

# Sound Mixed-Precision Optimization with Rewriting

Eva Darulova  
MPI-SWS  
eva@mpi-sws.org

Einar Horn  
UIUC  
eahorn2@illinois.edu

Saksham Sharma  
IIT Kanpur  
sakshams@cse.iitk.ac.in

## ABSTRACT

Finite-precision arithmetic computations face an inherent tradeoff between accuracy and efficiency. The points in this tradeoff space are determined, among other factors, by different data types but also evaluation orders. To put it simply, the shorter a precision's bit-length, the larger the roundoff error will be, but the faster the program will run. Similarly, the fewer arithmetic operations the program performs, the faster it will run; however, the effect on the roundoff error is less clear-cut. Manually optimizing the efficiency of finite-precision programs while ensuring that results remain accurate enough is challenging. The unintuitive and discrete nature of finite-precision makes estimation of roundoff errors difficult; furthermore the space of possible data types and evaluation orders is prohibitively large.

We present the first fully automated and sound technique and tool for optimizing the performance of floating-point and fixed-point arithmetic kernels. Our technique *combines* rewriting and mixed-precision tuning. Rewriting searches through different evaluation orders to find one which minimizes the roundoff error at no additional runtime cost. Mixed-precision tuning assigns different finite precisions to different variables and operations and thus provides finer-grained control than uniform precision. We show that when these two techniques are designed and applied together, they can provide higher performance improvements than each alone.

## KEYWORDS

mixed-precision tuning, floating-point arithmetic, fixed-point arithmetic, static analysis, optimization

### ACM Reference format:

Eva Darulova, Einar Horn, and Saksham Sharma. 2017. Sound Mixed-Precision Optimization with Rewriting. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17)*, 11 pages.  
DOI: 10.1145/nnnnnnnn.nnnnnnn

## 1 INTRODUCTION

Finite-precision computations, such as those found in embedded and scientific computing applications, face an inherent tradeoff between accuracy and efficiency due to unavoidable roundoff errors whose magnitude depends on several aspects. One of these is the data type chosen: in general, the larger the data type (e.g. in terms of bits), the smaller the roundoff errors will be. However, increasing the bit-length typically leads to decreases in performance. Additionally, finite-precision arithmetic is not associative or distributive. Thus, an attempt to reduce the running time of a computation by reducing the number of arithmetic operations

(e.g.  $a * b + a * c \rightarrow a * (b + c)$ ) may lead to a higher and possibly unacceptable roundoff error. Due to the unintuitive nature of finite-precision and the subtle interactions between accuracy and efficiency, manual optimization is challenging and automated tool support is needed.

*Mixed-precision Tuning.* In order to save valuable resources like time, memory or energy, we would like to choose the smallest data type which still provides sufficient accuracy. Not all applications require high precision, but how much precision an application needs depends highly on context: on the computations performed, the magnitude of inputs, and the expectations of the environment, so that no one-size-fits-all solution exists. Today, the common way to program is to pick a seemingly safe, but often overprovisioned, data type — for instance, uniform double floating-point precision.

Mixed-precision, where different operations are performed in potentially different precisions, increases the number of points on the accuracy-efficiency tradeoff space and thus increases the possibility for more resource savings. With uniform precision, if one precision is just barely not enough, we are forced to upgrade all operations to the next higher precision. This can increase the running time of the program substantially. Therefore, it would be highly desirable to upgrade only part of the operations; just enough to meet the accuracy target, while increasing the execution time by the minimum.

One of the challenges in choosing a finite precision — uniform or mixed — is ensuring that the roundoff errors remain below an application-dependent acceptable bound. Recent work has provided automated techniques and tools which help the programmer choose between different uniform precisions by computing sound worst-case numerical error bounds [12, 19, 28, 43].

However, selecting a suitable mixed precision is significantly more difficult than choosing a uniform precision. The number of different type assignments to variables and operations is too large to explore exhaustively. Furthermore, neither roundoff errors nor the performance of mixed-precision programs follow intuitive rules, making manual tuning very challenging. For instance, changing one particular operation to lower precision may produce a smaller roundoff error than changing two (other) operations. Furthermore, mixed-precision introduces cast operations, which may increase the running time, even though the accuracy decreases.

In high-performance computing (HPC), mixed-precision tuning [25, 40] approaches use dynamic techniques to estimate roundoff errors, and thus do not provide accuracy guarantees. This makes them unsuitable, for instance, for many safety-critical embedded applications. The FPTuner tool [9] is able to soundly tune mixed-precision for straight-line programs, but it requires user guidance for choosing which mixed-precision variants are more efficient and is thus not fully automated. Furthermore, its tuning time can be prohibitively large.

*Rewriting.* Another possibility to improve the efficiency of finite-precision arithmetic is to reduce the number of operations that need to be carried out. This can be achieved without changing the *real-valued* semantics of the program by rewriting the computation using laws like distributivity and associativity. Unfortunately, these laws do not hold for finite-precision computations: changing the order of a computation changes the magnitude of roundoff errors committed, but in unintuitive ways. Previous work has focused on automated techniques for finding a rewriting (i.e. re-ordering of computations) such that roundoff errors are minimized [14, 33]. However, optimizing for accuracy may increase the number of arithmetic operations and thus the execution time.

*Combining Mixed-Precision Tuning and Rewriting.* We propose to combine mixed-precision tuning with rewriting in a *fully automated* technique for performance optimization of finite-precision arithmetic computations. Our approach is *sound* in that generated programs are guaranteed to satisfy user-specified roundoff error bounds, while our performance improvements are best effort (due to the complexity and limited predictability of today's hardware). Our rewriting procedure takes into account both accuracy and the number of arithmetic operations. It can reduce the running time of programs directly, but more importantly, by improving the accuracy, it allows for more aggressive mixed-precision tuning. To the best of our knowledge, this is the first combination of mixed-precision tuning and rewriting for performance optimization.

We combine a search algorithm which was successfully applied in HPC mixed-precision tuning [40] with a sound static roundoff error analysis [12] and a static performance cost function to obtain a mixed-precision tuning technique which is both sound and fully automated as well as efficient. We furthermore modify a rewriting optimization algorithm based on genetic programming [14] to consider both accuracy and performance.

While most of the building blocks of our approach have been presented before, their effective combination requires careful adaptation and coordination, as our many less-successful experiments have shown. Just as a manual optimization is challenging due to the subtle interactions of finite-precision accuracy and performance, so is the design of an automated technique.

We focus on arithmetic kernels, and do not consider conditionals and loops. Our technique (as well as FPTuner's) can be extended to conditionals by considering individual paths separately as well as to loops by optimizing the loop body only and thus reducing both to straight-line code. The challenge currently lies in the sound roundoff error estimation, which is known to be hard and expensive for conditionals and loops [13, 20], and is largely orthogonal to the focus of this paper.

Our technique is applicable and implemented in a tool called Anton for both floating-point as well as fixed-point arithmetic. While floating-point support is standardized, fixed-point arithmetic is most effective in combination with specialized hardware. We focus in this paper on the algorithmic aspects of optimizing arithmetic kernels and leave a thorough investigation of specialized hardware implementations for future work.

For floating-point arithmetic, we evaluate Anton on standard benchmarks from embedded systems and scientific computing. We observe that rewriting alone improves performance by up to 17%

and for some benchmarks even more by reducing roundoff errors sufficiently to enable uniform double precision where the original benchmark requires (uniform) quad precision. Mixed-precision tuning improves performance by up to 45% when compared to the uniform precision version which would be needed to satisfy the required accuracy. In combination with rewriting, Anton improves performance by up to 54% (93% for those cases where rewriting enables uniform double precision), and we also observe that it improves performance for more benchmarks than when using mixed-precision tuning or rewriting alone.

We focus on performance, although our algorithm is independent of the optimization objective such that - with the corresponding cost function - memory or energy optimization is equally possible.

*Contributions.* To summarize, in this paper we present:

- An optimization procedure based on rewriting, which takes into account both accuracy and performance.
- A sound, fully automated and efficient mixed-precision tuning technique.
- A carefully designed combination of rewriting and mixed-precision tuning, which provides more significant performance improvements than each of them alone.
- An implementation in a tool called Anton, which generates optimized source programs in Scala and in C and which supports both floating-point as well as fixed-point arithmetic. We plan to release Anton as open source.
- We show the effectiveness of our tool on a set of arithmetic kernels from embedded systems and scientific computing and compare it against the state-of-the-art.

## 2 OVERVIEW

We first provide a high-level overview and explain the key ideas of our approach using an example. Inspired by the tool Rosa [12], the input to our tool Anton is a program written in a real-valued specification language. (Nothing in our technique depends on this particular frontend though.) Each program consists of a number of functions which are optimized separately. The following nonlinear embedded controller [14] is one such example function:

```
def rigidBody1(x1: Real, x2: Real, x3: Real): Real = {
  require(-15.0 <= x1 && x1 <= 15 && -15.0 <= x2 &&
    x2 <= 15.0 && -15.0 <= x3 && x3 <= 15)
  -x1*x2 - 2*x2*x3 - x1 - x3
} ensuring(res => res +/- 1.75e-13)
```

In the function's precondition (the **require** clause) the user provides the ranges of all input variables, on which the magnitudes of roundoff errors depend. The postcondition (the **ensuring** clause) specifies the required accuracy of the result in terms of worst-case absolute roundoff error. For our controller, this information may be, e.g., determined from the specification of the system's sensors as well as the analysis of the controller's stability [29]. The function body consists of an arithmetic expression (with +, -, \*, /,  $\sqrt{\phantom{x}}$ ) with possibly local variable declarations.

As output, Anton generates a mixed-precision source-language program, including all type casts, which is guaranteed to satisfy the given error bound and is expected to be the most efficient one

among the possible candidates. Anton currently supports fixed-point arithmetic with bitlengths of 16 or 32 bits or IEEE754 single (32 bit) and double (64 bit) floating-point precision as well as quad precision (128 bit). The latter can be implemented on top of regular double-precision floating-points [2]. Anton can be easily extended to support other fixed- or floating-point precisions; here we have merely chosen a representative subset.

Our approach decouples rewriting from the mixed-precision tuning. To find the optimal program, i.e. the most efficient one given the error constraint, we would need to optimize both the evaluation order as well as mixed-precision simultaneously: i) the evaluation order determines which mixed-precision type assignments to variables and operations are feasible, and ii) the mixed-precision assignment influences which evaluation order is optimal. Unfortunately, this would require an exhaustive search [14], which is computationally infeasible. We thus choose to separate rewriting from mixed-precision tuning and further choose (different) efficient search techniques for each.

*Step 1: Rewriting.* Anton first rewrites the input expression into one which is equivalent under a real-valued semantics, but one which has a smaller roundoff error when implemented in finite-precision and which does not increase the number of arithmetic operations. Rewriting can increase the opportunities for mixed-precision tuning, because a smaller roundoff error may allow more lower-precision operations. The second objective makes sure that we do not accidentally increase the execution time of the program by performing more arithmetic operations and even lets us improve the performance of the expression directly.

Anton's rewriting uses a genetic algorithm to search the vast space of possible evaluation orders efficiently. At every iteration, the algorithm applies real-valued identities, such as associativity and distributivity, to explore different evaluation orders. The search is guided by a fitness function which bounds the roundoff errors for a candidate expression - the smaller the error, the better. This error computation is done wrt. uniform precision, as the mixed-precision type assignment will only be determined later. While the precision can affect the result of rewriting, we empirically show that the effect is small (section 4).

This approach is heavily inspired by the algorithm presented in Darulova et al. [14] which optimized for accuracy only. We have made important adaptations, however, to make it work in practice for optimizing for performance as well as to work well with mixed-precision tuning.

For our running example, the rewriting phase produces the following expression, which improves accuracy by 30.39% and does not change the number of operations:

$$(-(x1 * x2) - (x1 + x3)) - ((2.0 * x2) * x3)$$

To give an intuition why this seemingly small change makes such a difference, note that the magnitude of roundoff errors depends on the possible ranges of intermediate variables. Even small changes in the evaluation order can have large effects on these ranges and consequently also on the roundoff errors.

*Step 2: Code Transformation.* To facilitate mixed-precision tuning, Anton performs two code transformations: constants are assigned

to fresh variables and the remaining code is converted into three-address form. By this, every constant and arithmetic operation corresponds to exactly one variable, whose precision will be tuned during phase 4. If not all arithmetic operations should be tuned, i.e. a more coarse grained mixed-precision is desired, then this step can be skipped.

*Step 3: Range Analysis.* The evaluation order now being fixed, Anton computes the real-valued ranges of all intermediate subexpressions and caches the results. Ranges are needed for bounding roundoff errors during the subsequent mixed-precision tuning, but because the *real-valued* ranges are not affected by different precisions, Anton computes them only once for efficiency.

*Step 4: Mixed-precision Tuning.* To effectively search the space of possible mixed-precision type assignments, we choose a variation of the delta-debugging algorithm used by Precimonious [40], which prunes the search space in an effective way. It starts with all variables in the highest available precision and attempts to lower variables in a systematic way until it finds that no further lowering is possible while still satisfying the given error bound. We have also tried to apply a genetic algorithm for mixed-precision tuning, but observed that it was quite clearly not a good fit (we omit the experimental results for space reasons).

Unlike Precimonious, which evaluates the accuracy and performance of different mixed-precisions dynamically, Anton uses a static sound error analysis as well as a static (but heuristic) performance cost function to guide the search. The performance cost function assigns (potentially different) abstract costs to each arithmetic as well as cast operation. Using static error and cost functions reduces the tuning time significantly, and further allows tuning to be run on different hardware than the final generated code.

For our running example, Anton determines that uniform double floating-point precision is not sufficient and generates a tuned program which runs 43% faster than the quad uniform precision version, which is the next available uniform precision in the absence of mixed-precision:

```
def rigidBody1(x1: Quad, x2: Quad, x3: Double): Double = {
  (-d (x1 *q x2) - d (x1 +q x3)) - d ((x2 *q 2.0f) *d x3)
}
```

For readability, we have inlined the expression and use the letters 'd' and 'q' to mean that the operation is performed in double and quad precision respectively. The entire optimization including rewriting takes about 4 seconds. Had we used only the mixed-precision tuning without rewriting, the program would still run 28% faster than quad precision.

*Step 5: Code Generation.* Once mixed-precision tuning finds a suitable type configuration, Anton generates the corresponding finite-precision program (in Scala or C), inserting all necessary casts, and in the case of fixed-point arithmetic all necessary bit-shift operations.

Our entire approach is parametric in the finite-precision used, and thus works equally for fixed-point arithmetic. Furthermore, it is geared towards optimizing the performance of programs under the (hard) constraint that the given error bound is guaranteed to be satisfied. Other optimization criteria like memory and energy

are also conceivable, and would only require changing the cost function.

### 3 BACKGROUND

We first review necessary background about finite-precision arithmetic and sound roundoff error estimation, which is an important building block for both rewriting and mixed-precision tuning.

#### 3.1 Floating-point Arithmetic

We assume standard IEEE754 single and double precision floating-point arithmetic, in rounding-to-nearest mode and the following standard abstraction of IEEE754 arithmetic operations:

$x \circ_{fl} y = (x \circ y)(1 + \delta)$ ,  $|\delta| \leq \epsilon_m$  where  $\circ \in +, -, *, /$  and  $\circ_{fl}$  denotes the respective floating-point version, and  $\epsilon_m$  bounds the maximum relative error (which is  $2^{-24}$ ,  $2^{-53}$  and  $2^{-113}$  for single, double and quad precision respectively). Unary minus and square root follow similarly. We further consider NaNs (not-a-number special values), infinities and ranges containing only denormal floating-point numbers to be errors and Anton's error computation technique detects these automatically. We note that under these assumptions the abstraction is indeed sound.

#### 3.2 Fixed-point Arithmetic

Floating-point arithmetic requires dedicated support, either in hardware or in software, and depending on the application, this support may be too costly. An alternative is fixed-point arithmetic which can be implemented with integers only, but which in return requires that the radix point alignments are precomputed at compile time. While no standard exists, fixed-point values are usually represented as (signed) bit vectors with an integer and a fractional part, separated by an implicit radix point. At runtime, the alignments are then performed by bit-shift operations. These shift operations can also be handled by special language extensions for fixed-point arithmetic [21]. For more details please see [1], whose fixed-point semantics we follow. We use truncation as the rounding mode for arithmetic operations. The absolute roundoff error at each operation is determined by the fixed-point format, i.e. the (implicit) number of fractional bits available, which in turn can be computed from the range of possible values at that operation.

#### 3.3 Sound Roundoff Error Estimation

We build upon Rosa's static error computation [12], which we review here. Keeping with Rosa's notation, we denote by  $f$  and  $x$  a mathematical real-valued arithmetic expression and variable, respectively, and by  $\tilde{f}$  and  $\tilde{x}$  their floating-point counterparts. The worst-case absolute error that the error computation approximates is  $\max_{x \in [a, b]} |f(x) - \tilde{f}(\tilde{x})|$  where  $[a, b]$  is the range for  $x$  given in the precondition. The input  $x$  may not be representable in finite-precision arithmetic, and thus we consider an initial roundoff error:  $|x - \tilde{x}| = |x| * \delta$ ,  $\delta \leq \epsilon_m$  which follows from subsection 3.1. This definition extends to multi-variate  $f$  component-wise.

At a high level, error bounds are computed by a data-flow analysis over the abstract syntax tree, which computes for each intermediate arithmetic expression (1) a bound on the real-valued range, (2) using this range, the propagated errors from subexpressions

and the newly committed worst-case roundoff error. For a more detailed description, please see [12].

For our rewriting procedure, since intermediate ranges change with different evaluation orders, we compute both the ranges and the errors at the same time. For mixed-precision tuning, where real-valued ranges remain constant, we separate the computations and only compute ranges once.

Anton currently does not support additional input errors, e.g. from noisy sensors, but note that an extension is straight-forward. In fact, a separate treatment of roundoff errors and propagated input errors may be desirable [13].

We compute absolute errors. An automated and general estimation of relative errors ( $|f(x) - \tilde{f}(\tilde{x})|/|f(x)|$ ), though it may be more desirable, presents a significant challenge today. To the best of our knowledge, state-of-the-art static analyses only compute relative errors from an absolute error bound, which is then not more informative. Furthermore, relative error is only defined if the range of the expression in question (i.e. the range of  $f(x)$ ) does not include zero, which unfortunately happens very often in practice.

*Range Estimation.* Clearly, accurately estimating ranges is the main component in the error bound computation, and is known to be challenging, especially for nonlinear arithmetic. This challenge was addressed in previous work on finite-precision verification [12, 19, 43]. Interval arithmetic (IA) [31] is an efficient choice for range estimation, but one which often introduces large overapproximations as it does not consider correlations between variables. Affine arithmetic [16] tracks *linear* correlations, and is thus sometimes better (though not always) in practice. The overapproximations due to nonlinear arithmetic ( $*$ ,  $/$ ,  $\sqrt{\phantom{x}}$ ) can be mitigated by refining ranges computed by IA with a complete (though expensive) nonlinear arithmetic decision procedure inside the Z3 [17] SMT solver [12]. Anton's computation builds on this work and is parametric in the range arithmetic and currently supports interval and affine arithmetic as well as the combination with SMT.

### 4 REWRITING OPTIMIZATION

The goal of Anton's rewriting optimization is to find an order of computation which is equivalent to the original expression under a real-valued semantics, but which exhibits a smaller roundoff error in finite-precision - while not increasing the execution time. We first review previous work that we build upon and then describe the concrete adaptation in Anton.

#### 4.1 Genetic Search for Rewriting

An exhaustive search of all possible rewritings or evaluation orders is computationally infeasible. Even for only linear arithmetic, the problem of finding an optimal order is NP-hard [14] and does not allow a divide-and-conquer or gradient-based method such that a heuristic and incomplete search becomes necessary.

Genetic programming [35] is an evolutionary heuristic search algorithm which iteratively evolves (i.e. improves) a population of candidate expressions, guided by a fitness function. The search is initialized with copies of the initial expression. At every iteration, expressions are selected from the current population based on their fitness, and then mutated to form the next population. For rewriting, the fitness is the worst-case roundoff error - the smaller

the better. The selected expressions are randomly mutated using mathematical real-valued identities, e.g.  $a + (b + c) \rightarrow (a + b) + c$ . In this fashion, the algorithm explores different rewritings. The key idea is that the likelihood of an expression to be selected depends on its fitness - fitter expressions are more likely to be selected - and thus the search converges with each iteration towards expressions with smaller roundoff errors. Furthermore, even less-fit expressions have a non-zero probability of being selected, thus helping to avoid local minima.

The output of the procedure is the expression with the least roundoff error seen during the run of the search. Darulova et al. [14] used a static analysis as described in subsection 3.3 as the fitness function, with smaller roundoff error being better. Their approach was implemented for fixed-point arithmetic only, and the optimization goal was to reduce roundoff errors.

## 4.2 Rewriting in Anton

We instantiate the algorithm described above with a population of 30 expressions, 30 iterations and tournament selection [35] for selecting which expressions to mutate. These are successful settings identified in [14]. We do not use the crossover operation, because it had only limited effects. We further extend the rather limited set of mutation rules in [14] with the more complete one used by the (un-sound) rewriting optimization tool called Herbie [33] (see section 7). These rules are still based on mathematical identities. For the static error function, as described in subsection 3.3, we choose interval arithmetic for computing ranges and affine arithmetic for tracking errors, which provide a good accuracy-performance tradeoff.

The algorithm described in subsection 4.1 reduces roundoff errors, but may - and as we experimentally observed often does - increase the number of operations and thus the execution time. This may negate any advantage reduced roundoff errors bring for mixed-precision tuning. Furthermore, it is not clear at this point with respect to which precision to perform the rewriting as a mixed-precision type assignment is not available.

**4.2.1 Optimizing for Performance.** We modify the search algorithm to return the expression which does not increase the number of arithmetic operations beyond the initial count, and which has the smallest worst-case roundoff error. We do not use a more sophisticated cost function, as for this the actual final type assignment would be needed (which only becomes available after mixed-precision tuning). We have also implemented a variation of the search which minimizes the number of arithmetic expressions, while not increasing the roundoff error beyond the error of the initial expression. However, we found empirically in our experiments that this produces worse overall results in combination with mixed-precision tuning, i.e. reducing the roundoff was more beneficial than reducing the operation count. For space reasons, we omit this experimental comparison.

**4.2.2 Optimizing with Uniform Precision.** The static error analysis, which we use as the fitness function during search has to be performed wrt. to some mixed or uniform precision, and different choices may result in the algorithm returning syntactically different rewritten expressions. As the final (ideal) mixed-precision type

assignment is not available when Anton performs rewriting, it has to choose some precision without knowing the final assignment.

The main aspect which determines which evaluation order is better over another are the ranges of intermediate variables - the larger the ranges, the more already accumulated roundoff errors will be magnified. These intermediate ranges differ only little between different precisions, because the roundoff errors are small in comparison to the real-valued ranges. Thus, we do not expect that different precision affect the result of rewriting very much.

We performed the following experiment to validate this intuition. We ran rewriting repeatedly on the same expression, but with the error analysis wrt. uniform single, double and quad floating-point precision as well as up to 50 random mixed-precision type assignments. We picked each mixed-precision assignment as the baseline in turn. We evaluated the roundoff errors of the expressions returned by the uniform precision rewritings under this mixed-precision assignment. If rewriting in uniform precision produces an expression which has the same or a similar error as the expressions returned with rewriting wrt. mixed-precision, then we consider the uniform precision to be a good proxy. We counted how often each of the three uniform precisions was such a good proxy, where we chose the threshold to be that the errors should be within 10%. For space reasons, we only summarize the results. Single and double floating-point precision were a good proxy in roughly 80% of the cases, whereas quad precision in 75%. When the mixed-precision assignments were in fixed-point precision, single, double and quad uniform precision all achieve 69% accuracy. Performing the rewriting wrt. fixed-point arithmetic is not more beneficial either. Finally, rewriting in uniform precision never increased the errors (when evaluated in the mixed-precision baseline). We thus (randomly) choose to perform the rewriting with respect to double floating-point precision.

## 5 SOUND MIXED-PRECISION TUNING

After rewriting, Anton pre-processes expressions as described in step 2 in section 2 and computes the now-constant ranges of intermediate expressions. Since the range computation needs to be performed only once, we choose the more expensive but also more accurate range analysis using a combination of interval arithmetic and SMT [12]. We again first review previous work that we build upon before explaining Anton's technique in detail.

### 5.1 Delta-Debugging for Precision Tuning

Delta-debugging has been originally conceived in the context of software testing for identifying the smallest failing testcase [45]. Rubio-González et al. [40] have adapted this algorithm for mixed-precision tuning in the tool Precimonious. It takes as input:

- a list of variables to be tuned ( $\tau$ ) as well as a list of all other variables with their constant precision assignments ( $\phi$ )
- an *error function* which bounds the roundoff error of a given precision assignment
- a *cost function* approximating the expected performance
- an error bound  $e_{max}$  to be satisfied.

The output is a precision assignment for variables in  $\tau$ . A partial sketch of the algorithm is depicted in Figure 1, where the boxes represent sets of variables in  $\tau$ . Consider the case where variables

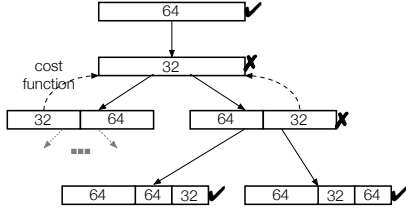


Figure 1: Sketch of the delta-debugging algorithm

can be in single (32 bit) and double (64 bit) floating-point precision. The algorithm starts by assigning all variables in  $\tau$  to the highest precision, i.e. to double precision. It uses the error function to check whether the roundoff error is below  $e_{max}$ . If it is not, then an optimization is futile, because even the largest precision cannot satisfy the error bound.

If the check succeeds, the algorithm tries to *lower* all variables in  $\tau$  by assigning them to single precision. Again, it computes the maximum roundoff error. If it is below  $e_{max}$ , the search stops as single precision is sufficient. If the error check does not succeed, the algorithm splits  $\tau$  into two equally sized lists  $\tau_1$  and  $\tau_2$  and recurses on each separately. When recursing on  $\tau_1$ , the new list of variables considered for lowering becomes  $\tau' = \tau_1$  and the list of constant variables becomes  $\phi' = \phi + \tau_2$ . The case for recursing on  $\tau_2$  is symmetric. When a type assignment is found which satisfies the error bound  $e_{max}$ , the recursion stops. Since several valid type assignments can be found, a cost function is used to select the one with lowest cost (i.e. best performance).

The algorithm is generalized to several precisions by first running it with the highest two precisions. In the second iteration, the variables which have remained in the highest precision become constant and move to  $\phi$ . The optimization is then performed on the new  $\tau$  considering the second and third highest precision.

## 5.2 Mixed-Precision Tuning in Anton

We have instantiated this algorithm in Anton and describe now our adaptations which were important to obtain *sound* mixed-precision assignments as well as good results.

**5.2.1 Static Error Analysis.** Precimonious estimates roundoff errors by dynamically evaluating the program on several random inputs. This approach is not sound, and also in general inefficient, as a large number of program executions is needed for a reasonably confident error bound. Anton uses a *sound static* roundoff error analysis, which is an extension of Rosa’s uniform-precision error analysis from subsection 3.3 to support mixed-precision. This extension uses affine arithmetic for the error computation and considers roundoff errors incurred due to down-casting operations.

Precimonious can handle any program, including programs with loops. Our static error function limits which kinds of programs Anton can handle to those for which the error bounds can be verified, but for those it provides accuracy *guarantees*. We note, however, that our approach can potentially be extended to loops with techniques from [13], by considering the loop body only.

**5.2.2 Tuning All Variables.** Unlike Precimonious, which optimizes only the precisions of declared variables, Anton optimizes the

precisions of all variables and intermediate expressions by transforming the program prior to mixed-precision tuning into three-address form (this transformation can also be skipped, if desired).

The precision of an arithmetic operation is determined by the precisions of its operands as well as the variable that the result is assigned to. In general, we follow a standard semantics, where the operation is performed in the highest operand precision, with one exception. For example, for the precision assignment  $\{x \rightarrow \text{single}, y \rightarrow \text{single}, z \rightarrow \text{double}\}$ , and expression `val z = x + y`, we choose the interpretation  $z = x.\text{toDouble} + y.\text{toDouble}$  instead of  $z = (x + y).\text{toDouble}$ , so that the operation is performed in the higher precision, thus loosing less accuracy. Our experiments (whose results are not shown for space reasons) have confirmed that this indeed provides better overall results.

Delta-debugging operates on a list of variables that it optimizes. We have observed in our experiments that it is helpful when the variables are sorted by order of appearance in the program. Our hypothesis is that delta-debugging is more likely to assign ‘neighboring’ variables the same type, which in general is likely to reduce type casts and thus cost.

We found that often constants are representable in the lowest precision, e.g. when they are integers. It is thus tempting to keep those constants in the lowest precision. However, we found that, probably due to cast operations, this optimization was not a universal improvement, so that Anton optimizes constants just like other variables.

**5.2.3 Static Cost Function.** Precimonious uses dynamic evaluation to estimate the expected running time. We note that this approach is quite inefficient, but also not entirely reliable, as running times can vary substantially between runs (our benchmarking tool takes several seconds per benchmark until steady-state). It furthermore restricts the tuning to the specific platform that the tuning is run on. FPTaylor, on the other hand, optimizes for the number of lower-precision operations (more is better) and provides a way for the user to manually restrict the overall number of cast operations, and provides the possibility to constrain certain variables to have the same precision (‘ganging’). Knowing up front how many cast operations are needed is quite challenging.

We instead propose a static cost function to obtain an overall technique which is efficient as well as fully automated. Note that this function needs to be able to distinguish only which of two mixed-precision assignments is the more efficient one, and does not need to predict the actual running times. We are aiming for a practical solution and are aware that more specialized approaches are likely to provide better prediction accuracy. As we focus in this paper on the algorithmic aspects, we leave this for future work. We have implemented and experimentally evaluated several cost function candidates for floating-point arithmetic, which all require only a few milliseconds to run:

- *Simple cost* assigns a cost of 1, 2 and 4 to single, double and quad precision arithmetic and cast operations respectively.
- *Benchmarked cost* assigns abstract costs to each operation based on benchmarked average running times, i.e. we benchmark each operation in isolation with random inputs. This cost function is platform specific and different

arithmetic operations have different costs (e.g. addition is generally cheaper than division).

- *Operation count cost* counts the number of operations performed in each precision, recorded as a tuple and ordered lexicographically, i.e. more higher-precision operations lead to a higher cost. This cost function is inspired by FPTuner and does not consider cast operations.
- *Absolute errors cost* uses the static roundoff error, with smaller values representing a higher cost. A smaller round-off usually implies larger data types, which should correlate with a higher execution time.

We evaluate our cost functions experimentally on complete examples (see section 6). For each example function, we first generate 42 random precision assignments and their corresponding programs in Scala. We calculate the cost of each with all cost functions and also benchmark the actual running time. We use Scalometer [36] for benchmarking (see section 6). We are interested in distinguishing pairs of mixed-precision assignments, thus for each benchmark program, we create pairs of all the randomly generated precision assignments. Then we count how often each static cost function can correctly distinguish which of the two assignments is faster, where the ‘ground truth’ is given by the benchmarked running times. The following table summarizes the results of our cost function evaluation. The rows ‘32 - 128’ and ‘32 - 64’ and give the proportion of correctly distinguished pairs of type assignments with the random types selected from single, double and quad and single and double precision, respectively.

precisions	bench	simple	opCount	errors
32 - 128	0.7692	<b>0.8204</b>	0.8106	0.5871
32 - 64	<b>0.6416</b>	0.5889	0.5477	0.5462

Given these results, we choose a two-pronged approach for floating-point arithmetic: whenever quad precision may appear (e.g. during the first round of delta-debugging), we use the naive cost function. Once no more quad precision appears, we use the benchmarked one (e.g. during the second round of delta-debugging for benchmarks which do not require quad precision). For optimizing fixed-point arithmetic we use the simple cost function.

## 6 IMPLEMENTATION AND EVALUATION

We have implemented Anton in the Scala programming language. Internal arithmetic operations are implemented with a rational data type to avoid roundoff errors and ensure soundness. Apart from the (optional) use of the Z3 SMT solver [17] for more accurate range computations, Anton does not have any external dependencies.

In this work, we focus on arithmetic kernels, and do not consider conditionals and loops. Our technique (as well as FPTuner’s) can be extended to conditionals by considering individual paths separately as well as to loops by optimizing the loop *body* and thus reducing it to straight-line code. The challenge currently lies in the sound roundoff error estimation, which is known to be hard and expensive [13, 20], and is largely orthogonal to the focus of this paper. Our error computation method can also be extended to transcendental functions [11] and we plan to implement this extension in the future.

We are not aware of any tool which combines rewriting and mixed-precision tuning and which supports both floating-point as well as fixed-point arithmetic. We experimentally compare Anton with FPTuner [9], which is the only other tool for *sound* (floating-point) mixed-precision tuning. FPTuner reduces mixed-precision tuning to an optimization problem. The optimization objective in FPTuner is performance, but it is up to the user to provide optimization constraints, e.g. in form of a maximum number of cast operations. FPTuner also supports ganging of operators where the user can specify constraints that limit specific operations to have the same precision, which can be useful for vectorization. Adding these kinds of constraints to our approach is straight-forward. However, this manual optimization requires user expertise, and we have chosen to focus on automated techniques and the combination of mixed-precision and rewriting. As section 5 shows, already determining the cost function without ganging is challenging.

As FPTuner only supports floating-point arithmetic and fixed-point arithmetic is most useful in combination with specialized hardware, which is beyond the scope of this paper, we perform the experimental evaluation here for floating-point arithmetic only.

We do not compare against Precimonious directly, since Anton uses the same search algorithm internally. We would thus be merely comparing sound and unsound error analyses, which in our view is not meaningful.

*Benchmarks.* We have experimentally evaluated our approach and tool on a number of standard finite-precision verification benchmarks [12, 14, 28]. The benchmarks rigidBody, invertedPendulum and traincar are embedded controllers; bsplines, sine, and sqrt are examples of functions also used in the embedded domain (e.g. sine approximates the transcendental function whose library function is often not available, or expensive). The benchmarks doppler, turbine, himmilbeau and kepler are from the scientific computing domain. Table 1 lists the number of arithmetic operations and variables for each benchmark. An asterisk (\*) marks nonlinear benchmarks. To evaluate scalability, we also include four ‘unrolled’ benchmarks (marked by ‘2x’ and ‘3x’), where we double (or triple) the arithmetic operation count, as well as the number of input variables.

Which mixed-precision assignment is possible crucially depends on the maximum allowed error. None of the standard benchmarks come with suitable bounds since the focus until now has been on uniform precision. We follow FPTuner in defining suitable error bounds for our benchmarks. For each original example program, we first compute roundoff errors for uniform single and double precision. Slightly rounded up, these are the error bounds for benchmarks denoted by  $F$  and  $D$  respectively. From these we generate error bounds which are multiples of 0.5, 0.1 and 0.01 of these, denoted by  $F_{0.5}$ ,  $F_{0.1}$ , etc. That is, we create benchmarks whose error bounds are half, an order of magnitude and two orders of magnitude smaller than in uniform precision. This corresponds to a scenario where uniform precision is just barely not enough and we would like to avoid the next higher uniform precision.

*Experimental Setup.* We have performed all experiment on a Linux desktop computer with Intel Xeon 3.30GHz and 32GB RAM. For benchmarking, we use Anton’s programs generated in Scala (version 2.11.6) and translate FPTuner’s output into Scala. This translation is done automatically by a tool we wrote, and does not

benchmark	ops - vars	FPTuner	Anton-mixed	Anton-full
bspline2*	10 - 1	4m 56s	34s	50s
doppler*	8 - 3	12m 48s	1m 8s	5m 4s
himmilbeau*	15 - 2	9m 7s	44s	1m 21s
invPendulum	7 - 4	3m 47s	32s	45s
kepler0*	15 - 6	19m 17s	43s	1m 2s
kepler1*	24 - 4	1h 26m 3s	2m 17s	2m 9s
kepler2*	36 - 6	1h 52m 38s	3m 36s	4m 22s
rigidBody1*	7 - 3	4m 45s	28s	36s
rigidBody2*	14 - 3	8m 0s	43s	1m 3s
sine*	18 - 1	9m 10s	1m 9s	3m 36s
sqroot*	14 - 1	4m 33s	40s	1m 11s
traincar	28 - 14	17m 17s	1m 13s	2m 11s
turbine1*	14 - 3	5m 15s	1m 22s	3m 56s
turbine2*	10 - 3	4m 41s	58s	2m 52s
turbine3*	14 - 3	4m 23s	1m 21s	3m 44s
kepler2 (2x)	73 - 12	15h 36m 59s	7m 57s	9m 40s
rigidbody2 (3x)	44 - 9	58m 55s	5m 33s	3m 44s
sine (3x)	56 - 3	22m 40s	8m 20s	13m 57s
traincar (2x)	57 - 28	33m 19s	4m 32s	5m 48s

Table 1: Optimization times of Anton and FPTuner

affect the results, as FPTuner is platform independent. We use the Scalometer tool [36] (version 0.7) for benchmarking, which first warms up the JVM and detects steady-state execution *after* the Just-In-Time compiler has run, and then benchmarks the function as it is run effectively in native compiled code. We use the `@strictfp` annotation to ensure that the floating-point operations are performed exactly as specified in the program (otherwise error bounds cannot be guaranteed). We intentionally choose this setup to benchmark the mixed-precision assignments produced by Anton and FPTuner and not compiler optimization effects, which are out of scope of this paper.

**Optimization Time.** Table 1 compares the execution times of Anton and FPTuner themselves. For Anton, we report the times for mixed-precision tuning only without rewriting, which corresponds to the functionality that FPTuner provides, as well as the time for full optimization, i.e. rewriting and mixed-precision tuning.

For each tool, we measured the running time 5 times for each benchmark variant separately with the `bash time` command, recording the average real time. In the table, we show the aggregated time for all the variants of a benchmark, e.g. the total time for the  $F$ ,  $F_{0.5}$ ,  $F_{0.1}$ , ... variants of one benchmark together. These times are end-to-end, i.e. they include all phases as well as occasional timeouts by the backend solvers (Gelpia [3] for FPTuner and Z3 for Anton). Anton is faster than FPTuner for all benchmarks, even with rewriting included, and often by large factors. We suspect that this is due to the fact that FPTuner is solving a global optimization problem, which is known to be hard.

**Performance Improvements.** To evaluate the effectiveness of Anton we have performed end-to-end performance experiments. For this, we benchmark each generated optimized program five times with Scalometer and use the average running time. Each run of a program consists of 100 000 executions on random inputs. Then, for each mixed-precision variant we compare its running time against

the corresponding uniform-precision program and report the relative improvement (i.e. mixed-precision running time/ uniform running time). Corresponding here means the smallest uniform precision which would satisfy the error bound, e.g. for the  $F_{0.5}$  benchmark, the smallest uniform precision satisfying this bound is double precision.

Table 2 shows the relative running time improvements for Anton with only mixed-precision tuning, with only rewriting and with both rewriting and mixed-precision tuning enabled as well as for FPTuner. We show the average speedups over all variants of a benchmark (row). Bold values mark performance improvements above 5% and underlined values those with over 5% slowdowns.

Overall, we can see that the biggest speedups occur for the  $F_{0.5}$  and  $D_{0.5}$  benchmarks, as expected. The very low values (e.g. 0.07 in Table 2b for `train4-st8- $D_{0.5}$` ) result from rewriting reducing the roundoff error sufficiently for uniform precision being enough (the baseline comparison running time is however the higher uniform precision). In the case of FPTuner, the very low and very high values are caused by different characteristics of the error analyses. As a consequence, FPTuner is able to compute smaller roundoff errors than Anton for some benchmarks (and assigning uniform double precision instead of mixed), while for others it computes bigger roundoff errors, where it cannot show that double precision is sufficient, whereas Anton can.

Comparing FPTuner with Anton’s mixed-precision tuning only (Table 2a), we observe that Anton is more conservative, both with performance improvements but also with slowdowns, increasing the execution time only rarely. Considering Anton’s significantly better optimization time, Anton provides an interesting tradeoff between mixed-precision tuning and efficiency.

Comparing the speedups obtained by mixed-precision tuning and rewriting, we note that both techniques are successful in improving the performance, though the effect of rewriting is modest. When the two techniques are combined, we obtain the biggest performance improvements, which are furthermore more than just the sum of both. Note that rewriting could also be combined with (i.e. run before) FPTuner’s mixed-precision tuning.

## 7 RELATED WORK

**Rewriting.** An alternative approach to rewriting was presented by Damouche et al. [10] which relies on a greedy search and an abstract domain which represents possible expression rewritings together with a static error analysis similar to ours. The tool Herbie [33] performs a greedy hill-climbing search guided by a dynamic error evaluation function, and as such cannot provide sound error bounds. It is geared more towards correcting catastrophic cancellations, by employing an ‘error localization’ function which pin-points an operation which commits a particularly large round-off error and then targets the rewriting rules at that part of the expression. It would be interesting in the future to compare these different search techniques for rewriting.

**Mixed-precision Tuning in HPC.** Mixed-precision is especially important in HPC applications, because floating-point arithmetic is widely used. Lam et al. introduced a binary instrumentation tool together with a breadth-first search algorithm to help programmers search for suitable mixed-precision programs [26]. This work was



benchmark	F	$F_{0.5}$	$F_{0.1}$	$F_{0.01}$	D	$D_{0.5}$	$D_{0.1}$	$D_{0.01}$	Q	avrg
bspline2	1.01	<b>0.55</b>	1.00	1.00	1.00	1.00	1.00	1.00	1.00	<b>0.95</b>
doppler	0.97	<b>0.89</b>	0.96	<b>0.95</b>	0.96	1.01	0.99	1.00	0.99	0.97
himmilbeau	0.98	<b>0.95</b>	1.02	0.98	1.02	<b>0.61</b>	1.04	1.00	1.00	0.96
invPend.	0.98	1.02	1.01	0.98	0.99	<b>0.85</b>	1.00	1.01	1.00	0.98
kepler0	1.00	<u>1.07</u>	1.00	1.00	1.00	<b>0.85</b>	0.97	1.00	1.00	0.99
kepler1	1.00	<b>0.90</b>	<b>0.95</b>	0.99	0.98	<b>0.89</b>	<u>1.10</u>	1.00	1.00	0.98
kepler2	1.00	1.02	<b>0.93</b>	1.00	1.00	<b>0.85</b>	<u>1.14</u>	1.00	1.00	0.99
rigidBody1	1.04	1.02	1.02	1.00	0.97	<b>0.72</b>	<b>0.94</b>	1.00	0.99	0.97
rigidBody2	0.97	<b>0.94</b>	1.01	1.04	1.01	0.98	<u>1.16</u>	1.00	1.00	1.01
sine	1.01	<b>0.64</b>	1.00	0.99	1.00	<u>1.24</u>	1.00	1.00	1.00	0.99
sqroot	1.02	<b>0.84</b>	0.99	1.01	1.00	<b>0.59</b>	1.00	1.00	1.00	<b>0.94</b>
traincar	0.98	0.97	0.98	1.01	1.00	<b>0.61</b>	<b>0.61</b>	<b>0.87</b>	1.00	<b>0.89</b>
turbine1	1.00	<b>0.63</b>	1.00	1.01	1.01	<u>1.18</u>	1.00	1.00	1.00	0.98
turbine2	0.98	<b>0.66</b>	0.98	1.01	0.98	<b>0.77</b>	1.00	1.00	1.00	<b>0.93</b>
turbine3	1.00	<b>0.62</b>	0.99	0.99	0.99	<u>1.18</u>	1.00	1.01	1.01	0.98
kepler2 (2x)	0.98	<b>0.91</b>	0.99	0.99	1.00	<b>0.65</b>	<u>1.13</u>	1.01	1.01	0.96
rigidBody2(3x)	0.99	1.02	1.00	0.97	0.97	<b>0.74</b>	<u>1.09</u>	1.00	1.00	0.97
sine (3x)	1.00	<b>0.75</b>	1.00	1.00	1.00	<b>0.69</b>	1.00	1.00	1.00	<b>0.94</b>
traincar (2x)	<u>1.13</u>	<u>1.05</u>	<u>1.05</u>	<u>1.12</u>	<u>1.14</u>	<b>0.61</b>	<b>0.87</b>	<b>0.93</b>	1.00	0.99

(a) Anton - mixed-precision tuning only

benchmark	F	$F_{0.5}$	$F_{0.1}$	$F_{0.01}$	D	$D_{0.5}$	$D_{0.1}$	$D_{0.01}$	Q	avrg
bspline2	0.99	0.99	0.99	0.99	0.98	<b>0.90</b>	<b>0.90</b>	<b>0.90</b>	<b>0.90</b>	<b>0.95</b>
doppler	1.02	<b>0.90</b>	<u>1.33</u>	<u>1.32</u>	<u>1.33</u>	<b>0.10</b>	1.01	1.02	1.01	1.01
himmilbeau	0.96	0.98	0.98	0.99	0.98	0.99	0.99	0.99	0.99	0.98
invPend.	1.03	0.99	1.00	1.03	1.03	<b>0.95</b>	<b>0.95</b>	<b>0.95</b>	0.96	0.99
kepler0	0.98	1.02	0.99	1.00	0.99	0.97	0.97	0.98	0.97	0.99
kepler1	0.99	<b>0.87</b>	<b>0.87</b>	<b>0.87</b>	<b>0.87</b>	1.00	1.00	1.00	1.00	<b>0.94</b>
kepler2	0.96	<b>0.92</b>	<b>0.92</b>	<b>0.95</b>	<b>0.93</b>	0.99	0.99	0.99	0.99	0.96
rigidBody1	0.99	0.99	1.01	1.01	0.99	<b>0.93</b>	<b>0.93</b>	<b>0.93</b>	<b>0.93</b>	0.97
rigidBody2	0.96	<b>0.88</b>	<b>0.89</b>	<b>0.90</b>	<b>0.90</b>	0.98	0.98	0.98	0.98	<b>0.94</b>
sine	0.99	<u>1.08</u>	<u>1.08</u>	<u>1.08</u>	<u>1.09</u>	0.98	0.98	0.98	0.98	1.03
sqroot	0.96	<b>0.95</b>	<b>0.93</b>	<b>0.92</b>	<b>0.93</b>	0.97	0.97	0.98	0.98	<b>0.95</b>
traincar	<b>0.89</b>	<b>0.85</b>	<b>0.89</b>	<b>0.89</b>	<b>0.91</b>	<b>0.07</b>	1.02	1.02	1.02	<b>0.84</b>
turbine1	1.02	0.97	0.97	0.96	0.96	<u>1.06</u>	<u>1.06</u>	<u>1.06</u>	<u>1.06</u>	1.01
turbine2	<b>0.94</b>	<b>0.94</b>	<b>0.94</b>	<b>0.94</b>	<b>0.95</b>	0.99	0.99	0.98	0.99	0.96
turbine3	1.01	0.96	0.97	<b>0.95</b>	<b>0.95</b>	<u>1.07</u>	<u>1.07</u>	<u>1.07</u>	<u>1.07</u>	1.01
kepler2 (2x)	<b>0.89</b>	<b>0.78</b>	<b>0.86</b>	<b>0.87</b>	<b>0.88</b>	0.96	0.98	1.00	0.96	<b>0.91</b>
rigidBody2(3x)	1.04	0.96	<b>0.94</b>	<b>0.94</b>	0.98	1.00	0.99	1.00	1.01	0.99
sine (3x)	<b>0.93</b>	<u>1.05</u>	<b>0.91</b>	<b>0.90</b>	<b>0.90</b>	0.99	0.99	1.00	0.98	0.96
traincar (2x)	0.98	<b>0.90</b>	0.98	0.99	1.01	<b>0.09</b>	1.03	1.02	1.02	<b>0.89</b>

(b) Anton - rewriting only

benchmark	F	$F_{0.5}$	$F_{0.1}$	$F_{0.01}$	D	$D_{0.5}$	$D_{0.1}$	$D_{0.01}$	Q	avrg
bspline2	1.04	<b>0.50</b>	0.99	0.97	0.98	<b>0.77</b>	<b>0.90</b>	<b>0.91</b>	<b>0.91</b>	<b>0.88</b>
doppler	1.03	<b>0.87</b>	0.98	<u>1.31</u>	<u>1.31</u>	<b>0.10</b>	<u>1.18</u>	1.02	1.01	0.98
himmilbeau	0.99	0.96	1.01	1.00	0.98	<b>0.60</b>	<u>1.07</u>	0.99	0.99	<b>0.95</b>
invPend.	0.99	0.99	0.99	0.97	0.97	<b>0.68</b>	0.97	<b>0.95</b>	<b>0.95</b>	<b>0.94</b>
kepler0	<b>0.94</b>	<u>1.05</u>	0.99	0.96	0.96	<b>0.55</b>	1.00	0.98	0.98	<b>0.93</b>
kepler1	0.98	<b>0.91</b>	<b>0.87</b>	<b>0.90</b>	<b>0.87</b>	<b>0.89</b>	0.98	1.00	1.00	<b>0.93</b>
kepler2	0.96	1.04	<b>0.94</b>	<b>0.93</b>	<b>0.93</b>	<b>0.55</b>	<u>1.08</u>	0.98	0.98	<b>0.93</b>
rigidBody1	1.00	1.02	0.97	0.96	0.99	<b>0.57</b>	1.04	<b>0.93</b>	<b>0.94</b>	<b>0.94</b>
rigidBody2	0.97	0.98	<b>0.90</b>	<b>0.88</b>	<b>0.88</b>	<b>0.61</b>	<u>1.16</u>	0.98	0.98	<b>0.93</b>
sine	0.98	<b>0.65</b>	<u>1.09</u>	<u>1.10</u>	<u>1.09</u>	<u>1.26</u>	0.99	0.98	0.98	1.01
sqroot	0.97	<b>0.87</b>	<b>0.95</b>	<b>0.92</b>	<b>0.93</b>	<b>0.64</b>	1.02	0.97	0.97	<b>0.92</b>
traincar	<b>0.88</b>	<b>0.85</b>	<b>0.86</b>	<b>0.89</b>	<b>0.91</b>	<b>0.07</b>	<b>0.66</b>	<b>0.90</b>	1.02	<b>0.78</b>
turbine1	<b>0.99</b>	<b>0.68</b>	0.96	0.96	0.96	<u>1.17</u>	<u>1.06</u>	<u>1.06</u>	<u>1.06</u>	0.99
turbine2	0.98	<b>0.67</b>	<b>0.94</b>	<b>0.95</b>	<b>0.95</b>	<b>0.74</b>	0.99	0.99	1.00	<b>0.91</b>
turbine3	1.00	<b>0.70</b>	0.96	0.96	<b>0.95</b>	<u>1.16</u>	<u>1.07</u>	<u>1.07</u>	<u>1.07</u>	0.99
kepler2 (2x)	<b>0.86</b>	<b>0.79</b>	<b>0.86</b>	0.96	<b>0.89</b>	<b>0.80</b>	<u>1.05</u>	1.00	0.97	<b>0.91</b>
rigidBody2(3x)	1.00	1.02	<b>0.94</b>	<b>0.95</b>	0.96	<b>0.55</b>	0.98	1.01	0.99	<b>0.93</b>
sine (3x)	<b>0.93</b>	<b>0.61</b>	<b>0.95</b>	<b>0.90</b>	<b>0.89</b>	<b>0.46</b>	1.03	0.99	0.99	<b>0.86</b>
traincar (2x)	0.96	<b>0.91</b>	<b>0.93</b>	0.97	1.00	<b>0.08</b>	<b>0.91</b>	0.99	1.03	<b>0.87</b>

(c) Anton - full optimization

benchmark	F	$F_{0.5}$	$F_{0.1}$	$F_{0.01}$	D	$D_{0.5}$	$D_{0.1}$	$D_{0.01}$	Q	avrg
bspline2	1.00	<b>0.53</b>	1.01	1.00	1.00	<b>0.65</b>	<u>1.21</u>	<u>1.21</u>	<u>1.22</u>	0.98
doppler	0.99	<b>0.87</b>	1.03	1.01	1.00	<u>0.07</u>	<u>1.10</u>	<u>1.57</u>	<u>1.62</u>	1.03
himmilbeau	0.98	<b>0.88</b>	<u>1.22</u>	0.99	1.01	<u>0.19</u>	<b>0.92</b>	<u>1.06</u>	<u>1.05</u>	<b>0.92</b>
invPend.	<u>1.05</u>	1.01	1.04	0.98	0.99	<b>0.61</b>	0.98	<u>1.07</u>	<u>1.05</u>	0.98
kepler0	<u>1.06</u>	<u>1.06</u>	<u>1.07</u>	1.01	1.01	<b>0.49</b>	1.01	<u>1.10</u>	<u>1.10</u>	0.99
kepler1	1.03	1.02	<u>1.22</u>	0.99	<u>5.46</u>	<b>0.57</b>	0.99	<u>1.07</u>	<u>1.07</u>	<u>1.49</u>
kepler2	0.98	<u>1.05</u>	<u>1.11</u>	<u>1.24</u>	<u>3.03</u>	<b>0.64</b>	<b>0.88</b>	1.04	1.04	<u>1.22</u>
rigidBody1	<u>1.08</u>	1.03	0.99	0.96	<u>4.66</u>	<b>0.61</b>	1.03	<u>1.21</u>	<u>1.21</u>	<u>1.42</u>
rigidBody2	<u>1.08</u>	<b>0.95</b>	<u>1.18</u>	1.02	<u>5.52</u>	<b>0.56</b>	<b>0.57</b>	<b>0.89</b>	<u>1.13</u>	<u>1.43</u>
sine	1.01	<b>0.43</b>	<b>0.86</b>	<u>0.93</u>	1.00	<b>0.20</b>	<b>0.48</b>	<b>0.68</b>	1.03	<b>0.74</b>
sqroot	<u>1.14</u>	<b>0.83</b>	<u>1.22</u>	<u>1.12</u>	<u>4.86</u>	<b>0.40</b>	<b>0.68</b>	<b>0.94</b>	<u>1.24</u>	<u>1.38</u>
traincar	1.01	<b>0.90</b>	<b>0.94</b>	<b>0.92</b>	<u>4.55</u>	<b>0.36</b>	<b>0.37</b>	<b>0.37</b>	1.04	<u>1.16</u>
turbine1	1.00	<b>0.56</b>	<b>0.88</b>	<b>0.95</b>	1.00	<b>0.07</b>	<b>0.86</b>	<b>0.92</b>	<u>1.18</u>	<b>0.82</b>
turbine2	1.00	<b>0.61</b>	<b>0.79</b>	0.98	1.00	<b>0.09</b>	<b>0.92</b>	<u>1.13</u>	<u>1.12</u>	<b>0.85</b>
turbine3	1.01	<b>0.58</b>	0.96	<b>0.92</b>	1.00	<b>0.07</b>	<b>0.68</b>	<b>0.91</b>	<u>1.18</u>	<b>0.81</b>
kepler2 (2x)	0.99	0.97	<u>1.13</u>	<u>1.09</u>	<u>2.91</u>	<b>0.55</b>	<b>0.86</b>	crash	1.03	<u>1.19</u>
rigidBody2(3x)	1.03	1.03	<u>1.13</u>	<u>1.08</u>	<u>4.97</u>	<b>0.45</b>	<b>0.61</b>	<b>0.83</b>	<u>1.13</u>	<u>1.36</u>
sine (3x)	0.99	<b>0.53</b>	<b>0.85</b>	1.02	<u>2.20</u>	<b>0.19</b>	<b>0.45</b>	<b>0.69</b>	1.02	<b>0.88</b>
traincar (2x)	<u>1.11</u>	0.96	0.99	0.99	<u>5.26</u>	<b>0.50</b>	<b>0.51</b>	<b>0.51</b>	1.03	<u>1.32</u>

(d) FPTuner

Table 2: Relative performance improvements for Anton and FPTuner

later extended to perform a sensitivity analysis [25] based on a more fine-grained approach. The Precimonious project [39, 40], whose delta-debugging algorithm we adapt, targets HPC kernels and library functions and performs automated mixed-precision tuning. These projects have in common that the roundoff error verification is performed dynamically on a limited number of inputs and thus does not provide guarantees. In contrast, our technique produces sound results, but is targeted at smaller programs and kernels which can be verified statically.

*Autotuning.* Another way to improve the performance of (numerical) computations is autotuning, which performs low-level transformations of the program in order to find one which empirically executes most efficiently. Traditionally, the approaches have

been semantics preserving [37, 44], but recently also non-semantics preserving ones have been proposed in the space of approximate computing [42]. These techniques represent another avenue for improving performance, but do not optimize mixed-precision.

*Bitlength Optimization in Embedded Systems.* In the space of embedded systems, much of the attention so far has focused on fixed-point arithmetic and the optimization of bitlengths, which can be viewed as selecting data types. A variety of static and dynamic approaches have been applied. For instance, Gaffar et al. considers both fixed-point and floating-point programs and uses automatic differentiation for a sensitivity analysis [18]. Mallik et al. present an optimal bit-width allocation for two variables and a greedy heuristic for more variables, and rely on dynamic error evaluation [30].

Unlike our approach, these two techniques cannot provide sound error bounds. Sound techniques have also been applied for both the range and the error analysis for bitwidth optimization, for instance in [24, 27, 32, 34] and Lee et al. provide a nice overview of static and dynamic techniques [27]. For optimization, Lee et al. have used simulated annealing as the search technique [27]. A detailed comparison of delta-debugging and e.g. simulated annealing would be very interesting in the future. We note that our technique is general in that it is applicable to both floating-point as well as fixed-point arithmetic, and the first to combine bitwidth optimization for performance with rewriting.

*Finite-precision Verification.* There has been considerable interest in static and sound numerical error estimation for finite-precision programs with several tools having been developed: Rosa [12], Fluctuat [19], FPTaylor [43] (which FPTuner is based on) and Real2Float [28]. The accuracies of these tools are mostly comparable [13], so that any of the underlying techniques could be used in our approach for the static error function. More broadly related are abstract interpretation-based static analyses, which are sound wrt. floating-point arithmetic [4, 8, 23]. These techniques can prove the absence of runtime errors, such as division-by-zero, but cannot quantify roundoff errors. Floating-point arithmetic has also been formalized in theorem provers such as Coq [6] [15] and HOL Light [22], and entire numerical programs have been proven correct and accurate within these [5, 38]. Most of these verification efforts are to a large part manual, and do not perform mixed-precision tuning. FPTaylor uses HOL Light and Real2Float Coq to generate certificates of correctness of the error bounds it computes. We believe that this facility could be extended to the mixed-precision case – this, however, would come after the tuning step, and hence these efforts are largely orthogonal.

Floating-point arithmetic has also been formalized in an SMT-lib [41] theory and SMT solvers exist which include floating-point decision procedures [7, 17]. These are, however, not suitable for roundoff error quantification, as a combination with the theory of reals would be necessary which does not exist today.

## 8 CONCLUSION

We have presented a fully automated technique which combines rewriting and sound mixed-precision tuning for improving the performance of arithmetic kernels. While each of the two parts is successful by itself, we have empirically demonstrated that their careful combination is more than just the sum of the parts. Furthermore, our mixed-precision tuning algorithm presents an interesting tradeoff as compared to state-of-the-art between efficiency of the algorithm and performance improvements generated.

## REFERENCES

- [1] A. Anta, R. Majumdar, I. Saha, and P. Tabuada. 2010. Automatic Verification of Control System Implementations. In *EMSOFT*.
- [2] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson. 2015. *C++/Fortran-90 double-double and quad-double package*. Technical Report. <http://crd-legacy.lbl.gov/~dhbailey/mpdist/>
- [3] M. S. Baranowski and I. Briggs. 2016. Global Extrema Locator Parallelization for Interval Arithmetic (Gelpia). <https://github.com/soarlab/gelpia>. (2016).
- [4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2003. A Static Analyzer for Large Safety-Critical Software. In *PLDI*.
- [5] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis. 2013. Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning* 50, 4 (2013), 423–456.
- [6] S. Boldo and G. Melquiond. 2011. Floq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *ARITH*.
- [7] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. 2013. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design* 45, 2 (Dec. 2013), 213–245.
- [8] L. Chen, A. Miné, and P. Cousot. 2008. A Sound Floating-Point Polyhedra Abstract Domain. In *APLAS*.
- [9] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, I. Briggs, M. S. Baranowski, and A. Solovyev. 2017. Rigorous Floating-point Mixed Precision Tuning. In *POPL*.
- [10] N. Damouche, M. Martel, and A. Chapoutot. 2015. Intra-procedural Optimization of the Numerical Accuracy of Programs. In *FMICS*.
- [11] E. Darulova and V. Kuncak. 2011. Trustworthy Numerical Computation in Scala. In *OOPSLA*.
- [12] E. Darulova and V. Kuncak. 2014. Sound Compilation of Reals. In *POPL*.
- [13] E. Darulova and V. Kuncak. 2017. Towards a Compiler for Reals. *ACM TOPLAS* 39, 2 (2017).
- [14] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha. 2013. Synthesis of Fixed-point Programs. In *EMSOFT*.
- [15] M. Daumas, L. Rideau, and L. Théry. 2001. A Generic Library for Floating-Point Numbers and Its Application to Exact Computing. In *TPHOLS*.
- [16] L. H. de Figueiredo and J. Stolfi. 2004. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms* 37, 1-4 (2004).
- [17] L. De Moura and N. Bjørner. 2008. Z3: an efficient SMT solver. In *TACAS*.
- [18] A. A. Gaffar, O. Mencer, W. Luk, and P. Y. K. Cheung. 2004. Unifying Bit-Width Optimisation for Fixed-Point and Floating-Point Designs. *FCCM* (2004).
- [19] E. Goubault and S. Putot. 2011. Static Analysis of Finite Precision Computations. In *VMCAI*.
- [20] E. Goubault and S. Putot. 2013. Robustness Analysis of Finite Precision Implementations. In *APLAS*.
- [21] ISO/IEC. 2008. *Programming languages — C — Extensions to support embedded processors*. Technical Report ISO/IEC TR 18037.
- [22] C. Jacobsen, A. Solovyev, and G. Gopalakrishnan. 2015. A Parameterized Floating-Point Formalization in HOL Light. *Electronic Notes in Theoretical Computer Science* 317 (2015), 101–107.
- [23] B. Jeannet and A. Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*.
- [24] A. B. Kinsman and N. Nicolici. 2009. Finite Precision Bit-Width Allocation using SAT-Modulo Theory. In *DATE*.
- [25] M. O. Lam and J. K. Hollingsworth. 2016. Fine-grained floating-point precision analysis. *Intl. Journal of High Performance Computing Applications* (June 2016).
- [26] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre. 2013. Automatically Adapting Programs for Mixed-precision Floating-point Computation. In *ICS*.
- [27] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. 2006. Accuracy-Guaranteed Bit-Width Optimization. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 25, 10 (2006), 1990–2000.
- [28] V. Magron, G. A. Constantinides, and A. F. Donaldson. 2015. Certified Roundoff Error Bounds Using Semidefinite Programming. *CoRR* abs/1507.03331 (2015).
- [29] R. Majumdar, I. Saha, and M. Zamani. 2012. Synthesis of Minimal-error Control Software. In *EMSOFT*.
- [30] A. Mallik, D. Sinha, P. Banerjee, and H. Zhou. 2007. Low-Power Optimization by Smart Bit-Width Allocation in a SystemC-Based ASIC Design Environment. *IEEE Trans. on CAD of Integrated Circuits and Systems* (2007).
- [31] R. Moore. 1966. *Interval Analysis*. Prentice-Hall.
- [32] W. G. Osborne, R. C. C. Cheung, J. Coutinho, W. Luk, and O. Mencer. 2007. Automatic Accuracy-Guaranteed Bit-Width Optimization for Fixed and Floating-Point Systems. In *Field Programmable Logic and Applications*. 617–620.
- [33] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *PLDI*.
- [34] Y. Pang, K. Radecka, and Z. Zilic. 2011. An Efficient Hybrid Engine to Perform Range Analysis and Allocate Integer Bit-widths for Arithmetic Circuits. In *ASPDAC*.
- [35] R. Poli, W. B. Langdon, and N. F. McPhee. 2008. *A Field Guide to Genetic Programming*. Lulu Enterprises.
- [36] A. Prokopec. 2012. ScalaMeter. <https://scalameter.github.io/>. (2012).
- [37] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. R. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson. 2004. Spiral - A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *IJHPCA* 18, 1 (2004), 21–45.
- [38] T. Ramanathan, P. Mountcastle, B. Meister, and R. Lethin. 2016. A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In *CPP*.
- [39] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough. 2016. Floating-Point Precision Tuning Using Blame Analysis. In *ICSE*.

- [40] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. 2013. Precimonious: Tuning Assistant for Floating-point Precision. In *SC*.
- [41] P. Rümmer and T. Wahl. 2010. An SMT-LIB Theory of Binary Floating-Point Arithmetic. In *SMT*.
- [42] E. Schkufza, R. Sharma, and A. Aiken. 2014. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *PLDI*.
- [43] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. 2015. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *FM*.
- [44] R. Vuduc, J. W. Demmel, and J. A. Biles. 2004. Statistical Models for Empirical Search-Based Performance Tuning. *Int. J. High Perform. Comput. Appl.* 18, 1 (Feb. 2004), 65–94.
- [45] A. Zeller and R. Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.