrated via Figure 2(a). Each rectangular box represents a change set under test. A gray band denotes the set of variables that are removed from the change set (thus can be allocated in lower precision). A cross means the change set results in an invalid configuration (the program either produces an insufficiently precise answer or violates the performance constraint), while a check means the change set results in a valid configuration. In this figure, the algorithm finds a 1-minimal valid configuration after 4 iterations (the one in the dotted oval). Notice that lowering the precision of any additional variable in this configuration would make it invalid. This is illustrated in the fourth iteration - all configurations created are invalid.

Delta-Debugging: The delta-debugging search algorithm [29] finds a 1-minimal test case with the average running time of $O(n \log n)$. The worst case running time is still $O(n^2)$ and it arises when each iteration results in the reduction of the change set by one element, which reduces to using a naïve algorithm.

Informally, instead of making one type change at a time, the algorithm divides the change set in two and increases the number of subsets if progress is not possible. At a high level, the delta-debugging algorithm consists of partioning the change set Δ into almost equal size subsets $\triangle_1, \triangle_2, \cdots \triangle_n$, which are pairwise disjoint. The complement of a delta $\triangle_i$ is defined as $\bigtriangledown_i = \Delta - \triangle_i$. The algorithm starts by partitioning Δ into two sets ($n = 2$). After partitioning, the algorithm proceeds to examine each change set in the partition and their corresponding complements. The change set is reduced if a smaller failure inducing set is found, otherwise the partition is refined with $n = n * 2$.



(a) Quadratic number of variants

(b) Binary search on quadratic search space

Figure 2: Examples of search strategies

This algorithm is illustrated via Figure 2(b). The original change set (higher precision rectangular box) is divided into two equal or almost equal size subsets. These two subsets result in two invalid configurations, so the partition granularity is increased to four. The original change set is divided into four subsets accordingly, in which one of the subsets results in a valid configuration. The algorithm then recurses on this smaller subset.

3. PRECISION TUNING

We discuss the realization of our technique as a practical and effective tool for tuning the precision of floating-point programs, called Precimonious. Figure 3 depicts its system architecture, which is built using the LLVM compiler infrastructure [21]. Precimonious receives a C program, a test suite, and an accuracy requirement as input. It outputs a type configuration that, when applied to the original program, will result in a faster program that still produces an accurate enough answer without throwing exceptions.

Precimonious consists of four main components: creating the search space for the program under analysis (Section 3.1), creating candidate type configurations using our search algorithm (Section 3.2), generating program variants that reflect the candidate type configurations (Section 3.3), and identifying valid type configurations (Section 3.4).

3.1 Creating the Search Space

We start by creating a *search file*, which consists of all variables whose precisions need to be tuned. The search file associates each floating-point variable with the set of floating-point types to be explored (e.g., `float, double`, and `long double`). The input to this process is the program under analysis in the format of `LLVM` bitcode. An excerpt of the search file for the `arclength` program from Section 2 is given below. The configuration entry states that the local variable `t1` in function `fun` can be of type `float`, `double` or `long double` (for ease of presentation, we assume local variables in the same function have unique names).

```
localVar: {
  function: fun
  name: t1
  type: [float, double, long double]
}
```

In the current implementation, the search file includes the local variables of all functions statically reachable from `main`. We include only those global variables that are accessed by these functions. We include both scalars and arrays. Optionally, the search file can also include entries associated with function calls. If the developer is aware of any called functions for which other implementations are available, these can be included in the search space as well. For example, the math library provides double and single precision implementations of many functions such as the `log` function (`logf` for single precision). The single precision implementation for many of these functions are often more efficient. If provided, Precimonious can also consider switching function calls in addition to floating-point variable types.

3.2 Search Algorithm

We devise a modified version of the delta-debugging algorithm [29] to find a type assignment for floating-point variables so that the resulting program uses less precision while producing results that satisty both accuracy and performance constraints.

Given a configuration search file, our goal is to find a type configuration that maps each variable to a type and that exhibits lower cost (performance improvement). Initially, each variable in the search file is associated with a set of types. Our algorithm iteratively refines each of these sets of types until it consists of only one type. In each iteration, the algorithm considers a pair of types, the highest and second-highest precision available. It then determines the set of variables that need to be allocated in the highest precision. For these variables, the corresponding set of types are refined to contain only the highest precision type. These variables are then ignored in later iterations. For those variables that can be in the second-highest precision, the highest precision can be removed from their set of types in the next iteration.

Because our algorithm is based on the delta-debugging search algorithm, with heuristic pruning, it is efficient in practice. To balance between the searching cost and the quality of the selected configuration, we choose to search for a local minimum configuration instead of a global minimum. As shown in Section 4, our algorithm finds type configurations for most of the analyzed programs (for the workloads under consideration), leading to perfomance speedup.

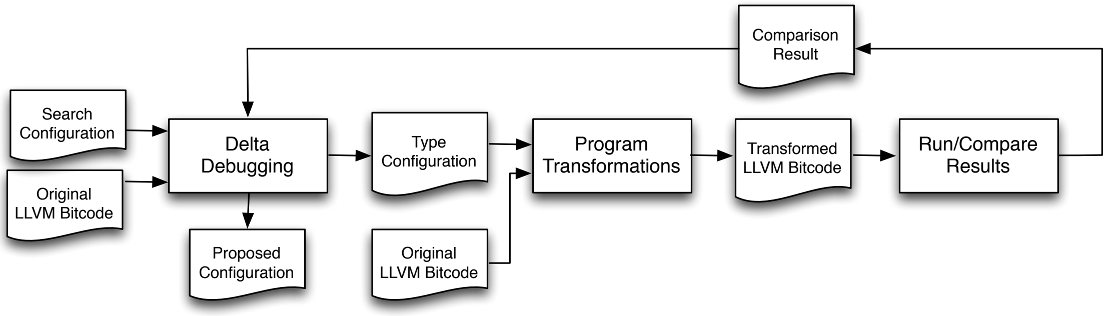


Figure 3: High-level framework components

Figure 4 shows our lower cost configuration search algorithm, entitled LCCSearch. In this algorithm, a *change set* is a set of variables. The variables in the change set must have higher precision. The algorithm outputs a minimal change set, which consists of a set of variables that must be allocated in the higher precision (all other variables of interest can be in lower precision) so that the transformed program produces an accurate enough result and satisfies the performance goal.

Procedure LCCSearch
Inputs
P : target program
Δ : change set

Outputs
A minimal change set

Algorithm

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

Figure 4: Lower Cost Configuration Search Algorithm

The algorithm starts by dividing the change set Δ into two subsets of equal or almost equal size Δ_1 and Δ_2. It also creates the complement set of these subsets $\nabla_1 = \Delta \setminus \Delta_1$ and $\nabla_2 = \Delta \setminus \Delta_2$ (lines 4-5). For each of these subsets, the algorithm creates the corresponding program variant (Section 3.3), checks whether the program variant produces an accurate enough result and is faster (Section 3.4), and records the one that has the lowest cost (lines 6-11). The function `accurate(P, Δ)` transforms the program `P` according to Δ and returns a boolean value indicating whether the transformed program is accurate enough. Similarly, the function `cost(P, Δ)` transforms the program `P` according to Δ and returns the cost of the transformed program. If a change set with a lower cost exists, the algorithm recurses with that smaller change set (lines 12-13 and 19); otherwise it restarts the algorithm with a finer-grained partition (lines 17-19). In the special case where the granularity can no longer be increased, the algorithm returns the current Δ, which is a local minimum type configuration (lines 15-16).

3.3 Generating Program Variants

We automatically generate program variants that reflect the type configurations produced by our LCCSearch algorithm. We compile the original C program[4] to `LLVM` intermediate representation (`LLVM IR`) and apply a set of program transformations to reflect a given set of variable type assignments. We support scalar and array types. Supporting structures is part of the future work. The result of the transformation is a binary file (`LLVM` bitcode file) of the program whose variable types have been changed accordingly. The following subsections describe our program transformations given our intermediate representation of choice. Note that most `LLVM` instructions, including all arithmetic and logical operations, are in three-address form. An instruction in three-address form takes one or two operands, and produces a single result. Thus, our transformations are directly applicable to any other three-address form intermediate representation.

3.3.1 LLVM Intermediate Representation

`LLVM` (Low Level Virtual Machine) is a compiler infrastructure for program analysis and transformation. `LLVM` supports a language-independent instruction set and type system. `LLVM` has a static single assignment (SSA) based intermediate representation, meaning that each variable (called a

[4] Because we are using the `LLVM IR`, we can also transform programs written in other languages for which an `LLVM` frontend is available (e.g., `Fortran`). However so far we have only transformed C programs.

```
define x86_fp80 @fun(x86_fp80) nounwind uwtable ssp {
  // allocating space for local variables
  %x = alloca x86_fp80, align 16
  %t1 = alloca x86_fp80, align 16
  %d1 = alloca x86_fp80, align 16
  // instructions to store initial values
  ...
  // t1 = x; (line 6)
  %2 = load x86_fp80* %x, align 16
  store x86_fp80 %2, x86_fp80* %t1, align 16
  ...
  // t1 = t1 + sin(d1 * x) / d1; (line 9)
  %10 = load x86_fp80* %t1, align 16
  %11 = load x86_fp80* %d1, align 16
  %12 = load x86_fp80* %x, align 16
  %13 = fmul x86_fp80 %11, %12
  %14 = fptrunc x86_fp80 %13 to double
  %15 = call double @sin(double %14) nounwind readnone
  %16 = fpext double %15 to x86_fp80
  %17 = load x86_fp80* %d1, align 16
  %18 = fdiv x86_fp80 %16, %17
  %19 = fadd x86_fp80 %10, %18
  store x86_fp80 %19, x86_fp80* %t1, align 16
  ...
  // return t1; (line 11)
  %24 = load x86_fp80* %t1, align 16
  ret x86_fp80 %24
}
```

(a) LLVM bitcode for original function

```
define x86_fp80 @fun(x86_fp80) nounwind uwtable ssp {
  // allocating space for local variables
  %x = alloca x86_fp80, align 16
  %t1 = alloca double, align 8
  %d1 = alloca x86_fp80, align 16
  // instructions to store initial values
  ...
  // t1 = x; (line 6)
  %2 = load x86_fp80* %x, align 16
  %3 = fptrunc x86_fp80 %2 to double
  store double %3, double* %t1, align 8
  ...
  // t1 = t1 + sin(d1 * x) / d1; (line 9)
  %11 = load double* %t1, align 8
  %12 = fpext double %11 to x86_fp80
  %13 = load x86_fp80* %d1, align 16
  %14 = load x86_fp80* %x, align 16
  %15 = fmul x86_fp80 %13, %14
  %16 = fptrunc x86_fp80 %15 to double
  %17 = call double @sin(double %16) nounwind readnone
  %18 = fpext double %17 to x86_fp80
  %19 = load x86_fp80* %d1, align 16
  %20 = fdiv x86_fp80 %18, %19
  %21 = fadd x86_fp80 %12, %20
  %22 = fptrunc x86_fp80 %21 to double
  store double %22, double* %t1, align 8
  ...
  // return t1; (line 11)
  %27 = load double* %t1, align 8
  %28 = fpext double %27 to x86_fp80
  ret x86_fp80 %28
}
```

(b) Transformed LLVM bitcode

Figure 5: LLVM bitcode excerpts of function `fun` from the `arclength` program of Figure 1

typed register) is assigned only once. Instructions can only produce *first-class* types. `LLVM` defines eight such types: integer, floating point, pointer, vector, structure, array, label, and metadata.

Figure 5(a) shows excerpts of the bitcode file produced for the function `fun` from the `arclength` program of Figure 1(a). These excerpts correspond to allocating space for local variables `x`, `t1`, and `d1`, and the instructions corresponding to the statements on lines 6, 9 and 11 (all statements involving variable `t1`). The instruction `alloca` allocates memory in the stack. Other instructions include `load` to read from memory, `store` to store in memory, `fpext` and `fptrunc` to cast floating-point numbers, and floating-point arithmetic operations such as `fadd`, `fsub`, `fmul` and `fdiv`. As mentioned earlier, each of these instructions is in three-address form, with no more than one operator on the right side of each assignment.

3.3.2 Program Transformations

We apply program transformations to change the type of variables according to the type configuration under consideration. We accomplish this by modifying the bitcode of the original program. For example, consider a type configuration that maps all floating-point variables to their original `long double` types except for variable `t1`, which is now suggested to be `double`. Figure 5(b) shows the transformed bitcode for the bitcode excerpts of Figure 5(a).

First, we replace `alloca` instructions to allocate the correct amount of memory for the variables whose type is to be changed. In this example, we allocate space for a `double` instead of a `long double` for variable `t1`. Second, we iterate through the uses of each of these instructions to identify other instructions that may need to be transformed. In this case, there are five instructions that use `t1` as an operand. These instructions are highlighted (in addition to the initial `alloca` instruction) in Figure 5(a). We define a set of program transformation rules to be applied depending on the instruction to be transformed. For each transformed instruction, we iterate through its uses to continue to propagate the changes.

For example, consider the transformation of the second highlighted `store` instruction in Figure 5(a). This instruction is storing the result of an addition (register `%19`) into `t1`. In this case, the source should be downcast before it can be stored in the new variable `t1`. Thus, the transformed bitcode, shown in Figure 5(b), includes an additional `fptrunc` instruction right before the new transformed `store` instruction. Note that in the case of arithmetic operation instructions, we run a final pass to make sure each operation is performed in the precision of its highest-precision operand. The resulting bitcode file verifies and standard compiler optimizations can be applied to it.

3.4 Checking Results and Performance

To determine whether a type configuration is valid, we optimize the transformed bitcode, compile it to native code, run it, and check for two criteria: correctness and performance.

First, we compare the result produced by the transformed program against the *expected result*. The expected result is the value (or values) obtained by running the original program on a given set of inputs. We take into account the error threshold provided by the programmer when comparing the results. Second, we measure running times for the original and transformed programs. We determine whether the transformed program is at least as fast as the original program. Note that the developer can specify the compiler to be used and the level of optimizations to be applied to both programs before measuring performance. In our experiments, we use the gcc and clang compilers with optimization level O2. Correctness and performance information is provided as feedback to our LCCSearch algorithm, which determines the new type configuration to be produced.

4. EXPERIMENTAL EVALUATION

As noted earlier, our implementation uses the LLVM compiler infrastructure [21]. We have written several LLVM passes in C++ to create search files, and to transform programs to have a given type configuration. We have implemented our LCCSearch algorithm in Python. Our result logger, result checker, and floating-point number generator have been implemented in C.

We present results for eight programs that use the GNU Scientific Library (GSL) [10], two NAS Parallel Benchmarks (SNU NPB Suite, C translation) [4], the arclength program described throughout the paper, a program that implements the Simpson's rule, and a program that calculates the infinite sum. We use Clang 3.0 to produce LLVM bitcode and a Python-based wrapper [24] to build whole-program (or whole-library) LLVM bitcode files. We run our experiments on an Intel Core i7-3770K 3.5Ghz Linux machine with 32GB RAM.

4.1 Experiment Setup

Precimonious can be used by developers with different levels of expertise in numerical analysis. In general, usage scenarios differ on the program inputs and the error thresholds selected for the analysis. A more experienced user might be more selective on the program inputs and the error thresholds to use. In the experiments presented in this section, we illustrate how a naïve user could employ our tool. We assume that the source code of the program is available, and the user knows which variable(s) will eventually store the result(s) produced by the program.

For the NAS Parallel Benchmarks (programs ep and cg), we use the provided input Class A. For the rest of the programs we assume no inputs are available, thus we generate 1000 random floating-point inputs, and classify these inputs based on code coverage. We construct a representative test input set by selecting one input from each resulting group, thus attempting to maximize code coverage. Note that this test input set can be replaced or augmented by an existing input set. We annotate each program to log and check the results produced. This usually requires adding a few function calls to our logger and checker routines.

In our experiments, we use four different error threshold values, which indicate the number of digits of accuracy required in each experiment. The non-expert user can compare the type configurations suggested by our tool for the various error thresholds, and evaluate the trade-off between accuracy and speedup to make a decision. In general, the user can specify any error threshold of interest. Finally, we run each program enough times to ensure that it runs long enough to obtain reliable timing measurements. In the case of the NAS Parallel Benchmarks, only one run is sufficient; the other programs require thousands or millions of runs.

Finally, we run each program thousands of times (or millions depending on its size) to ensure that the program runs long enough to obtain reliable timing measurements. We run the NAS Parallel Benchmarks only once because their running time is already long enough.

4.2 Experiment Results

Table 1 shows the initial type configuration for each program, which consists of the number of variables (long double (L), double (D), and float (F) types) and the number of double-precision function calls (C) included in the search. We considered switching functions calls for only a subset of the programs under analysis. We switched calls to C Math Library functions such as sin, log, and sqrt. Precimonious is general enough to switch other function calls specified by developers. Table 1 also shows the type configurations suggested by our tool for each error threshold (10^{-10}, 10^{-8}, 10^{-6}, and 10^{-4}). The error threshold indicates the number of accuracy digits required. For example, 10^{-4} roughly means that the result is required to be correct up to 4 digits. The column S corresponds to the number of function calls switched to single precision. The table also includes the number of configurations explored by our LCCSearch algorithm, and its performance [5].

For example, our analysis took as little as 1 minute 3 seconds to find that we can lower the precision of all the variables in the program roots from double to float when the error threshold is 10^{-4}. Note that even the simple experiment of lowering the precision of all the program variables to single precision would be a tedious and time consuming task if done by hand. Our tool finds the type configurations without any manual intervention, and provides a listing of the new type assignment (if any) for each floating-point variable in the program. If desired, the user can run the modified LLVM bitcode of the program as well.

Our analysis runs under an hour for 44 out of the 48 experiments from Table 1 (12 programs × 4 different threshold values). The most expensive analysis was for the program fft with threshold 10^{-10}, which took 116 minutes 51 seconds. In this experiment, 663 type configurations were explored. Note that in this case there are 2^{22} or $4,194,304$ possible program variants.

Table 2 shows the speedup with respect to the original programs. Precimonious finds at least one lower-precision type configuration that results in performance improvement for 9 out of 12 programs. The most significant improvement was observed for the arclength program with a speedup of 41.7% when the error threshold was set to 10^{-10} and 10^{-8}, followed by 37.1% for the simpsons program with error thresholds 10^{-6} and 10^{-4}, and 33.2% for the NAS Parallel Benchmark ep with error threshold 10^{-8} [6].

In general, we would expect that as the error threshold becomes larger, more program variables could be changed to a lower precision while improving performance. However, this is not necessarily true when the resulting configuration uses mixed precision. In some cases, the resulting type configuration might introduce many more casting operations, causing

[5] The error threshold used by the original NAS Parallel Benchmark ep is 10^{-8}, thus we do not consider the error threshold 10^{-10}.

[6] The NAS Parallel Benchmark ep consists of two parts: (1) generating random numbers, and (2) computing the gaussian deviate. Because only the second part is a candidate for precision tuning, we apply the analysis and report speedup for only that part of the program. The overall observed speedup is on average 14%.

Table 1: Analysis results and performance. The column *Initial* gives the number of floating-point variables (long double *L*, double *D*, and float *F*) and the number of functions calls (subcolumn C) included in the search. For each selected error threshold, we give the type configuration found by our algorithm LCCSearch (number of variables per precision and number of function calls switched to single precision (subcolumns S)). The column *#Config* gives the total number of configurations explored during the search. The running time of the analysis is given in column mm:ss.

	Initial				Error Threshold 10^{-10}						Error Threshold 10^{-8}					
Program	L	D	F	C	L	D	F	S	# Config	mm:ss	L	D	F	S	# Config	mm:ss
arclength	9	0	0	3	0	7	2	1	190	4:34	0	7	2	1	190	4:34
simpsons	9	0	0	2	4	5	0	2	91	2:59	0	8	1	2	89	2:53
bessel	0	18	0	0	0	18	0	0	108	32:07	0	18	0	0	10	3:14
gaussian	0	52	0	0	0	52	0	0	1435	86:32	0	52	0	0	1219	79:02
roots	0	19	0	0	0	1	18	0	32	7:03	0	1	18	0	71	12:24
polyroots	0	28	0	0	0	28	0	0	302	31:31	0	28	0	0	244	28:50
rootnewt	0	12	0	0	0	5	7	0	23	5:32	0	5	7	0	57	9:48
sum	0	31	0	0	0	31	0	0	179	16:32	0	31	0	0	193	19:34
fft	0	22	0	0	0	22	0	0	663	116:51	0	22	0	0	423	78:10
blas	0	17	0	0	0	17	0	0	105	23:20	0	17	0	0	105	23:20
ep	0	13	0	4	-	-	-	-	-	-	0	9	4	3	256	58:02
cg	0	32	0	3	0	21	11	2	1040	22:21	0	17	15	3	1131	23:11

	Initial				Error Threshold 10^{-6}						Error Threshold 10^{-4}					
Program	L	D	F	C	L	D	F	S	# Config	mm:ss	L	D	F	S	# Config	mm:ss
arclength	9	0	0	3	1	5	3	2	106	2:37	0	2	7	3	33	0:40
simpsons	9	0	0	2	0	0	9	2	4	0:07	0	0	9	2	4	0:07
bessel	0	18	0	0	0	18	0	0	96	26:20	0	18	0	0	130	37:11
gaussian	0	52	0	0	0	52	0	0	161	12:29	0	52	0	0	201	16:12
roots	0	19	0	0	0	1	18	0	15	4:36	0	0	19	0	3	1:03
polyroots	0	28	0	0	0	28	0	0	161	21:42	0	28	0	0	336	43:17
rootnewt	0	12	0	0	0	7	5	0	25	6:08	0	4	8	0	61	16:56
sum	0	31	0	0	0	31	0	0	267	24:50	0	9	22	0	325	28:14
fft	0	22	0	0	0	0	22	0	3	1:17	0	0	22	0	3	1:16
blas	0	17	0	0	0	0	17	0	3	1:26	0	0	17	0	3	1:06
ep	0	13	0	4	0	7	6	4	117	25:28	0	5	8	4	111	23:53
cg	0	32	0	3	0	32	0	0	684	15:27	0	2	30	3	44	0:57

the new program to run slower. Because our search constraints for valid configurations require both correctness and adequate speed, there may be cases when a smaller threshold might prevent certain variables from being lowered that might have caused the program to become slower due to castings, for example. Thus, a smaller threshold in some cases can lead to finding a type configuration that results in better program performance than a larger (more relaxed) error threshold. Note that a type configuration valid for a small error threshold is also valid for a larger error threshold.

For example, Precimonious finds a type configuration that lowers 18 out of 19 floating-point variables for the program `roots` with threshold 10^{-6}. However, when we use a smaller error threshold of 10^{-8} or 10^{-10}, different type configurations are found (also lowering 18 variables). In this case, the smaller threshold rules out invalid configurations that would have been valid for a larger error threshold but would have run slower.

Precimonious does not find a valid type configuration for any of the selected error thresholds for programs `bessel`, `gaussian`, and `polyroots`. In these cases, there are type configurations that use less precision while producing an accurate enough answer given those error thresholds, however none of these type configurations leads to performance improvement. At least for `bessel`, it is well-known that there exist better and more efficient algorithms that could be used if we know beforehand that the computation can be performed in single precision. In the future, Precimonious could be enhanced to incorporate this kind of domain knowledge.

Table 3 gives more details for the set of results corresponding to threshold value 10^{-6}. For a subset of analyzed programs, the table shows a comparison between the number of floating-point instructions executed by the original program and the suggested tuned program. In particular, we show the number of `float`, `double`, and `long double` loads, stores, arithmetic operations, comparison and casting instructions. In general, we find that it is difficult to determine how different kinds of instructions contribute to improving or degrading program performance.

Based on our results, we classify the programs under analysis into three categories: mixed precision, single precision and double precision. The following subsections explain each category and give some examples.

Mixed Precision: We find that for our test input set, the programs `arclength`, `simpsons`, `roots`, `rootnewt`, `sum`, and the NAS Parallel Benchmarks `ep` and `cg` indeed can use mixed precision to produce an accurate enough answer while improving program performance. For example, program `sum` takes the terms of a series and computes the extrapolated limit of the series using a Levin u-transform. There are 31 floating-point variables in this program. We find that we can lower the precision of 22 of these variables while improving program performance by 15.07%. Another example of mixed precision is illustrated by the `arclength` program described in Section 2, which leads to a performance improvement of 11.0% to 41.7%.

Single Precision: It is possible for Precimonious to determine that all variables in a program can be single precision. However, it is important to be aware that for certain mathematical functions, simply changing the precision of all variables might not be the best alternative to achieve better performance. Often, different algorithms are used for single precision, which can have a better performance. This is the case for programs such as `bessel` and `gaussian` (for which lowering the precision made the program slower, thus we did not suggest any type configuration). In this situation, the user might want to switch algorithms rather than lowering the precision of the double-precision algorithm. Precimonious can still be helpful to indicate that all variables can be single precision, and give a hint to the user to look for single-precision implementations of a given function. Precimonious already has the capability to switch calls to functions from the C Math Library. The user can provide additional functions.

Double Precision: Precimonious can also help to determine whether a given program has already an optimal precision, preventing the user from making any program changes. The program `polyroots` illustrates this scenario. Our tool correctly finds that none of the variables involved in the computation can be changed to a lower precision.

As an experiment, we re-ran our analysis on the `roots` program using two subsets of the originally selected inputs. We found that the type configurations found were different from the type configuration recommended when the full test input set is used. Additionally, both configurations used less precision. This result successfully illustrates the heterogeneity of the selected inputs for this program. As a second experiment, we ran the modified NAS Parallel Benchmarks `ep` and `cg` with new inputs classes B and C. Table 4 shows the verification results. In general, as higher accuracy is required, it is more likely a type configuration could become invalid with untested inputs. For error thresholds 10^{-10} and 10^{-8} the verification failed with inputs other than the input used during the analysis. On the other hand, for error thresholds 10^{-6} and 10^{-4}, 75% of the new inputs passed the verification. In summary, the more inputs used during the analysis, the more widely will the type configurations suggested by Precimonious be valid.

4.3 Additional Example

Figure 6 shows two methods to compute an infinite sum whose true value is known to be 9240. If this were not known, then an error analysis would have to take two sources of error into account.

One source is that only finitely many `(K)` terms have been added. A final addend `tail(K)` is intended to mostly compensate for the finite sum's omitted terms. Thus, `tail(K)` is, roughly, the integral of `term(k)` for `k > K`, and approximates the sum of omitted terms ever better as `K` increases because `term(k)` decreases slowly to zero. Note that `tail(K)` is orders of magnitude bigger than the first `term(K+1)` omitted from the while loop's sum. This loop determines `K` by running until `term(K)` contributes negligibly to its sum. If the error in `tail(K)` were taken into account even approximately, the loop could stop sooner with a far smaller `K`.

Table 2: Speedup observed after precision tuning

	Error Threshold			
Program	10^{-10}	10^{-8}	10^{-6}	10^{-4}
arclength	41.7%	41.7%	11.0%	33.3%
simpsons	13.7%	7.1%	37.1%	37.1%
bessel	0.0%	0.0%	0.0%	0.0%
gaussian	0.0%	0.0%	0.0%	0.0%
roots	6.8%	6.8%	4.5%	7.0%
polyroots	0.0%	0.0%	0.0%	0.0%
rootnewt	0.5%	1.2%	4.5%	0.4%
sum	0.0%	0.0%	0.0%	15.0%
fft	0.0%	0.0%	13.1%	13.1%
blas	0.0%	0.0%	24.7%	24.7%
ep	-	33.2%	32.3%	32.8%
cg	4.6%	2.3%	0.0%	15.9%

Table 4: Verification results for the NAS Parallel Benchmaks `ep` and `cg` when the suggested type configurations are exercised with new inputs

		Error Threshold			
Program	Input	10^{-10}	10^{-8}	10^{-6}	10^{-4}
ep	A	-	Pass	Pass	Pass
ep	B	-	Fail	Pass	Fail
ep	C	-	Fail	Pass	Fail
cg	A	Pass	Pass	Pass	Pass
cg	B	Fail	Fail	Pass	Pass
cg	C	Fail	Fail	Pass	Pass

The second source of error is roundoff. To render roundoff in `term(k)` inconsequential, the numerator 3465 has been chosen to turn the first three `term(k)` elements into small integers computed exactly. Later `term(k)` elements are so much smaller that roundoff in their last digits cannot matter. The rounding errors that matter occur in `sum + term(k)`; they accumulate to corrupt almost the last `log(K)` significant digits of the computed sum. Compensated summation [17] fends off that corruption at an added cost negligible compared with the cost of computing `term(k)`.

This example is interesting because the accuracy required determines the number of iterations needed to compute the sum. If single precision is sufficient, the number of terms reduces from 87290410 to only 3768 terms for the crude sum, and from 61728404 to only 2698 terms for compensated summation. This translates to speedups as high as 5000x. Thus, this summation program illustrates a different kind of numerical program in which, unlike those described in Section 4.2, the arithmetic's precision determines the number of iterations needed to compute a given value. For such programs choosing an error tolerance can be problematical.

5. LIMITATIONS

We note that our tool does not attempt to prove error bounds, or guarantee accurate answers for all possible inputs. These are different and more difficult problems. We rely on the user to provide a set of representative program inputs to be used during the precision tuning process. If the user applies the proposed type configurations to a much worse conditioned input, then we can make no guarantees; indeed even using the highest available precision everywhere in the program may give the wrong answer in this case.

Table 3: Number of instructions executed when the error threshold is 10^{-6}

		roots		rootnewt		fft		blas		arclength	
Instructions		Original	Tuned	Original	Tuned	Original	Tuned	Original	Tuned	Original	Tuned
Loads	L	0	0	0	0	0	0	0	0	27000000	6000000
	D	7786	6465	2506	2289	9106	0	37805	0	0	6000000
	F	0	1321	0	217	0	9106	0	37805	0	15000000
Stores	L	0	0	0	0	0	0	0	0	13000000	10000000
	D	3803	3162	1303	1155	4442	0	13004	0	0	6000000
	F	0	641	0	148	0	4442	13004	0	0	6000000
Arith Ops	L	0	0	0	0	0	0	0	0	20000000	5000000
	D	2209	1992	898	898	4727	49	24600	0	0	15000000
	F	0	217	0	0	0	4678	0	24600	0	0
Comp Ops	L	0	0	0	0	0	0	0	0	0	0
	D	1593	1514	335	335	0	0	603	603	0	0
	F	0	78	0	0	0	0	0	0	0	0
Castings	Trunc	0	207	0	69	0	28	0	0	5000000	11000000
	Ext	0	808	0	217	0	28	0	603	5000000	16000000

```
double t1(double x) {
  return 3465.0/x;
}

double tail(double k) {
  return t1(k + 0.5) + t1(k+1);
}

double t2(double x) {
  return 3465.0/(x*x - 0.0625);
}

double term(double k) {
  return t2(k) + t2(k + 0.5);
}
```

(a) Common Functions

```
int main() {
  double sum, oldsum;
  long int k;
  sum = 0.0;
  oldsum = -1.0;
  k = 0;

  while(sum > oldsum) {
    oldsum = sum;
    k = k + 1;
    sum = sum + term(k);
  }

  sum = sum + tail(k);
  // sum contains the infinite sum
}
```

(b) Crude Sum

```
int main() {
  double sum, comp, oldsum;
  int k;
  sum = 0.0;
  comp = 0.0;
  oldsum = -1.0;
  k = 0;

  while(sum > oldsum) {
    oldsum = sum;
    k = k + 1;
    comp = term(k) + comp;
    sum = sum + comp;
    comp = (oldsum - sum) + comp;
  }
  comp = tail(k) + comp;
  sum = sum + comp;
  // sum contains the infinite sum
}
```

(c) Compensated Sum

Figure 6: Two implementations to calculate infinite sum $\sum_{k>0} (3465/((k+1/2)^2 - 1/16) + 3465/(k^2 - 1/16)) = 9240$

The type configurations suggested by Precimonious need to be mapped to the source code of the program. This step has been performed manually, and requires a significant amount of work for large programs. It would be even more useful to have Precimonious produce a modified program at the source level, without the need to map the suggested type configurations by hand. We hope to automate this process in the near future. Another alternative is to develop a new front-end for our tool to make the program changes at the source code level directly, avoiding the need to perform lower-level program transformations in bitcode to reflect the different type configurations explored by our search algorithm.

The program transformations that Precimonious performs are limited to changing variable declarations, and switching functions calls, when multiple implementations are available. It might be desirable to be able to change the precision of intermediate variables introduced dynamically, e.g., the "same" static variable could have a different precision in certain iterations of a loop. This is doable at the LLVM bitcode level, and is future work. Moreover, there are algorithmic changes that may also lead to significant performance improvement in numerical programs. Precimonious does not consider this currently. Note that these kinds of program transformations would impose additional challenges when mapping the suggested type changes to the source code.

Last, a limitation of our seach algorithm is that we might get trapped in a local minimum. So far, Precimonious does not take into account how variables interact wtih each other. For example, if two variables are often used as operands to the same arithmetic operators, it might be worth grouping them as a single change. This is because changing only one of them would lead to an additional casting that would slow things down. We are planning to perform an initial static analysis to determine the relationship between floating-point variables, which could be taken into account during the search.

6. RELATED WORK

Our approach considers the problem of automatically finding a type configuration for floating-point variables so that the resulting program still produces an accurate enough answer without exceptions while improving performance. In recent work developed independently of our own, Lam et al. propose a framework for automatically finding mixed-precision floating-point computation [20]. This work appears to most resemble ours. Their scheme finds double precision instructions that can be replaced by single precision. They propose a brute-force search using heuristics. Their goal is to maximize the number of instructions that can be changed to single precision, and they operate on the binary code. The configurations proposed might not produce accurate results for tested inputs, and speedup is not explicitly considered as a goal. Furthermore, precision changes are described at the instruction level, thus it might be more difficult to map them to the source code.

FloatWatch is a dynamic execution profiling tool for floating point programs which is designed to identify instructions that can be computed in a lower precision [8]. It works by first computing the overall range of values for each instruction of interest. Using this information, the tool recommends to use less precision if possible. Darulova and Kuncak also implemented a dynamic range analysis feature for the `Scala` language [11]. The approach uses interval and affine forms to represent the input, and examine how errors are magnified by each operation during execution. Their work might also be used for precision tuning purposes, by first computing a dynamic range for each instruction of interest and then tuning the precision based on the computed range, similar to *FloatWatch*. However, range analysis often incurs overestimates too large to be useful for precision tuning analysis. *Gappa* is another tool that uses range analysis to verify and prove formal properties of floating-point programs [12]. In the context of precision tuning, one can use *Gappa* to verify ranges for certain variables and expressions in a program, and then choose the appropriate precision for these variables and expressions. Nevertheless, Gappa scales only to small programs with simple structures and several hundreds of operations, and thus is used mostly for verifying elementary functions.

Our work is also related to a large body of work on accuracy analysis. Benz et al. [6] presented a dynamic analysis approach for finding accuracy problems. Their approach computes every floating-point instructions side by side in higher precision. The higher precision computation is stored in a *shadow value*. If the differences between the original value and the shadow value become too large, their tool reports a potential accuracy problem. *FPInst* is another tool that computes floating point errors for the purpose for detecting accuracy problem [1]. It also computes a shadow value side by side, but it stores an absolute error in double precision instead. Lam et al. [19] propose a tool for detecting cancellation. Cancellation is detected by first computing the exponent of the result and the operands. If the exponent of the result is less than the maximum of those of the two operands, an cancellation has occurred. *Fluctuat* is another static analysis tool based on abstract interpretation and affine arithmetic to analyze the propagation of rounding errors in C programs. It can detect potential errors like run-time errors or unstable tests due to numerical problems [13]. PRECIMONIOUS can complement accuracy analysis for debugging purposes in the following way. It can attempt to tune the set of potentially error-generating floating-point instructions to have higher precision until the accuracy problem goes away. The cost model could be changed to favor a configuration that requires the fewest changes.

Autotuning of codes to improve performance is a very large area of research, just a few citations being [7, 14, 23, 27, 28]. That previous work has however not tried to tune floating-point precision in the way this work does.

7. CONCLUSION

In this paper, we have presented an automated algorithm for tuning the floating-point precision of numerical programs. Our algorithm attempts to find a type configuration for floating-point variables so that the program still produces an accurate enough answer without exceptions while improving peformance. We have implemented our algorithm in an efficient and publicly available tool called PRECIMONIOUS. Initial evaluation on eight programs using the `GSL` library, two NAS Parallel Benchmarks, and three other numerical programs shows encouraging results: we are able to discover type configurations that result in performance improvements as high as 41%.

In the future we would like to apply our technique to a wider range of programs beyond the scientific computing domain. We would also like to combine our tool with test generation techniques such as concolic testing [5, 25], so that, in the absence of a test suite, we can generate a more representative input set, making our tuning recommendations more stable across different inputs. Finally, we can generalize our technique to support not only variable precision but also operation, language construct and algorithm choices, which is also an interesting future research direction.

The PRECIMONIOUS source code, and all the data and results presented in this paper are available under BSD license at `https://github.com/corvette-berkeley/precimonious`.

8. ACKNOWLEDGEMENTS

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 number DE-SC0008699 and through a gift from Oracle.

9. REFERENCES

[1] D. An, R. Blue, M. Lam, S. Piper, and G. Stoker. Fpinst: Floating point error analysis using dyninst, 2008. URL `http://www.freearrow.com/downloads/files/fpinst.pdf`.

[2] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.

[3] D. H. Bailey. Resolving numerical anomalies in scientific computation, 2008. URL `http://www.davidhbailey.com/dhbpapers/numerical-bugs.pdf`.

[4] D. H. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0, 1995.

[5] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In R. Giacobazzi and R. Cousot, editors, *POPL*, 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, *PLDI*, 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] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.*, 34(4):17:1–17:22, July 2008. . URL `http://doi.acm.org/10.1145/1377596.1377597`.

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

[11] E. Darulova and V. Kuncak. Trustworthy numerical computation in scala. In C. V. Lopes and K. Fisher, editors, *OOPSLA*, pages 325–344. ACM, 2011.

[12] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Trans. Comput.*, 60(2):242–253, Feb. 2011. . URL `http://dx.doi.org/10.1109/TC.2010.128`.

[13] 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 *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '09, pages 53–69, Berlin, Heidelberg, 2009. Springer-Verlag. . URL `http://dx.doi.org/10.1007/978-3-642-04570-7_6`.

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

[15] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[16] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, ARITH '01, pages 155–, Washington, DC, USA, 2001. IEEE Computer Society. URL `http://dl.acm.org/citation.cfm?id=872021.872445`.

[17] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[18] W. Kahan. A Mathematical Curiosity, 2013.

[19] M. O. Lam, J. K. Hollingsworth, and G. W. Stewart. Dynamic floating-point cancellation detection. In *1st International Workshop on High-performance Infrastructure for Scalable Tools*, 2011.

[20] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In A. D. Malony, M. Nemirovsky, and S. P. Midkiff, editors, *ICS*, pages 369–378. ACM, 2013.

[21] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[22] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, June 2002. . URL `http://doi.acm.org/10.1145/567806.567808`.

[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. URL `https://github.com/travitch/whole-program-llvm`.

[25] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272. ACM, 2005.

[26] I. C. Society. IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008, Aug. 2008. URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4610935`.

[27] 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.

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

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