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, widely used in embedded systems for numerical calculations, faces 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.

***Index Terms*—floating-point; mixed-precision; rewriting; static analysis;**

I. Introduction

Finite-precision computations, used for much of the numerical computations in embedded applications, face an inherent tradeoff between accuracy and efficiency due to 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 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.

We propose the **novel combination of mixed-precision tuning with rewriting** in a *fully automated* technique for performance optimization of finite-precision arithmetic kernels, such as those generated by Simulink. 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). In contrast to state-of-the-art, our approach can handle both floating-point as well as fixed-point arithmetic kernels.

a) Mixed-precision Tuning: In order to save valuable resources like time, memory or energy, we would like to choose the smallest data type that 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, especially on resource constraint embedded platforms where arithmetic often relies on software implementations. 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 [1]–[4].

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, mixed-precision roundoff errors and performance is often unintuitive. For instance, changing one particular operation to lower precision may produce a smaller roundoff error than changing two (other) operations. Additionally, mixed-precision introduces cast operations, which may increase the running time, even though the accuracy decreases.

In high-performance computing (HPC), mixed-precision tuning approaches [5], [6] use dynamic techniques to estimate roundoff errors. For safety-critical embedded applications such unsound approaches are inherently problematic as they do not provide any accuracy guarantees. The FPTuner tool [7] 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 entirely automated. Furthermore, its tuning time can be prohibitively large.

b) 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 [8], [9]. However, optimizing for accuracy may increase the number of arithmetic operations and thus the execution time and consequently also energy consumption.

c) Combining Mixed-Precision Tuning and Rewriting: In this paper, we propose the first exploration of rewriting *together with* mixed-precision tuning. We show that if such a combination is carefully designed, it is more successful than each part alone in improving the performance under guaranteed roundoff error bounds. 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 accuracy, it allows for more aggressive mixed-precision tuning. We extend an unsound mixed-precision tuning algorithm [6] from the HPC domain with sound static roundoff error analysis [2] as well as a novel static performance cost function to obtain a mixed-precision tuning technique which is both sound, fully automated and efficient.

We focus on arithmetic kernels, and do not consider conditionals or loops, which are challenging for sound roundoff error estimation [10], [11], and are largely orthogonal to the focus of this paper. Our technique is applicable and implemented in a tool called Daisy for both floating-point as well as fixed-point arithmetic. Our evaluation focuses on floating-point arithmetic; we leave a thorough investigation of specialized hardware implementations, which are required for efficient fixed-point arithmetic, for future work.

For floating-point arithmetic, we evaluate Daisy on standard benchmarks from embedded systems and scientific computing. We observe that rewriting alone improves performance by up to 17% (compared to uniform precision) and for some benchmarks even more by reducing roundoff errors sufficiently to enable a smaller *uniform* precision. Mixed-precision tuning improves performance by up to 45% and in combination with rewriting, Daisy improves performance by up to 54% (93% for those cases where rewriting enables lower uniform precision). We furthermore observe that it improves performance for more benchmarks than when using mixed-precision tuning or rewriting alone.

d) Contributions: To summarize, in this paper we present:

- an optimization procedure based on rewriting which takes into account both accuracy and performance,
- the first mixed-precision tuning technique which is sound, fully automated as well as efficient and scalable,
- a carefully designed and novel combination of rewriting and mixed-precision tuning, which provides more significant performance improvements than each of them alone,
- to show the non-triviality of the combination, we show empirically several alternate approaches which were not successful,
- an implementation in the tool Daisy, which generates optimized source programs in Scala and in C and which supports both floating-point as well as fixed-point arithmetic (available at github.com/malyzajko/daisy),
- an experimental comparison against state-of-the-art on a standard set of (embedded) arithmetic kernels.

While we focus here on performance, our algorithm is independent of the optimization objective, and with the corresponding cost function, memory or energy optimization are equally possible.

II. Overview and Key Ideas

The input to Daisy 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. Consider, for instance, the following nonlinear embedded controller [8]:

```
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 [12]. The function body consists of an arithmetic expression, with $+, -, *, /, \sqrt{}$, commonly used transcendental functions and possibly local variable declarations.

As output, Daisy 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. Daisy 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 [13]. Daisy 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 [8] with each point in the search space requiring an expensive evaluation, which is computationally infeasible. We thus choose to separate rewriting from mixed-precision tuning and further choose different efficient search techniques for each.

a) Step 1: Rewriting: Daisy 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.

Daisy'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 chosen for the fitness function can affect the result of rewriting, we empirically show that the effect is small (section IV).

This approach is heavily inspired by the algorithm presented in [8] which optimized fixed-point arithmetic expressions and 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)
```

Note that the magnitude of roundoff errors depends on the possible ranges of intermediate variables. Thus even small changes in the evaluation order can have large effects on these ranges and consequently also on the roundoff errors.

b) Step 2: Code Transformation: To facilitate mixed-precision tuning, Daisy performs two code transformations: constants as well as the results of each arithmetic operation are assigned to fresh variables. During phase 4, Daisy tunes the precision of exactly these variables. If not all arithmetic operations should be tuned, i.e. a more coarse grained mixed-precision is desired, then this step can be skipped.

c) Step 3: Range Analysis: Daisy 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, Daisy computes them only once for efficiency.

d) 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 [6], 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.

Unlike Precimonious, which evaluates the accuracy and performance of different mixed-precisions dynamically, Daisy 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, Daisy 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.

e) Step 5: Code Generation: Once mixed-precision tuning finds a suitable type configuration, Daisy 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.

III. Background

We first briefly review necessary background about finite-precision arithmetic and sound roundoff error estimation.

A. Floating-point Arithmetic

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

$$x \circ_{fl} y = (x \circ y)(1+\delta)\ ,\ \ |\delta| \leq \epsilon_m \tag{1}$$

where $\circ \in +, -, *, /$ and $\circ_{fl}$ denotes the respective floating-point version, and ϵ_m bounds the maximum relative error (2^{-24}

and 2^{-53} for single and double precision respectively). Unary minus and square root follow similarly. The same abstraction also applies to quad precision, which is usually implemented in software on top of standard double floating-point precision (e.g. [13]). For quad precision, we assume a machine epsilon of 2^{-113}. We further consider NaNs (not-a-number special values), infinities and ranges containing only denormal floating-point numbers to be errors and Daisy's error computation technique detects these automatically. The last check ensures that we soundly over-approximate subnormal roundoff errors, which do not follow Equation 1 (however it holds that the roundoff of the smallest normal number is larger than the roundoff of the largest subnormal).

B. 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. For more details please see [14], 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, which can be computed from the range of possible values at that operation.

C. Sound Roundoff Error Estimation

We build upon Rosa's static error computation [2], 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 finite-precision 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: $|x-\tilde{x}| = |x|*\delta, \quad \delta \leq \epsilon_m$ which follows from Equation 1. This definition extends to multivariate f component-wise.

The magnitude of absolute roundoff errors depends on the magnitude of values of expressions. Thus, if we want to accurately bound roundoff errors, we need to be able to bound the ranges of all intermediate expressions first. At a high level, Daisy computes error bounds using Rosa's forward data-flow analysis over the abstract syntax tree in two steps:

1) compute sound real-valued range bounds
2) using these ranges, propagate errors from subexpressions and compute the new worst-case roundoff errors.

For our rewriting procedure, since the ranges change for each rewritten expression, we compute both the ranges and the errors at the same time. For mixed-precision tuning, where the ranges remain constant, we can separate the computations entirely and only do the range computation only once.

Clearly, accurate range arithmetic is a main component in the error bound computation, and is known to be challenging, especially for nonlinear arithmetic [1]–[3]. Daisy supports (as Rosa) interval [15] and affine arithmetic [16] as well as a more accurate, but also more expensive combination of interval arithmetic and SMT [2]. For more details on these, please see the appendix.

FPTaylor takes an entirely different approach and formulates the estimation of roundoff errors as a global optimization problem. While it has been shown to produce often slightly tighter error bounds [3], [10], it is not applicable to fixed-point arithmetic. We thus choose the dataflow approach, which works for both floating-point as well as fixed-point arithmetic.

Finally, 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 [17]. For instance, when the range of $f(x)$ includes zero, relative errors are not well defined and this is often the case in practice.

IV. Rewriting Optimization

The goal of Daisy'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 Daisy.

A. Background: Genetic Search for Rewriting

An exhaustive search of all possible rewritings is computationally infeasible. Even for only linear arithmetic, the problem of finding an optimal order is NP-hard [8] and does not allow a divide-and-conquer or gradient-based method such that a heuristic and incomplete search becomes necessary.

The Xfp tool [8] used genetic programming [18], which 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 a pre-defined number copies of the initial expression—the size of the population. At every iteration, each new generation of the population is built by repeating the following steps until the new generation is full:

1) selecting an expression from the current population based on their fitness,
2) randomly mutating the selected expression,
3) evaluating the fitness of the new expression.

Using the so-called tournament selection for step 1, the algorithm picks four candidate expressions at random from the current population and chooses the one with the smallest error. This ensures that even expressions with bigger errors have a certain probability of contributing to the next iteration and thus enable the search to overcome local minima. Step 2 picks an AST node at random and applies one of the applicable rewrite rules, again at random. These rules are standard real-valued semantics preserving mathematical identities such as distributivity or associativity. Finally, step 3 uses a sound roundoff error estimation as the fitness function, where smaller roundoff errors denote fitter candidates. The output of the procedure after a pre-determined number of iterations is the

expression with the least roundoff error seen during the run of the search.

In this fashion, the algorithm explores different rewritings. The key idea is that the likelyhood 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.

B. Rewriting in Daisy

We instantiate the algorithm described above with a population of 30 expressions, 30 iterations and tournament selection for selecting which expressions to mutate. These are successful settings identified in Xfp. We do not use the crossover operation, because it had only limited effects. We further extend the rather limited set of mutation rules in Xfp with the more complete one used by the (unsound) rewriting optimization tool Herbie [9] (for a more detailed comparison, see section VII). All rules are based on real-valued mathematical identities, e.g.:

```
      a + (b + c) → (a + b) + c
      a * (b + c) → (a * b) + (a * c)
    (- a) + (- b) → -(a + b)
(a / b) + (c / d) → (a * d + b * c) / (b * d)
```

For the fitness function, we use the static error analysis as described in subsection III-C, with interval arithmetic for computing ranges and affine arithmetic for tracking errors. This provides a good accuracy-performance tradeoff, which is important as this fitness function is being called repeatedly. We furthermore extend rewriting to both floating-point as well as fixed-point arithmetic; Xfp only handled the latter.

The algorithm in Xfp aims to reduce 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 with respect to which precision to perform the rewriting as the final mixed-precision type assignment is not available at this point.

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.

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 Daisy performs rewriting, we have to choose some precision without knowing the final assignment.

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

We performed an experiment to validate this intuition, which we detail for space reasons in the appendix. In summary, it confirms that our choice to perform rewriting wrt. uniform double floating-point precision provides a good proxy for other (possibly mixed) precisions.

V. Sound Mixed-Precision Tuning

After rewriting, Daisy pre-processes expressions as described in step 2 in section II 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 [2]. We again first review previous work that we build upon before explaining Daisy's technique in detail.

A. Background: Delta-Debugging for Precision Tuning

Delta-debugging has been originally conceived in the context of software testing for identifying the smallest failing testcase [19]. Precimonious [6] adapts this algorithm for mixed-precision tuning of floating-point arithmetic programs. It takes as input:

- a list of variables to be tuned (τ)
- a list of other variables with their constant precision assignments (ϕ)
- 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 τ.

A partial sketch of the algorithm is depicted in Figure 1, where the boxes represent sets of variables in τ. For ease of presentation, consider the case where variables can be in single (32 bit) and double (64 bit) floating-point precision; we explain the generalization to more precisions later. The algorithm starts by assigning all variables in τ 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 τ 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

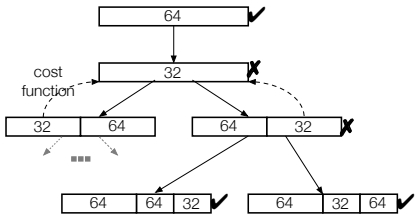

Fig. 1: Sketch of the delta-debugging algorithm

does not succeed, the algorithm splits τ into two equally sized lists τ_1 and τ_2 and recurses on each separately. When recursing on τ_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 τ_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.)

At this point, another optimization step is possible (not shown in Figure 1). For a type assignment which satisfies the error bound, for some τ and ϕ, the variables in τ will be assigned to lower precision and all variables in ϕ to higher precision. However, it is possible that only some variables in ϕ need to be in high precision. Hence, the algorithm divides ϕ into ϕ_1 and ϕ_2 and tries to lower the variables in each in turn, selecting the one which satisfies the error bound and has lowest cost.

The algorithm generalizes 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 ϕ. The optimization is then performed on the new τ considering the second and third highest precision, etc.

B. Mixed-Precison Tuning in Daisy

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

a) 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. Daisy uses a *sound static* roundoff error analysis, which is an extension of Rosa's uniform-precision error analysis from subsection III-C 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 Daisy 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 [10], by considering the loop body only.

Daisy currently supports mixed-precision tuning wrt. to floating-point arithmetic (between single, double and quad precision) or wrt. to fixed-point arithmetic (16 or 32 bitlengths). Since the error computation is parametric in the precisions, this list can be straight-forwardly extended.

b) Tuning All Variables: Unlike Precimonious, which optimizes only the precisions of declared variables, Daisy optimizes the precisions of all variables and intermediate expressions by automatically transforming the program prior to mixed-precision tuning into three-address form.

c) Semantics of Type Assignment: For an arithmetic operation in three-address form, e.g. `val z = x + y`, the precision of the arithmetic operation is determined by the precisions of its operands (`x, y`) as well as the variable that the result is assigned to (`z`). In general, we follow standard semantics, where the operation is performed in the highest operand precision, with one exception. For example, for the precision assignment $\{x \rightarrow \text{single}, y \rightarrow \text{single}, z \rightarrow \text{double}\}$, we choose the interpretation `z = x.toDouble + y.toDouble` instead of `z = (x + y).toDouble`, so that the operation is performed in the higher precision, thus loosing less accuracy. Our experiments have shown that this indeed provides better overall results.

d) Order of Variables in Optimization: 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.

e) Precision of Contants: 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 Daisy optimizes constants just like other variables.

C. 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. FPTuner, 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 each 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 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 roundoff usually implies larger data types, which should correlate with a higher execution time.

We evaluate our cost functions experimentally on complete examples (see section VI). 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 Scalameter [20] for benchmarking (see paragraph VI-0c).

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.

a) Experimental Results: The following table summarizes the results of our cost function evaluation. The rows '32 - 128' and '32 - 64' 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. Since quad precision is implemented in a software library, its effect on performance is larger and we thus perform two experiments, with and without quad precision included.

available precisions	simple	bench	opCount	absErrors
32 - 128	**0.8204**	0.7692	0.8106	0.5871
32 - 64	0.5889	**0.6416**	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. If no 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.

VI. Implementation and Evaluation

We have implemented Daisy in the Scala programming language. Internal arithmetic operations are implemented with a rational data type to avoid internal roundoff errors and ensure soundness, together with outwards rounding where necessary. Apart from the (optional) use of the Z3 SMT solver [21] for more accurate range computations, Daisy 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 [10], [11], and is largely orthogonal to the focus of this paper. Our error computation method can also be extended to transcendental functions [22] and we plan to implement this extension in the future.

a) Comparison with FPTuner: 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 Daisy with FPTuner [7], which is the only other tool for *sound* mixed-precision tuning. FPTuner only supports floating-point arithmetic, so that we perform the experimental evaluation here for floating-point arithmetic only. FPTuner reduces mixed-precision tuning to an optimization problem, where the optimization objective is performance. The user has to provide optimization constraints, e.g. in form of a maximum number of cast operations. FPTuner also supports ganging of operators limiting specific operations to have the same precision, which can be useful for vectorization. Adding these kinds of constraints to our approach is straightforward, however, this optimization requires user expertise and manual input. As we focus on automated techniques, we do not consider it here.

We specify FPTuner input programs such that all arithmetic operations are to be optimized separately. For an automated approach, we do not limit the number of cast operations.

We do not compare against Precimonious, since Daisy uses the same search algorithm internally. We would thus be merely comparing sound and unsound error analyses, which in our view is not meaningful: even if under an unsound error estimation a mixed-precision assignment with better performance is found, we cannot tell whether it is actually a valid candidate assignment which satisfies the error bound.

b) Benchmarks: We have experimentally evaluated our approach and tool on a number of standard finite-precision verification benchmarks [2], [4], [8]. The benchmarks rigidBody, invertedPendulum and traincar are embedded controllers; bsplines, sine, and sqrt are examples of functions frequently used in the embedded domain, and doppler, turbine, himmilbeau and kepler are from the scientific computing domain. 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. Table I lists the number of arithmetic operations and variables for each benchmark. An asterisk ($*$) marks nonlinear benchmarks.

We follow FPTuner and select error bounds which are to be satisfied as follows. The error bounds for benchmarks denoted by F and D, are those satisfiable by *uniform* single and double precision, 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. This corresponds to a scenario where uniform precision is just barely not enough and we would like to avoid the next higher uniform precision.

c) Experimental Setup: We have performed all experiments on a Linux desktop computer with Intel Xeon 3.30GHz and 32GB RAM. For benchmarking, we use programs generated by Daisy in Scala (version 2.11.6) and translate FPTuner's output into Scala (FPTuner's output is platform independent). We use the Scalameter tool [20] (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 Daisy and FPTuner and not compiler optimization effects, which are out of scope of this paper.

d) Optimization Time: Table I compares the execution times of Daisy and FPTuner (average real time measured by the bash `time` command). For Daisy, we report the times for mixed-precision tuning only without rewriting, as well as the time for full optimization, i.e. rewriting and mixed-precision tuning. 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 together.

Daisy is faster than FPTuner for all benchmarks, even with rewriting included, and often by large factors. Daisy's mixed-precision tuning also appears to be more scalable. On the longer benchmarks (marked '2x' and '3x'), Daisy's running time scales roughly linearly with the size of the benchmark, whereas FPTuner's times for the kepler2 and rigidBody2 benchmarks increase by a factor of 17 and 7, respectively. We suspect that this is due to the fact that FPTuner is solving a global optimization problem, which is known to be hard.

e) Performance Improvements: To evaluate the effectiveness of our mixed-precision tuning and rewriting, we performed end-to-end performance experiments. We benchmark each generated optimized program five times with Scalameter on 10^5 random inputs from the valid input range and record the average running time. For each mixed-precision variant, we compare its running time against the running time of the corresponding, i.e. next higher, uniform-precision program (e.g. uniform double for the $F_{0.5}$ benchmark, and uniform quad for $D_{0.01}$).

Figure 2 shows the relative improvements, i.e. mixed-precision/uniform running time, averaged over all benchmarks.

benchmark	#ops-#vars	FPTuner	Daisy-mixed	Daisy-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 I: Optimization times of Daisy and FPTuner

The full experimental results are available in the appendix. We compare the results for Daisy with mixed-precision tuning only, with rewriting only, and with both rewriting and tuning, as well as the results of FPTuner. We furthermore ran FPTuner on programs which were rewritten by Daisy, simulating a possible combination of rewriting with FPTuner's mixed-precision tuning.

Overall, we see that the biggest speedups occur for the $F_{0.5}$ and $D_{0.5}$ benchmarks, as expected.

Comparing FPTuner with Daisy's mixed-precision tuning, we observe that the results are comparable for the benchmarks where single and double precision is suitable. For the benchmarks $D_{0.5}$, FPTuner provides more aggressive improvements, but on the other hand, for others it also produces programs which are slower than the original program (see the appendix for non-aggregated data). Daisy is more conservative, both with performance improvements but also with slowdowns, increasing the execution time only rarely. This suggests that using a static (benchmarked) cost function is a feasible and successful approach. Considering Daisy's significantly better optimization time and scalability, we believe that Daisy provides an interesting tradeoff between mixed-precision tuning performance 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 alone is modest. When the two techniques are combined, we obtain the biggest performance improvements, which are furthermore more than just the sum of both parts.

Finally, rewriting also has a positive effect on FPTuner's results, which suggests that it should be included in mixed-precision optimization in general.

VII. Related Work

a) Rewriting: An alternative approach to rewriting was presented by [23] which relies on a greedy search and an

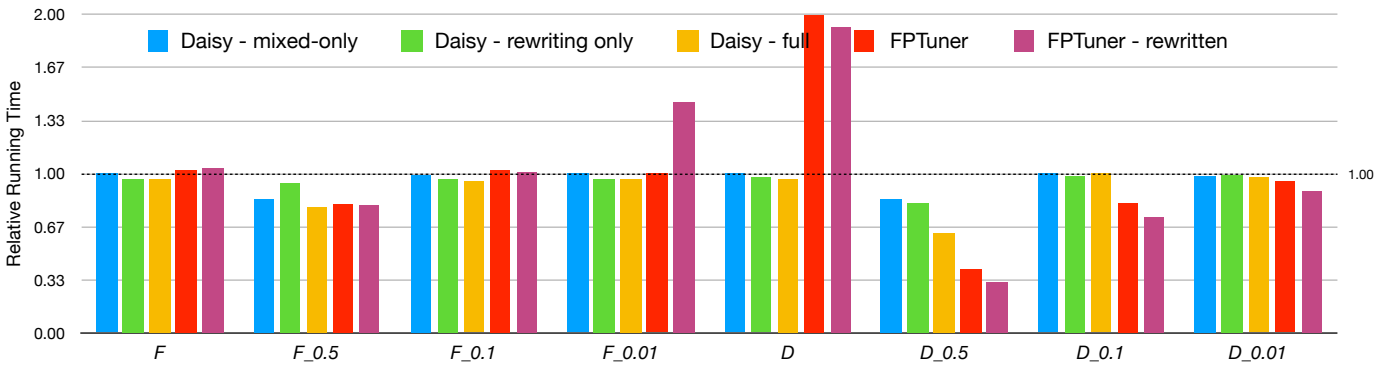

Fig. 2: Average performance improvements (lower is better) over all benchmarks. 1.0 means no change.

abstract domain which represents possible expression rewritings together with a static error analysis similar to ours. The tool Herbie [9] 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 roundoff error and then targets the rewriting rules at that part of the expression. We do not employ this localization, as we have not observed catastrophic cancellations in our benchmarks and because we also want to optimize the non-catastrophic case. Our sound error computation would correctly determine large roundoff errors if there was a cancellation. It would be interesting in the future to compare these different search techniques for rewriting.

b) Mixed-precision Tuning in HPC: Mixed-precision is especially important in HPC applications, because floating-point arithmetic is widely used. [24] introduced a binary intrumentation tool together with a breadth-first search algorithm to help programmers search for suitable mixed-precision programs. This work was later extended to perform a sensistivity analysis [5] based on a more fine-grained approach. The Precimonious project [6], [25], 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.

c) 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 [26], [27], but recently also non-semantics preserving ones have been proposed in the space of approximate computing [28]. These techniques represent another avenue for improving performance, but do not optimize mixed-precision.

d) 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, [29] considers both fixed-point and floating-point programs and uses automatic differentiation for a sensitivity analysis. [30] present an optimal bit-width allocation for two variables and a greedy heuristic for more variables, and rely on dynamic error evaluation. 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 [31]–[34] and [31] provide a nice overview of static and dynamic techniques. For optimization, [31] have used simulated annealing as the search technique. Like most of the work in this space, the target is area reduction for FPGAs or ASICs and not performance. Rewriting has also not been considered to the best of our knowledge. 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.

e) 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 [2], Fluctuat [1], FPTaylor [3] (which FPTuner is based on) and Real2Float [4]. The accuracies of these tools are mostly comparable [10], 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 [35]–[37]. 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 [38], [39] and HOL Light [40], and entire numerical programs have been proven correct and accurate within these [41], [42]. 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 [43] theory and SMT solvers exist which include floating-point decision procedures [21], [44]. 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.

VIII. 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] E. Goubault and S. Putot, "Static Analysis of Finite Precision Computations," in *VMCAI*, 2011.

[2] E. Darulova and V. Kuncak, "Sound Compilation of Reals," in *POPL*, 2014.

[3] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan, "Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions," in *FM*, 2015.

[4] V. Magron, G. A. Constantinides, and A. F. Donaldson, "Certified Roundoff Error Bounds Using Semidefinite Programming," *CoRR*, vol. abs/1507.03331, 2015.

[5] M. O. Lam and J. K. Hollingsworth, "Fine-grained floating-point precision analysis," *Intl. J. of High Performance Computing Applications*, 2016.

[6] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning Assistant for Floating-point Precision," in *SC*, 2013.

[7] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, I. Briggs, M. S. Baranowski, and A. Solovyev, "Rigorous Floating-point Mixed Precision Tuning," in *POPL*, 2017.

[8] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha, "Synthesis of Fixed-point Programs," in *EMSOFT*, 2013.

[9] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically Improving Accuracy for Floating Point Expressions," in *PLDI*, 2015.

[10] E. Darulova and V. Kuncak, "Towards a compiler for reals," *ACM TOPLAS*, vol. 39, no. 2, 2017.

[11] E. Goubault and S. Putot, "Robustness Analysis of Finite Precision Implementations," in *APLAS*, 2013.

[12] R. Majumdar, I. Saha, and M. Zamani, "Synthesis of Minimal-error Control Software," in *EMSOFT*, 2012.

[13] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson, "C++/Fortran-90 double-double and quad-double package," Tech. Rep., 2015. [Online]. Available: http://crd-legacy.lbl.gov/~dhbailey/mpdist/

[14] A. Anta, R. Majumdar, I. Saha, and P. Tabuada, "Automatic Verification of Control System Implementations," in *EMSOFT*, 2010.

[15] R. Moore, *Interval Analysis*. Prentice-Hall, 1966.

[16] L. H. de Figueiredo and J. Stolfi, "Affine Arithmetic: Concepts and Applications," *Numerical Algorithms*, vol. 37, no. 1-4, 2004.

[17] A. Izycheva and E. Darulova, "On sound relative error bounds for floating-point arithmetic," in *FMCAD*, 2017.

[18] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Lulu Enterprises, 2008.

[19] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input." *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[20] A. Prokopec, "Scalameter," https://scalameter.github.io/, 2012.

[21] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008.

[22] E. Darulova and V. Kuncak, "Trustworthy Numerical Computation in Scala," in *OOPSLA*, 2011.

[23] N. Damouche, M. Martel, and A. Chapoutot, "Intra-procedural Optimization of the Numerical Accuracy of Programs," in *FMICS*, 2015.

[24] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically Adapting Programs for Mixed-precision Floating-point Computation," in *ICS*, 2013.

[25] C. Rubio-Gonzáles, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough, "Floating-Point Precision Tuning Using Blame Analysis," in *ICSE*, 2016.

[26] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. R. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson, "Spiral - A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms." *IJHPCA*, vol. 18, no. 1, pp. 21–45, 2004.

[27] R. Vuduc, J. W. Demmel, and J. A. Bilmes, "Statistical models for empirical search-based performance tuning," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 65–94, Feb. 2004.

[28] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic Optimization of Floating-point Programs with Tunable Precision," in *PLDI*, 2014.

[29] A. A. Gaffar, O. Mencer, W. Luk, and P. Y. K. Cheung, "Unifying Bit-Width Optimisation for Fixed-Point and Floating-Point Designs." *FCCM*, 2004.

[30] A. Mallik, D. Sinha, P. Banerjee, and H. Zhou, "Low-Power Optimization by Smart Bit-Width Allocation in a SystemC-Based ASIC Design Environment." *IEEE Trans. on CAD of Integ. Cir. and Sys.*, 2007.

[31] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-Guaranteed Bit-Width Optimization," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 25, no. 10, 2006.

[32] W. G. Osborne, R. C. C. Cheung, J. Coutinho, W. Luk, and O. Mencer, "Automatic Accuracy-Guaranteed Bit-Width Optimization for Fixed and Floating-Point Systems," in *FPL*, 2007.

[33] A. B. Kinsman and N. Nicolici, "Finite Precision Bit-Width Allocation using SAT-Modulo Theory," in *DATE*, 2009.

[34] Y. Pang, K. Radecka, and Z. Zilic, "An Efficient Hybrid Engine to Perform Range Analysis and Allocate Integer Bit-widths for Arithmetic Circuits," in *ASPDAC*, 2011.

[35] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A Static Analyzer for Large Safety-Critical Software," in *PLDI*, 2003.

[36] L. Chen, A. Miné, and P. Cousot, "A Sound Floating-Point Polyhedra Abstract Domain," in *APLAS*, 2008.

[37] B. Jeannet and A. Miné, "Apron: A Library of Numerical Abstract Domains for Static Analysis," in *CAV*, 2009.

[38] S. Boldo and G. Melquiond, "Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq," in *ARITH*, 2011.

[39] M. Daumas, L. Rideau, and L. Théry, "A Generic Library for Floating-Point Numbers and Its Application to Exact Computing," in *TPHOLs*, 2001.

[40] C. Jacobsen, A. Solovyev, and G. Gopalakrishnan, "A Parameterized Floating-Point Formalizaton in HOL Light," *Electronic Notes in Theoretical Computer Science*, vol. 317, pp. 101–107, 2015.

[41] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis, "Wave Equation Numerical Resolution," *Journal of Automated Reasoning*, vol. 50, no. 4, pp. 423–456, 2013.

[42] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin, "A Unified Coq Framework for Verifying C Programs with Floating-Point Computations," in *CPP*, 2016.

[43] P. Rümmer and T. Wahl, "An SMT-LIB Theory of Binary Floating-Point Arithmetic," in *SMT*, 2010.

[44] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening, "Deciding floating-point logic with abstract conflict driven clause learning," *Formal Methods in System Design*, vol. 45, no. 2, pp. 213–245, Dec. 2013.

[45] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT Solver for Nonlinear Theories over the Reals," in *CADE*, 2013.

floating-point uniform prec. as proxy for	32	5% of 32	10% of 32	64	5% of 64	10% of 64	128	5% of 128	10% of 128
floating-point mixed precision	0.15	0.55	0.10	0.15	0.55	0.10	0.20	0.48	0.08
fixed mixed-precision	0.50	0.15	0.05	0.50	0.15	0.05	0.08	0.55	0.06

fixed uniform precision as proxy for	16	5% of 16	10% of 16	32	5% of 32	10% of 32
fixed mixed-precision	0.15	0.49	0.04	0.15	0.49	0.04

Fig. 3: Experimental results for rewriting stability experiment

APPENDIX

A. Range Arithmetic in Daisy

Daisy supports (as Rosa) interval [15] and affine arithmetic [16] as well as a more accurate, but also more expensive combination of interval arithmetic and SMT [2]. Here we provide some more background on these.

Interval arithmetic (IA) [15] is an efficient choice for range estimation, which computes a bounding interval for each basic operation as $x \circ y = [\min(x \circ y), \max(x \circ y)], \circ \in \{+, -, *, /\}$ and analogously for square root. Interval arithmetic cannot track correlations between variables (e.g. $x - x \neq 0$), and thus can introduce significant over-approximations of the true ranges, especially when the computations are longer.

Affine arithmetic (AA) [16] tracks *linear* correlations by representing possible values of variables as affine forms:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i, \quad \text{where } \epsilon_i \in [-1, 1]$$

where x_0 denotes the central value (of the represented interval) and each *noise term* $x_i \epsilon_i$ denotes a deviation from this central value. The range represented by an affine form is computed as $[\hat{x}] = [x_0 - rad(\hat{x}), x_0 + rad(\hat{x})]$, $rad(\hat{x}) = \sum_{i=1}^{n} |x_i|$. Linear operations are performed term wise and are computed exactly, whereas nonlinear ones need to be approximated and thus introduce over-approximations. Overall, AA produces sometimes (though not always) tighter ranges in practice. In particular when the individual noise terms (x_i's) are small, e.g. when they track roundoff errors, AA is often beneficial.

Combination of IA and SMT The overapproximation due to nonlinear arithmetic can be mitigated by refining ranges computed by IA with a binary search in combination with a complete (though expensive) nonlinear arithmetic decision procedure inside the Z3 [21] SMT solver [2]. In order to support transcendental functions, Daisy also features an interface to the dReal [45] solver, which is only δ-complete, but can handle also transcendental functions.

B. Rewriting Stability under Different Precisions

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.

In the following experiment, we test whether choosing one precision over another has a significant effect on the results. If the effects are small, then we can choose to perform rewriting wrt. to any precision, which in Daisy is double floating-point precision.

a) Experimental Setup: 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 then 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 expression returned with rewriting wrt. some 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 as thresholds to be that the errors should be within 5%, respectively 10%.

b) Results: Figure 3 summarizes our experimental results. The top table shows the percentage of times where each of the floating-point uniform precisions was a good proxy. Here '32, 64, 128' note uniform single, double and quad precision, respectively. '5% of 32' denotes that the errors were within 5% of the uniform single precision errors, and similarly for the other precisions. Overall, single and double floating-point precision were a good proxy (within 10%) in roughly 80% of the cases, whereas quad precision in 75%. When the mixed-precision assignments were in fixed-point precision (second line in the table), single, double and quad uniform precision all achieve 69% accuracy.

Performing the rewriting wrt. fixed-point arithmetic is not more beneficial either, as the bottom table shows. The overall accuracy is 68%. Here the considered uniform precisions were 16- and 32-bit fixed-point formats. Finally, rewriting in uniform precision never increased the errors (when evaluated in the mixed-precision baseline). We thus choose to perform the rewriting with respect to double floating-point precision. We could have equally chosen single floating-point precision; quad precision is more expensive, however, due to implementation details.

C. Detailed Experimental Results

Table II shows detailed relative improvements, i.e. mixed-precision/uniform running time. Very low values for Daisy result from rewriting reducing the roundoff error sufficiently for a smaller *uniform* precision being enough. For FPTuner, the very low and very high values are caused by different characteristics of its error analyses, which for some benchmarks computes smaller roundoff errors than Daisy (and thus assigns a smaller uniform precision), while for others it computes bigger roundoff errors.

benchmark	F	$F_{0.5}$	$F_{0.1}$	$F_{0.01}$	D	$D_{0.5}$	$D_{0.1}$	$D_{0.01}$	avrg
bspline2	1.01	**0.55**	1.00	1.00	1.00	1.00	1.00	1.00	**0.94**
doppler	0.97	**0.89**	0.96	**0.95**	0.96	1.01	0.99	1.00	0.97
himmilbeau	0.98	**0.95**	1.02	0.98	1.02	**0.61**	1.04	1.00	**0.95**
invPend.	0.98	1.02	1.01	0.98	0.99	**0.85**	1.00	1.01	0.98
kepler0	1.00	1.07	1.00	1.00	1.00	**0.85**	0.97	1.00	0.99
kepler1	1.00	**0.90**	**0.95**	0.99	0.98	**0.89**	1.10	1.00	0.98
kepler2	1.00	1.02	**0.93**	1.00	1.00	**0.85**	1.14	1.00	0.99
rigidBody1	1.04	1.02	1.02	1.00	0.97	**0.72**	**0.94**	1.00	0.96
rigidBody2	0.97	**0.94**	1.01	1.04	1.01	0.98	1.16	1.00	1.01
sine	1.01	**0.64**	1.00	0.99	1.00	1.24	1.00	1.00	0.99
sqroot	1.02	**0.84**	0.99	1.01	1.00	**0.59**	1.00	1.00	**0.93**
traincar	0.98	0.97	0.98	1.01	1.00	**0.61**	**0.61**	**0.87**	**0.88**
turbine1	1.00	**0.63**	1.00	1.01	1.01	1.18	1.00	1.00	0.98
turbine2	0.98	**0.66**	0.98	1.01	0.98	**0.77**	1.00	1.00	**0.92**
turbine3	1.00	**0.62**	0.99	0.99	0.99	1.18	1.00	1.01	0.97
kepler2(2x)	0.98	**0.91**	0.99	0.99	1.00	**0.65**	1.13	1.01	0.96
rigidBody2(3x)	0.99	1.02	1.00	0.97	0.97	**0.74**	1.09	1.00	0.97
sine (3x)	1.00	**0.75**	1.00	1.00	1.00	**0.69**	1.00	1.00	**0.93**
traincar(2x)	1.13	1.05	1.05	1.12	1.14	**0.61**	**0.87**	**0.93**	0.99
average	1.00	**0.84**	0.99	1.00	1.00	**0.85**	1.00	0.98	0.96

(a) Daisy - mixed-precision tuning only

benchmark	F	$F_{0.5}$	$F_{0.1}$	$F_{0.01}$	D	$D_{0.5}$	$D_{0.1}$	$D_{0.01}$	avrg
bspline2	0.99	0.99	0.99	0.99	0.98	**0.90**	**0.90**	**0.90**	0.96
doppler	1.02	**0.90**	1.33	1.32	1.33	**0.10**	1.01	1.02	1.00
himmilbeau	0.96	0.98	0.98	0.99	0.98	0.99	0.99	0.99	0.98
invPend.	1.03	0.99	1.00	1.03	1.03	**0.95**	**0.95**	**0.95**	0.99
kepler0	0.98	1.02	0.99	1.00	0.99	0.97	0.97	0.98	0.99
kepler1	0.99	**0.87**	**0.87**	**0.87**	**0.87**	1.00	1.00	1.00	**0.93**
kepler2	0.96	**0.92**	**0.92**	**0.95**	**0.93**	0.99	0.99	0.99	0.96
rigidBody1	0.99	0.99	1.01	1.01	0.99	**0.93**	**0.93**	**0.93**	0.97
rigidBody2	0.96	**0.88**	**0.89**	**0.90**	**0.90**	0.98	0.98	0.98	**0.93**
sine	0.99	1.08	1.08	1.08	1.09	0.98	0.98	0.98	1.03
sqroot	0.96	**0.95**	**0.93**	**0.92**	**0.93**	0.97	0.97	0.98	**0.95**
traincar	**0.89**	**0.85**	**0.89**	**0.89**	**0.91**	**0.07**	1.02	1.02	**0.82**
turbine1	1.02	0.97	0.97	0.96	0.96	1.06	1.06	1.06	1.01
turbine2	**0.94**	**0.94**	**0.94**	**0.94**	**0.95**	0.99	0.99	0.98	0.96
turbine3	1.01	0.96	0.97	**0.95**	**0.95**	1.07	1.07	1.07	1.01
kepler2(2x)	**0.89**	**0.78**	**0.86**	**0.87**	**0.88**	0.96	0.98	1.00	**0.90**
rigidBody2(3x)	1.04	0.96	**0.94**	**0.94**	0.98	1.00	0.99	1.00	0.98
sine (3x)	**0.93**	1.05	**0.91**	**0.90**	**0.90**	0.99	0.99	1.00	0.96
traincar(2x)	0.98	**0.90**	0.98	0.99	1.01	**0.09**	1.03	1.02	**0.87**
average	0.97	**0.95**	0.97	0.97	0.97	**0.82**	0.99	0.99	**0.95**

(b) Daisy - rewriting only

benchmark	F	$F_{0.5}$	$F_{0.1}$	$F_{0.01}$	D	$D_{0.5}$	$D_{0.1}$	$D_{0.01}$	avrg
bspline2	1.04	**0.50**	0.99	0.97	0.98	**0.77**	**0.90**	**0.91**	**0.88**
doppler	1.03	**0.87**	0.98	1.31	1.31	**0.10**	1.18	1.02	0.97
himmilbeau	0.99	0.96	1.01	1.00	0.98	**0.60**	1.07	0.99	**0.95**
invPend.	0.99	0.99	0.99	0.97	0.97	**0.68**	0.97	**0.95**	**0.94**
kepler0	**0.94**	1.05	0.99	0.96	0.96	**0.55**	1.00	0.98	**0.93**
kepler1	0.98	**0.91**	**0.87**	**0.90**	**0.87**	**0.89**	0.98	1.00	**0.93**
kepler2	0.96	1.04	**0.94**	**0.93**	**0.93**	**0.55**	1.08	0.98	**0.93**
rigidBody1	1.00	1.02	0.97	0.96	0.99	**0.57**	1.04	**0.93**	**0.94**
rigidBody2	0.97	0.98	**0.90**	**0.88**	**0.88**	**0.61**	1.16	0.98	**0.92**
sine	0.98	**0.65**	1.09	1.10	1.09	1.26	0.99	0.98	1.02
sqroot	0.97	**0.87**	**0.95**	**0.92**	**0.93**	**0.64**	1.02	0.97	**0.91**
traincar	**0.88**	**0.85**	**0.86**	**0.89**	**0.91**	**0.07**	**0.66**	**0.90**	**0.75**
turbine1	0.99	**0.68**	0.96	0.96	0.96	1.17	1.06	1.06	0.98
turbine2	0.98	**0.67**	**0.94**	**0.95**	**0.95**	**0.74**	0.99	0.99	**0.90**
turbine3	1.00	**0.70**	0.96	0.96	**0.95**	1.16	1.07	1.07	0.98
kepler2(2x)	**0.86**	**0.79**	**0.86**	0.96	**0.89**	**0.80**	1.05	1.00	**0.90**
rigidBody2(3x)	1.00	1.02	**0.94**	**0.95**	0.96	**0.55**	0.98	1.01	**0.93**
sine (3x)	**0.93**	**0.61**	**0.95**	**0.90**	**0.89**	**0.46**	1.03	0.99	**0.84**
traincar(2x)	0.96	**0.91**	**0.93**	0.97	1.00	**0.08**	**0.91**	0.99	**0.84**
average	0.97	**0.82**	**0.95**	0.97	0.97	**0.62**	1.00	0.98	**0.91**

(c) Daisy - full optimization (mixed-precision and rewr)

benchmark	F	$F_{0.5}$	$F_{0.1}$	$F_{0.01}$	D	$D_{0.5}$	$D_{0.1}$	$D_{0.01}$	avrg
bspline2	1.00	**0.53**	1.01	1.00	1.00	**0.65**	1.21	1.21	**0.95**
doppler	0.99	**0.87**	1.03	1.01	1.00	**0.07**	1.10	1.57	0.96
himmilbeau	0.98	**0.88**	1.22	0.99	1.01	**0.19**	**0.92**	1.06	**0.91**
invPend.	1.05	1.01	1.04	0.98	0.99	**0.61**	0.98	1.07	0.97
kepler0	1.06	1.06	1.07	1.01	1.01	**0.49**	1.01	1.10	0.98
kepler1	1.03	1.02	1.22	0.99	5.46	**0.57**	0.99	1.07	1.54
kepler2	0.98	1.05	1.11	1.24	3.03	**0.64**	**0.88**	1.04	1.25
rigidBody1	1.08	1.03	0.99	0.96	4.66	**0.61**	1.03	1.21	1.45
rigidBody2	1.08	**0.95**	1.18	1.02	5.52	**0.56**	**0.57**	**0.89**	1.47
sine	1.01	**0.43**	**0.86**	0.93	1.00	**0.20**	**0.48**	**0.68**	**0.70**
sqroot	1.14	**0.83**	1.22	1.12	4.86	**0.40**	**0.68**	**0.94**	1.40
traincar	1.01	**0.90**	**0.94**	**0.92**	4.55	**0.36**	**0.37**	**0.37**	1.18
turbine1	1.00	**0.56**	**0.88**	**0.95**	1.00	**0.07**	**0.86**	**0.92**	**0.78**
turbine2	1.00	**0.61**	**0.79**	0.98	1.00	**0.09**	**0.92**	1.13	**0.81**
turbine3	1.01	**0.58**	0.96	**0.92**	1.00	**0.07**	**0.68**	**0.91**	**0.77**
kepler2(2x)	0.99	0.97	1.13	1.09	2.91	**0.55**	**0.86**	crash	1.21
rigidBody2(3x)	1.03	1.03	1.13	1.08	4.97	**0.45**	**0.61**	**0.83**	1.39
sine (3x)	0.99	**0.53**	**0.85**	1.02	2.20	**0.19**	**0.45**	**0.69**	**0.86**
traincar(2x)	1.11	0.96	0.99	0.99	5.26	**0.50**	**0.51**	**0.51**	1.35
average	1.03	**0.81**	1.03	1.01	2.61	**0.40**	**0.82**	**0.95**	1.08

(d) FPTuner

benchmark	F	$F_{0.5}$	$F_{0.1}$	$F_{0.01}$	D	$D_{0.5}$	$D_{0.1}$	$D_{0.01}$	avrg
bspline2	1.00	**0.47**	1.11	0.98	0.98	**0.08**	**0.92**	1.07	**0.82**
doppler	1.08	**0.91**	1.02	1.36	1.37	**0.10**	0.98	1.08	0.99
himmilbeau	1.02	**0.87**	1.27	1.10	0.99	**0.04**	**0.81**	1.11	**0.90**
invPendulum	1.00	0.99	1.02	1.00	1.01	**0.33**	**0.84**	0.97	**0.90**
kepler0	1.00	1.05	1.11	1.00	0.99	**0.27**	**0.95**	1.05	**0.93**
kepler1	1.39	1.07	1.07	**0.85**	**0.85**	**0.59**	**0.89**	1.05	0.97
kepler2	0.98	1.01	1.23	1.11	**0.94**	**0.58**	**0.88**	0.98	0.96
rigidBody1	0.98	1.03	1.13	0.99	0.99	**0.48**	1.00	1.11	0.96
rigidBody2	1.06	**0.95**	1.22	10.21	4.26	**0.45**	**0.48**	**0.93**	2.45
sine	1.00	**0.40**	**0.91**	1.02	1.09	**0.07**	**0.38**	**0.68**	**0.69**
sqroot	1.04	1.15	1.12	1.10	9.14	0.99	**0.59**	**0.81**	1.99
traincar	**0.89**	**0.89**	**0.85**	**0.85**	2.55	**0.21**	**0.21**	**0.21**	**0.83**
turbine1	1.03	**0.58**	**0.79**	**0.82**	0.98	**0.07**	**0.76**	**0.93**	**0.75**
turbine2	1.01	**0.62**	**0.75**	**0.94**	**0.93**	**0.08**	**0.79**	1.12	**0.78**
turbine3	1.02	**0.59**	**0.94**	**0.88**	6.13	**0.06**	**0.60**	**0.84**	1.38
kepler2(2x)	**0.88**	**0.78**	1.03	-	**0.85**	**0.48**	**0.88**	-	**0.82**
rigidBody2(3x)	**0.91**	1.05	-	1.11	0.96	**0.27**	**0.47**	**0.77**	**0.79**
sine (3x)	1.43	**0.48**	**0.72**	**0.88**	**0.90**	**0.16**	**0.38**	**0.68**	**0.70**
traincar(2x)	1.01	**0.91**	**0.87**	**0.88**	2.40	-	**0.23**	**0.24**	**0.93**
average	1.04	**0.80**	1.01	1.45	1.92	**0.32**	**0.73**	**0.89**	1.02

(e) FPTuner - with rewriting

TABLE II: Relative performance improvements for Daisy and FPTuner, **bold**: > 5% improvement, underline: > 5% slowdown. '-' denotes a crash/no result.