Regime Inference for Sound Floating-Point Optimizations

ROBERT RABE and ANASTASIIA IZYCHEVA, Fakultät für Informatik, TU München, Germany
EVA DARULOVA, MPI-SWS, Germany

Efficient numerical programs are required for proper functioning of many systems. Today's tools offer a variety of optimizations to generate efficient floating-point implementations that are specific to a program's input domain. However, sound optimizations are of an "all or nothing" fashion with respect to this input domain—if an optimizer cannot improve a program on the specified input domain, it will conclude that no optimization is possible. In general, though, different parts of the input domain exhibit different rounding errors and thus have different optimization potential. We present the first regime inference technique for *sound* optimizations that automatically infers an effective subdivision of a program's input domain such that individual sub-domains can be optimized more aggressively. Our algorithm is general; we have instantiated it with mixed-precision tuning and rewriting optimizations to improve performance and accuracy, respectively. Our evaluation on a standard benchmark set shows that with our inferred regimes, we can, on average, improve performance by 65% and accuracy by 54% with respect to whole-domain optimizations.

CCS Concepts: • **Computing methodologies → Optimization algorithms**; • **Software and its engineering → Automated static analysis;**

Additional Key Words and Phrases: Floating-point arithmetic, regime inference

ACM Reference format:
Robert Rabe, Anastasiia Izycheva, and Eva Darulova. 2021. Regime Inference for Sound Floating-Point Optimizations. *ACM Trans. Embedd. Comput. Syst.* 20, 5s, Article 81 (September 2021), 23 pages.
https://doi.org/10.1145/3477012

1 INTRODUCTION

Numerical algorithms in many domains require accurate yet efficient implementations, and this is particularly true for often resource-constraint embedded systems. When using reals [8] is prohibitively expensive, IEEE-754 floating-point arithmetic [26] provides a convenient and practical trade-off: finite precision enables an efficient execution, at the expense of rounding errors and thus reduced accuracy. Floating-point arithmetic is specifically suited for systems that come with a hardware floating-point unit, as many microcontrollers do today [3], and that need to perform complex math [4, 24], including elementary functions such as sine and logarithm that

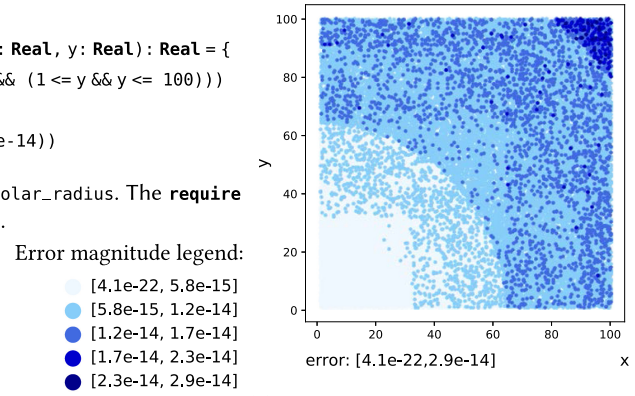
This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2021.
Authors' addresses: R. Rabe and A. Izycheva, Department of Informatics, Technical University of Munich, Boltzmannstr.3, 85748 Garching near Munich, Germany; emails: robert.rabe@tum.de, izycheva@in.tum.de; E. Darulova, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Strasse, G 26, D-67663 Kaiserslautern, Germany; email: eva@mpi-sws.org.

1539-9087/2021/09-ART81 $15.00
https://doi.org/10.1145/3477012

Corrected Version of Record. V.1.1. Published October 8, 2021.

```
1  def carthesianToPolar_radius(x: Real, y: Real): Real = {
     require(((1 <= x && x <= 100) && (1 <= y && y <= 100)))
3    sqrt(((x * x) + (y * y)));
   } ensuring(res => (res +/- 2.51e-14))
```

(a) Source code of `carthesianToPolar_radius`. The **`require`** clause specifies the input domain.



(b) Error profile for `carthesianToPolar_radius`

Fig. 1. Example program `carthesianToPolar_radius` that computes polar 'radius' from Cartesian coordinates.

are conveniently and efficiently implemented in library functions. Depending on the needs of an application, we can increase accuracy as needed by choosing a higher precision [1, 2, 6], at the expense of longer running time.

However, choosing a suitable precision even for relatively short arithmetic expressions is challenging. Each arithmetic operation potentially introduces a rounding error that propagates in unintuitive ways through the remaining computation [19]. Additionally, the magnitude of rounding errors further depends on the magnitude of an expression's variables, as well as the exact order of computations [12, 37]. This makes *manual* navigation of the accuracy-efficiency trade-off essentially infeasible, especially for non-numerical experts.

The need for automated tool support motivated the development of a number of recent tools that analyze [17, 18, 20, 35, 42] and optimize numerical programs w.r.t performance or accuracy. For instance, *mixed-precision tuning* [9, 10, 12, 27, 38] automatically assigns different precisions to different arithmetic operations (hence mixed precision), while meeting some overall, user-specified error bound. Operations that significantly contribute to the final error are assigned a higher precision, whereas other operations can be performed in a lower precision in order to *improve performance*. Orthogonally, we can *increase accuracy* of floating-point expressions by *rewriting* them [12, 15, 37, 44, 46] using real-valued identities or approximations. Associativity and distributivity do not hold in floating-point arithmetic, and thus different orders of computation will exhibit different errors.

Since rounding errors depend on the magnitude of an expression's variables, such optimizations are necessarily specialized for a particular user-defined input domain. However, most current tools consider the specified input domain as a whole, that is, they generate one optimized expression for the entire domain. This often leads to suboptimal results, because rounding errors typically vary across the input domain, and so the optimized expression may not be the ideal choice for a large part of the input domain. Consider, for instance, the program `carthesianToPolar_radius` and its error profile (for uniform double precision) in Figure 1 that shows the absolute errors of the function's results for different inputs; darker color depicts larger rounding errors. The plot indicates that inputs from the top right corner induce higher rounding errors, and thus sub-domains closer to the top-right corner will require a higher (mixed) precision assignment than bottom-left parts of the domain, and may require a different rewriting than other parts of the domain.

A few recent tools [37, 44, 46] propose to generate *regimes*—a partition of the input domain into sub-domains, each with a different optimized version of the program. These tools apply rewrites

in order to repair high rounding errors in certain parts of the domain. While they can successfully improve programs that suffer from large numerical issues, they estimate errors using a dynamic analysis (sampling) and thus do not provide accuracy *guarantees*. Furthermore, they are not immediately applicable for optimizing numerically stable code, i.e. *without* particularly large rounding errors.

In this paper, we present the first regime inference for *sound* floating-point optimizations, i.e. for optimizations whose accuracy analysis computes guaranteed worst-case error bounds for all possible (specified) inputs. Our approach partitions the input domain and optimizes each part separately with an existing sound mixed-precision tuning or rewriting optimization routine, improving performance or accuracy, respectively. By doing so, we provide a significant benefit also for numerically stable code.

Inferring *effective* regimes for sound optimizations is nontrivial. In particular, we cannot simply take partitions derived by a dynamic analysis from one of the existing tools [37, 44, 46]. While the error profile in Figure 1(b), which was also obtained with a dynamic analysis, appears to suggest certain partitions, these do not necessarily lead to partitions for which sound optimizations will provide any improvement. This is because we need to find regimes for which a sound optimization routine can *prove* that a particular error bound holds. Since sound error analyses necessarily abstract rounding errors, they commit over-approximations that are hard to predict a priori, and that make it difficult to guess a regime by sampling. Furthermore, the space of possible optimizations is highly discontinuous. For instance, mixed-precision tuning considers only a small number of distinct precisions for each variable, and changing the precision of only a single variable can have a disproportionate impact on the overall error.

In principle, there are infinitely many possible regimes and enumerating all of them is infeasible, especially because today's sound optimizations are relatively expensive. Furthermore, each domain split results in at least one conditional statement in the final generated code, which introduces an additional cost at runtime and decreases code readability. Moreover, for multivariate programs splitting along one variable's domain can be more beneficial than splitting along the other.

Due to the inherent complexity, we do not attempt to infer optimal regimes. Instead, we focus on finding regimes with *interval sub-domains* and combine two heuristic approaches inspired by techniques that have been successful in the area of floating-point analysis. Interval sub-domains allow to keep the regimes' cost low, as lower and upper bounds of each variable can be efficiently checked for at runtime.

Our algorithm first starts generating regimes *bottom-up*. Inspired by interval subdivision [13, 20], it splits the input domains of all variables to create a fixed number of sub-domains. It optimizes each separately and then attempts to merge sub-regimes with the same optimized expressions. Next, the algorithm proceeds in a *top-down* fashion inspired by branch-and-bound techniques [41]. Starting with the sub-domains generated by the bottom-up phase, the algorithm iteratively splits some of them on-demand, in each iteration selecting the variable to split based on the optimization objective. This combined technique allows to explore the space of regimes in both breadth as well as depth and avoids getting stuck at a local optimum.

Our regime inference algorithm is generic in the optimization and the optimizer. In this paper, we instantiate it with the currently available sound optimizations for floating-point arithmetic: rewriting and two mixed-precision tuning routines from Daisy [12] and FPTuner [9], respectively. The overall soundness of our approach follows from the soundness of the individual optimizers, i.e. our approach guarantees that an overall error bound (specified by the user) is met. While we do not have a formal proof that can be checked independently by an interactive theorem prover, existing techniques for formal verification of rounding errors already support interval subdivision [7] so we expect them to generalize to our generated regimes as well.

```
def azimuth(lat1: Real,lat2: Real,lon1: Real,lon2: Real):Real = {
  require(((0.0 <= lat1) && (lat1 <= 0.4) &&
   (0.5 <= lat2) && (lat2 <= 1.0) &&
   (0.0 <= lon1) && (lon1 <= 3.142) &&
   (-3.142 <= lon2) && (lon2 <= -0.5)))

  val dLon: Real = (lon2 - lon1)
  val slat1: Real = sin(lat1)
  val clat1: Real = cos(lat1)
  val slat2: Real = sin(lat2)
  val clat2: Real = cos(lat2)
  val sdLon: Real = sin(dLon)
  val cdLon: Real = cos(dLon)
  atan((clat2 * sdLon) / ((clat1 * slat2) - ((slat1 * clat2) * cdLon)))
 } ensuring((res) => (res +/- 4.57e-14)) // 0.5x double error
// ensuring((res) => (res +/- 9.15e-15)) // 0.1x double error
```

Fig. 2. Source code of the `azimuth` benchmark.

Daisy and FPTuner can optimize straight-line numerical expressions consisting of arithmetic and elementary operations. Unfortunately, no sound tool is available that can directly optimize loops since even bounding rounding errors in loops is a largely unsolved orthogonal problem [14]. That said, errors or noise in embedded control loops are often handled with control-theoretic techniques [5], so that the loop body can be optimized in isolation, and thus by our approach. For loops with a limited number of iterations, our approach can be applied to the unrolled loop, at the expense of (significantly) increased program size.

In this paper, we only consider IEEE-754 floating-point arithmetic, which is supported by both Daisy and FPTuner. However, our approach is also applicable to other arithmetics. Provided a suitable hardware-specific cost function, our algorithm can infer regimes for fixed-points programs (e.g., using Daisy, FPTuner does not support fixed-point arithmetic). For more complicated arithmetics, like mixed integer and finite-precision programs, currently there are no tools to perform sound optimizations directly (i.e. without approximating integers by floating-points), but once such tools appear, they can be immediately used within our framework (together with an appropriate cost function).

We evaluate our approach on 100 benchmarks from the standard benchmark set FPBench [11] and show that our algorithm infers regimes that on average improve performance by 65% and accuracy by 54%, compared to Daisy's whole-domain optimizations, and by 52% compared to FPTuner's mixed-precision optimizations.

Contributions. In summary, this paper makes the following contributions:

- we present the first *sound* regime inference algorithm,
- that we implement in a prototype tool called Regina, available open-source at https://github.com/malyzajko/daisy, and
- extensively evaluate it using mixed-precision tuning and rewriting optimizations and show that regime inference is highly beneficial for sound floating-point optimizations.

2 EXAMPLE

Before explaining our regime inference algorithm in detail, we provide a high-level overview. Consider the function `azimuth` in Figure 2 that computes the angle between an observer and a point of

interest with a reference plane. Similar computations frequently appear in domains such as cyber-physical systems or robotics, where they are executed often, so should run fast, but where they may also need high accuracy. The example is specified as a real-valued function, together with a precondition (**`require`** clause on line 2) that bounds the possible input values, as well as a postcondition (the **`ensuring`** clause in line 15) that specifies a maximum absolute error on the result. In practice, the input domain would be, for instance, determined from valid ranges of sensors, and the maximum allowed error from the sensitivity of actuators, or stability proofs in the case of controllers [5].

Regina's goal is to find an effective partition of the program's input domain such that the program can be soundly optimized on each sub-domain separately, leading to an overall reduction in a given cost metric. In this paper, we consider two optimizations, mixed-precision tuning and rewriting, that consider running time and accuracy of the generated code as the cost metric, respectively. We call an optimized expression together with the sub-domain it has been optimized on a *sub-regime*. A set of non-overlapping sub-regimes covering the whole input domain is called a *regime*, and the *size of regime* denotes the number of sub-regimes.

Mixed-Precision Tuning. Mixed-precision tuning [9, 12, 38] is a process of assigning (potentially different) finite-precision types to each variable and arithmetic operation. Its goal is to increase performance while satisfying a user-defined error bound. Since some operations contribute to the final rounding error more than others, mixed-precision tuning attempts to assign higher precisions to these high-error inducing operations and uses less costly lower precision on others. Suppose that a user has specified that the worst-case absolute error of the function `azimuth` should be 4.57e-14. The sound roundoff error analysis tool Daisy [13] determines a worst-case error of 9.15e-14 when all operations are in uniform double floating-point precision, which is not enough to meet the target error bound. Since the error is close to the target error (less than an order of magnitude), it might be enough to increase the precision for only a subset of variables. Unfortunately, when Daisy's mixed-precision tuning algorithm [12] optimizes the program on the whole specified input domain, it assigns quad precision[1] to all but the input variables. The resulting program is as slow as the uniform quad precision implementation.

Our regime inference tool Regina, parametric in the optimization and optimizer, can be used on top of Daisy's mixed-precision tuning to find a faster program than Daisy alone. First, Regina subdivides the input domain into 24 sub-domains, and runs Daisy's mixed-precision tuning on each of them separately. This optimization results in the same precision assignment on several sub-domains, which are subsequently (partially) merged. For the target error of 4.57e-14, the 24 original sub-domains are merged into 5. In a second step, our top-down phase starts from these 5 sub-domains and attempts to find additional splits that reduce the (abstract) cost. In our example, Regina finds one more beneficial split and returns 6 sub-domains. In fact, Regina finds that only a single sub-domain (out of the resulting 6) actually requires mixed-precision:

$$\text{lat1} \in [\mathbf{0.2}, \mathbf{0.4}], \text{lat2} \in [\mathbf{0.5}, \mathbf{0.625}], \text{lon1} \in [\mathbf{2.094}, \mathbf{3.142}], \text{lon2} \in [-\mathbf{3.142}, -\mathbf{1.821}]$$

Regina encodes the sub-domains using if-then-else statements. We show the structure of the resulting generated program in Figure 3(a), and the code generated for the one mixed-precision sub-domain in Figure 3(b). The remaining 5 sub-domains use uniform **`double`** precision. The generated C code meets the user-specified error bound and *runs 93% faster* than the mixed-precision implementation that Daisy alone generates.

[1] IEEE quad precision has 128 bits, but libraries, such as GCC's quadmath [1] that we use often provide slightly less precision for increased performance.

```
if ((lat2 <= 0.75)) {
  if ((lat2 <= 0.625)) {
    if ((lon1 <= 2.0943951)) {
      // uniform double
    } else {
      if ((lon2 <= -1.820796325)) {
        if ((lat1 <= 0.2)) {
          // uniform double
        } else {
          // **mixed-precision**
        }
      } else {
        // uniform double
      }
    }
  } else {
    // uniform double
  }
} else {
  // uniform double
}
```

(a) Structure of generated code

```
__float128 dLon = ((__float128)lon2 - (__float128)lon1);
__float128 slat1 = sinq((__float128)lat1);
__float128 clat1 = cosq((__float128)lat1);
__float128 slat2 = sinq((__float128)lat2);
__float128 clat2 = cosq((__float128)lat2);
double sdLon = (double)sinq(dLon);
double cdLon = (double)cosq(dLon);
double _tmp3 = (double)(clat2 * (__float128)sdLon);
double _tmp1 = (double)(clat1 * slat2);
double _tmp = (double)(slat1 * clat2);
double _tmp2 = (_tmp * cdLon);
double _tmp4 = (_tmp1 - _tmp2);
double _tmp5 = (_tmp3 / _tmp4);
return atan(_tmp5);
```

(b) Mixed-precision sub-regime

Fig. 3. Sub-regimes in C generated for `azimuth` benchmark.

For a tighter error bound of 9.15e-15 as on line 16 in Figure 2 (an order of magnitude smaller than the uniform double error), Regina generated 11 sub-regimes of which 5 need mixed precision and the remaining ones can still be implemented in uniform double precision. The bottom-up phase again generates 24 sub-domains initially, and merges these into 10. The top-down phase subsequently splits one these so that 11 sub-regimes are generated overall. The generated code runs 84% faster than the uniform quad implementation that Daisy alone generates.

Rewriting. We further instantiate Regina's regime inference with the rewriting optimization (also provided by Daisy) that attempts to improve the worst-case absolute error bound of an expression using real-valued equivalence rules (i.e. the cost metric is accuracy). For our example, such rewriting can be applied on the computation on line 14 in Figure 2. Assuming uniform double precision, Daisy's rewriting applied to the whole domain is only able to improve the maximum error by 1%.

Regina generates 23 sub-regimes with 11 unique rewritten expressions, including, for instance:

```
sdLon * ( clat2 / ((slat2 * clat1) - (clat2 * (cdLon * slat1)) ) )
(clat2 * sdLon) / ((clat1 * slat2) - (slat1 * (clat2 * cdLon)))
clat2 * (sdLon / ((clat1 * slat2) - ((slat1 * cdLon) * clat2)))
```

The inferred regime has only 11 distinct rewritings, because sub-regimes with the same rewritten expression are not necessarily neighboring each other and thus cannot be merged. Note that we did not limit the number of branches for this optimization, since we optimized for accuracy, though it is straight-forward to customize Regina to limit the number of branches, if needed. We observed the cost of our simple conditional branches to be very small, especially compared to the additional

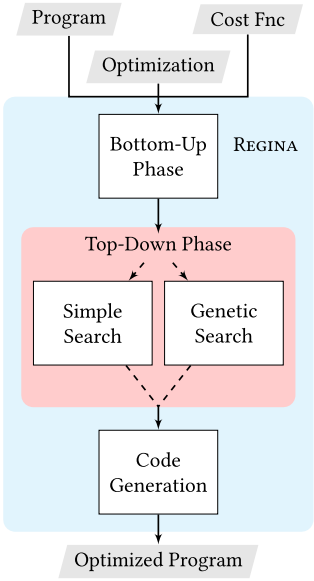


Fig. 4. Regime inference algorithm.

cost that rewriting may introduce due to additional operations (e.g. when applying distributivity). The generated code has a (proven) worst-case error that is 36% smaller than the expression which Daisy generates for the whole domain.

3 REGIME INFERENCE ALGORITHM

We focus on sound floating-point optimizations that guarantee or optimize a worst-case rounding error bound that holds for all specified inputs. Since the magnitude of rounding errors heavily depends on the domain of an expression's variables, different domains allow for different optimizations. While it is possible to partition an input domain using complex expressions, in this work we focus on sub-domains described by intervals for two main reasons. First, at runtime a program needs to evaluate partition conditions to decide which optimized version (sub-regime) to execute. Evaluating complex expressions takes longer than a simple interval bounds check, and may offset the performance improvements of each sub-regime (when optimizing for performance with mixed-precision tuning). Secondly, the rounding error analysis in state-of-the-art sound numerical programs' optimizers [9, 13] is fundamentally interval-based and thus does not leverage non-interval sub-domains well.

Interval sub-domains can be checked for efficiently with conditional statements (see Section 3.3). While such checks incur a negligible cost compared to the rest of the execution, we nonetheless want to limit the number of sub-regimes—to improve readability of the generated code and to reduce the running time of our regime inference procedure itself.

Regina's algorithm works in two phases: first running the bottom-up phase to explore the sub-domains up to an initial depth, and then the top-down phase that explores the space further, on demand, with one of two available search procedures. Figure 4 illustrates the high-level algorithm. The bottom-up phase, explained in detail in Section 3.1, exhaustively subdivides the input space and then attempts to merge sub-regimes with equal optimizations. The top-down phase, explained in Section 3.2, includes two alternative search procedures, each of which divides the given domain

on demand, guided by a (static) cost function. Both phases are motivated by two techniques that have been successful in reducing over-approximations in rounding error analysis in the tools Daisy and FPTuner, respectively.

Our approach is generic w.r.t. the floating-point optimization and the cost metric that is being optimized. We will use the function **`optimize`** to stand for some optimization routine that takes as input a domain, an expression and possibly a target rounding error bound and that returns a new, optimized expression. The **`cost`** function takes an expression and its domain as input and returns a numeric value reflecting the optimization objective; we will assume that lower cost is better. In Section 4, we show how we instantiate these algorithms with two different optimizations, mixed-precision tuning and rewriting, that optimize performance and accuracy, respectively.

We illustrate our algorithm using a running example, in which we infer a regime for mixed-precision tuning on the `carthesianToPolar_radius` function from Figure 1(a). In our example, the **`optimize`** function uses Daisy's mixed-precision tuning, and **`cost`** statically estimates the abstract performance of each tuned regime of `carthesianToPolar_radius`. **`cost`** does not estimate the actual running time, but rather an abstract cost that only needs to distinguish which of two regimes is likely to be faster. The detailed instantiation of **`optimize`** and **`cost`** for mixed-precision tuning is described in Section 4.1. In contrast to our approach, Daisy's mixed-precision tuning alone applied on the `carthesianToPolar_radius`'s whole input domain ($x \in [1.0, 100.0]$, $y \in [1.0, 100.0]$) did not result in any measurable performance improvements.

3.1 Bottom-Up Phase

The regime inference algorithm starts by exploring possible regimes in a bottom-up phase. It is inspired by interval subdivision, a technique that has been used in static rounding error analysis tools to reduce over-approximations [13, 20].

Figure 5 shows the pseudo-code of the bottom-up phase. First, it subdivides the input domain uniformly into smaller pieces and optimizes each one individually. We split the each variable's interval into equal pieces, as it is not obvious up front, i.e. before running the actual optimization, which sub-division will be beneficial. The number of initial sub-regimes clearly influences the possible improvements of regime inference, however, calling optimization on too many sub-regimes is expensive. In our implementation we currently limit the maximum number of sub-regimes that a method can generate to 32. Hence, depending on the number of input variables, we subdivide each variable's domain between 16 and 2 times.[2] We found empirically that larger initial sub-division size only increases the algorithm's running time, and does not change resulting regimes significantly. Understandably, on sufficiently small sub-domains an optimizer can no longer improve an individual sub-regime cost (e.g. in mixed-precision it already uses the lowest available precision).

The obtained optimized expressions on individual sub-domains constitute the initial regime. We have observed that frequently many optimized program bodies in this initial regime are equal. In a second step, our algorithm tries to find a smaller regime by merging the neighboring sub-domains whose body is the same (function **`mergeSameBodies`** in Figure 5). We call two sub-domains (and the corresponding sub-regimes) *neighboring* if they differ in ranges for exactly one variable, and after merging these ranges the new sub-domain does not overlap with any existing sub-domain. Having a smaller regime (i.e. fewer sub-regimes) is beneficial for readability of the generated code, as well as for the running time of our tool, as the running time of the successive top-down phase heavily depends on size of the given regime.

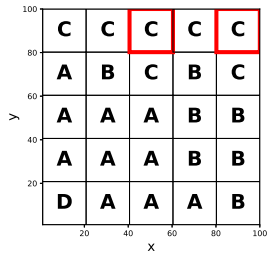
The initial regime for our running example is shown in Figure 6(a). Here, the input domain is split into 25 sub-domains (splitting the interval for both variables `x` and `y` into 5 equal-sized

[2]For programs with more than 5 variables we subdivide the 5 largest input intervals in half and leave the rest unchanged.

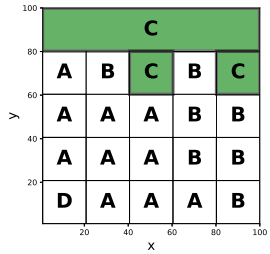
```
def bottom_up_phase(inputRanges, program, target):
  // produce regimes
  subdomains = splitTillMax(inputRanges)
  // run optimization
  regime = []
  for sub in subdomains:
    optProgram = optimize(sub, program, target)
    regime = regime ∪ (sub, optProgram)
  // merge sub-regimes
  regime = mergeSameBodies(regime)
  return regime

// merge sub-regimes with the same bodies
def mergeSameBodies(regime):
  subdomains = regime.subdomains
  for sub1, sub2 in neighbors(subdomains)
    if programIn(regime,sub1) == programIn(regime,sub2):
      upd = sub1 ∪ sub2
      expr = programIn(regime,sub1)
      toRemove = {(sub1,expr),(sub2,expr)}
      regime = regime ∪ (upd,expr) \ toRemove
      subdomains = subdomains ∪ {upd} \ {sub1,sub2}
```

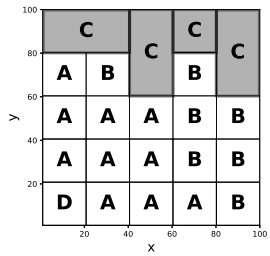
Fig. 5. Bottom-up phase.



(a) Input regime



(b) Optimal



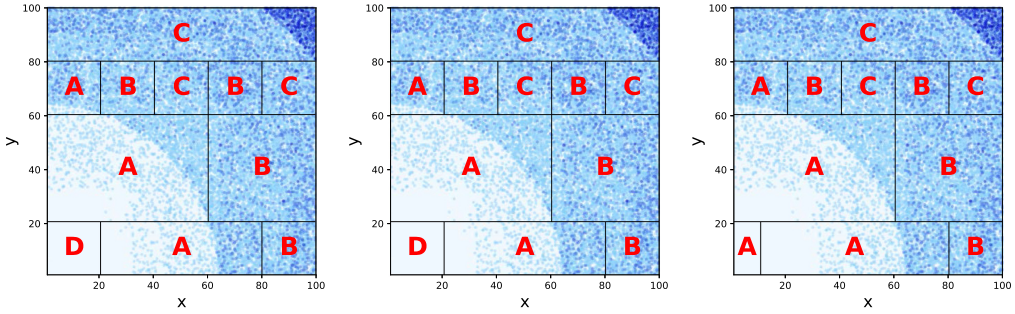
(c) Sub-optimal

Fig. 6. Merge strategy.

intervals), and the labels A-D denote different optimized expressions: A - all operations and intermediate results are assigned a `double` precision, B - one intermediate result is assigned `quad` precision, the rest use `double`, in C two variables have `quad` precision, and in D - four.

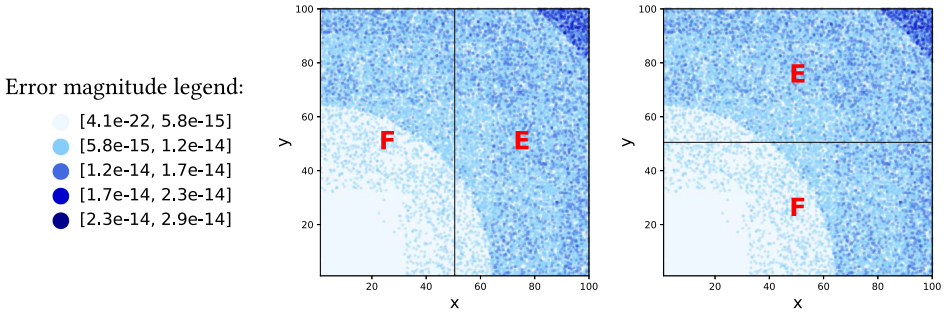
Finding the optimal merge with minimum number of resulting sub-regimes is an exact set cover problem, which is known to be NP-complete. Hence, our algorithm uses a heuristic to decide which neighbors should be merged: it starts with the first input variable (as they appear in the source code) and performs all possible merges along this variable, then repeats for the rest of input variables. The algorithm merges along each variable once. Depending on the order of variables in the source code such a heuristic can overlook beneficial joins. Consider an initial regime in Figure 6(a) that contains 10 sub-regimes with the optimized expression C. Red squares mark sub-regimes that have neighbors with the same optimized expressions along two variables: `x` and `y`. Merging along the variable `x` (horizontal axis) is clearly beneficial, as it will result in fewer sub-regimes in total (see Figure 6(b)). However, if the variable `y` appears in the list of function arguments before `x`, the algorithm will first merge along `y` (vertically) and create a sub-optimal result shown in Figure 6(c).

For our running example the bottom-up phase produces a regime that includes 11 sub-regimes, as shown in Figure 7(a). Compared to the version of `carthesianToPolar_radius` optimized on the



(a) Bottom-Up Phase (b) Bottom-Up + Top-Down Simple (c) Bottom-Up + Genetic

Fig. 7. Regimes inferred at different stages of the algorithm.



(a) Top-Down Simple Search (b) Top-Down Genetic Search

Fig. 8. Regimes inferred by top-down phase starting with the whole input domain.

whole specified domain, a program with this regime runs around 45% faster. Starting from this regime, our algorithm will try to further improve performance in the top-down phase.

3.2 Top-Down Phase

The next phase is inspired by the branch-and-bound technique that is being used by FPTuner [41] to find a domain-specific optimum of an arithmetic expression. Since the errors are not necessarily distributed uniformly over the input domain, it is often beneficial to sub-divide on-demand along selected variables. Our top-down phase starts from the regime found by the bottom-up phase (in Figure 7(a)) and tries to find a regime with even lower cost by repeatedly splitting variables' domains. We consider two approaches: a 'simple' search as well as genetic search. Both search approaches are guided by a static cost function, they stop if either no further cost improvement is possible in a single split, or when the algorithm has reached some maximum number of iterations.

For multivariate programs it is non-trivial to choose how to split a domain. One can split along one variable—split the range of one variable, while keeping ranges of the other variables intact—split along a subset of variables or all of them. Furthermore, we need to select a split point on each of the ranges. As for the merging strategies of the bottom-up phase, there is no way to know in advance which direction of split will be beneficial. Similarly, finding the most beneficial split point would require additional analysis of the domain and is expensive.

```
def top_down_phase(bUpRegime, program, target):
 def simpleSearch(regime, oldCost, depth):
  candidates = []
  depth += 1
  for var in program.inputVars:
    newRegime = []
    newSplit = splitInHalf(regime.subdomain, var)
    for sub in newSplit:
      newProgram = optimize(sub, program, target)
      newRegime = newRegime ∪ {(sub, newProgram)}
    newCost = cost(newRegime)
    candidates = candidates ∪ (newRegime, newCost)
  // regime with the smallest cost
  bestRegime, bestCost = sortByCostAsc(candidates).head
  // check for improvement
  if bestCost < oldCost:
    if depth < maxDepth
     return simpleSearch(bestRegime, bestCost, depth)
    else:
     return bestRegime // the depth is exhausted
  else:
    return regime // previous regime

bUpCost = cost(bUpRegime)
return simpleSearch(bUpRegime, bUpCost, 0)
```

(a) Top-down phase with simple search

```
def geneticSearch(regimes, counter):
 candidates = []
 for r in regimes:
   newCost = cost(r)
   candidates = candidates∪{(r, newCost)}
   sortedRegimes = sortByCostAsc(candidates)
 if counter > maxGenerations:
   bestRegime, bestCost = sortedRegimes.head
   return bestRegime
 else:
  nextPopulation = []
  for i in range(0 until populationSize):
   // mutate regimes
   regToMutate = rankedChoice(sortedRegimes)
   vars = regToMutate.inputVars
   sortedVars = sortByRangeWidth(vars)
   varToMutate = rankedChoice(sortedVars)

   newSplit = splitRandomly(regToMutate,varToMutate)
   newProgram = optimize(newSplit,program,target)
   newRegime = (newSplit, newProgram)
   nextPopulation = nextPopulation ∪ newRegime

  return geneticSearch(nextPopulation, counter+1)
```

(b) Genetic search procedure

Fig. 9. Top-down phase with alternative regime search procedures.

3.2.1 Simple Search. Due to this inherent complexity, our simple search procedure attempts to find a lower-cost regime using a heuristic: split only along one variable in the middle of its range.

The pseudo-code for the top-down phase with the simple search is shown in the Figure 9. The algorithm attempts to split across each variable separately, then selects the split with the lowest cost and disregards the others. As a result, at every step the algorithm splits along one variable that provides the largest cost improvement. Such a strategy allows us to create regimes that are at least as good as the starting point (according to the cost function).

The algorithm terminates if it found a regime, upon which it cannot improve by splitting further. In general, the top-down algorithm is not guaranteed to terminate, therefore we limit the depth of splitting by a constant `maxDepth`.

For our running example the inferred regime has not changed after applying the simple search (see Figure 7(b)), splitting either of sub-domains in the middle did not improve the cost.

3.2.2 Genetic Search. Note that the regime returned by the top-down phase is not necessarily optimal. As illustrated by our running example, the search can give up too early, when a single split exactly in the middle of a variable's range does not improve the cost, but a different (potentially deeper) split would. To allow different splits and help overcome the local minima problem, we propose an alternative search procedure `geneticSearch` from Figure 9(b) that uses randomization in form of a genetic algorithm. Unlike the simple search, the genetic approach allows for multiple possible regimes to exist in parallel and selects sub-domains to be split and a split point randomly.

We instantiate the genetic algorithm framework in the following way: a regime represents an individual to be mutated, a collection of regimes is a population, and splitting the range of a

variable is a mutation. As in the simple top-down search we start from the regime returned by the bottom-up phase, this regime forms the initial population. In every generation, the algorithm chooses a regime and variable that will be mutated using ranked choice [45]. First, all regimes are sorted based on their cost and one is selected. Inside the selected regime, the variables are ranked according to the width of their range. Once the variable is selected, the algorithm splits the range at a random point (but at least 5% of the width).

Note that selecting a regime and variable to be mutated with ranked choice is simply an instantiation of randomness and can be replaced by any other heuristic. Because of the randomness and multiple regimes existing in parallel, the genetic search is less likely to get stuck in a local minimum, compared to the simple search that at each iteration keeps only one regime with the smallest cost and splits sub-domains exactly in the middle of variable's range.

We use a population size of 10 and repeat the loop for 10 generations (we have not observed a noticeable improvement in the results with larger values). On the final population the algorithm performs a post-processing step that merges identical neighbors using the `mergeSameBodies` function from bottom-up algorithm.

Starting from the results of the bottom-up phase the genetic top-down phase has inferred a regime for our running example shown in Figure 7(c).[3] While it is largely similar to the results of the bottom-up phase with and without the simple top-down search, a slight shift of the sub-domain bounds (bottom left corner) allows a sound optimizer to prove smaller error bounds and assign lower precision, thus further increasing the performance gain by 10%. A version of `carthesianToPolar_radius` with the resulting regime in Figure 7(c) runs 54% faster than the whole-domain optimized version.

3.2.3 Input Regime. The quality of regimes produced by the top-down phase with both simple and genetic search clearly depends on the starting point, i.e. input regime. While it is also possible to start from the whole domain (the program's initial input ranges) even with randomization the top-down phase is likely to give up too early when it gets stuck in a local optimum. To illustrate this, we have applied the top-down phase with both search procedures to our running example's whole domain. It inferred 2 sub-regimes shown in Figure 8, both of which are using at least 5 variables in `quad` precision. Even though with more fine-grained regime it is possible to use lower precision (as illustrated in Figure 7), the top-down phase was not able to find a lower-cost regime with one split. The bottom-up phase, on the other hand, has already explored the domain up to a certain depth. For many benchmarks this preliminary exploration is sufficient to overcome the local optimum.

3.3 Code Generation

Once a regime was found, our tool Regina generates code that uses conditional branches to select the appropriate expressions at runtime. A naive approach to generating the output code for a regime of size n would be to output a linear succession of conditional statements that cover exactly one sub-regime each. However, this approach requires $O(kn)$ number of tests, where k is the number of input variables, and n is the size of a regime.

Instead, Regina generates nested conditional statements leading to the individual regime bodies, where in each test we check the value of only a single variable. The asymptotic behavior of

[3]Our heuristic merging strategy could not merge two neighboring sub-regimes with expression A in the bottom-left corner. `mergeSameBodies` merges along each variable only once, here, first `x`, then `y`. When merging along `x`, the sub-domain $x \in [1.0, 10.9]$, $y \in [1.0, 19.8]$ was further split along `y` and therefore did not satisfy the definition of a neighboring sub-domain.

the number of tests performed is now $O(log(n))$, keeping the cost of conditional branches low, especially since every test itself is relatively cheap.

While for a single input variable it is always possible to generate one path for each sub-regime, this does not hold in general for multivariate functions. By generating nested conditionals, we generate code with more paths than the number of sub-regimes. That is, a sub-regime may be described by a union of several paths. Regina generates the conditional branches one input dimension at a time, starting with the variable for which there exists a sub-regime with the largest sub-domain.

For mixed-precision tuning, code generation furthermore needs to account for the fact that different sub-regimes may assign different precisions to the function's input and output variables. To preserve soundness, Regina assigns for each input variable and the return expression the highest precision that one of the sub-regimes has assigned. For each regime part where the input or return precisions do not exactly match the upper bound of all input and return precisions, we introduce downcasts. Note that this casting procedure is also accounted for in the cost function in order to penalize inferred regimes that necessitate many casts.

4 OPTIMIZATIONS

We instantiate regime inference in our implemented tool Regina with two optimizations: mixed-precision tuning and rewriting that optimize for average running time and worst-case absolute error, respectively. Note that our regime inference algorithms can be instantiated with any optimization objective. For instance, one could optimize the regimes with respect to worst-case performance, or average rounding error. The only adjustment necessary to change the objective is to provide an appropriate cost function.

4.1 Regime Inference for Mixed-Precision Tuning

For mixed-precision tuning we consider the two existing openly available sound tuning tools Daisy and FPTuner.

Daisy. We first instantiate Regina with Daisy's mixed-precision tuning routine [12], by calling Daisy as the `optimize` function. Daisy uses delta-debugging, a kind of divide-and-conquer algorithm, to search through different mixed-precision assignments, and calls a dataflow analysis to compute the rounding error of each.

FPTuner. Separately, we instantiate Regina with FPTuner's mixed-precision optimization [9], which uses a fundamentally different technique. FPTuner formulates the search for a mixed-precision assignment as an optimization problem, which it solves using a sound branch-and-bound interval solver. While this technique often produces better results (programs with lower running time) [12], the tuning process is also more expensive than the one in Daisy.

Cost Function. For instantiating our regime inference algorithms, we further need a cost function that reflects the optimization objective and that will guide the search and compare the quality of regimes. We choose to optimize the *average performance* of a program and assume that a program's inputs are uniformly distributed in the input domain. We assume a uniform distribution for convenience and simplicity, and note that it is possible to take into account a different distribution by adjusting the parameters of the cost function. We optimize for average, instead of worst-case, performance, because often one of the sub-regimes will need to use the highest precision, and thus no optimizations would be possible under a worst-case metric. That said, it is possible to account for the worst-case execution time by estimating it additionally and pruning the regimes that do not satisfy a constraint.

To estimate the (abstract) average performance of a program with regime, our cost function first computes an abstract *arithmetic cost* on each individual sub-regime. We use the term arithmetic cost to denote the performance of a floating-point expression without branches (i.e. on a single sub-domain). To compute the arithmetic cost, we use Daisy's existing simple mixed-precision cost function. It assigns to each 128-bit arithmetic and cast operation twice the cost of the same operation in 64 bits, and has been shown to work well for mixed-precision tuning between double and quad precision [12]. Note that it is also possible to tune between any other two precisions (e.g. 64 and 32 bits), provided a suitable arithmetic cost function. For computing the arithmetic cost, we are deliberately using an existing cost function previously shown to be adequate, as it is Regina's parameter and not a contribution of this work.

Since the goal is to increase average performance, next, the cost function computes a weighted average arithmetic cost of each sub-regime. Finally, the cost function adds an offset for the number of sub-regimes to account for branching. The final cost of a regime is thus computed as follows:

$$cost_{mp}(regime) = \sum_{i=1}^{n} w_i A_i + (n-1)$$

where n is the number of sub-regimes, w_i is an ith sub-regime's weight that corresponds to the sub-domain's volume normalized to the whole domain's volume,[4] and A_i is an arithmetic cost of the ith sub-regime. Even though evaluation of branching conditions is cheap, we still add a small offset $(n-1)$ to avoid inferring branches with negligible performance improvements.

4.2 Regime Inference for Rewriting

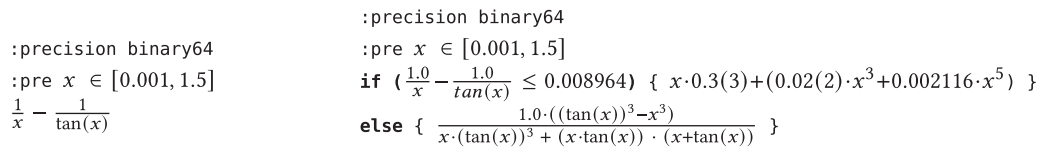
For rewriting, Regina calls Daisy's rewriting routine [12] as the `optimize` function. This optimization searches for an order of evaluation that is equivalent to the original expression under a real-valued semantics, but for which Daisy can prove a smaller rounding error bound (using its sound analysis). Since floating-point arithmetic does not satisfy common real-valued identities such as associativity and distributivity, reordering a computation in general leads to different results (and roundoff errors), even though the expression is equivalent under the reals. Daisy searches through the different evaluation orders using a genetic algorithm, applying real-valued identities as the mutation operation.

The goal of our regime inference for sound rewriting is to minimize the *worst-case rounding error* across sub-regimes. Accordingly, Regina uses a regime's maximum rounding error as the `cost` function. First, the cost function computes the arithmetic cost of an individual sub-regime. We use Daisy's worst-case rounding error analysis with the interval abstract domain to bound variables' ranges and affine arithmetic for errors. The overall cost is the maximum error seen across all sub-regimes:

$$cost_{rw}(regime) = \max_{i \in [1,n]} err_i$$

where n is the number of sub-regimes, and err_i is the worst-case absolute rounding error of the i-th sub-regime. Since we are optimizing for accuracy and not performance, and testing inputs' bounds does not affect accuracy of the computed value, we do not add any cost to prune additional branches. The regime inference algorithms themselves limit a total number of sub-regimes, so the resulting program will not have unreasonably many branches, and the branches generated can be evaluated efficiently. If needed, the cost function can straight-forwardly be extended to also account for the increased running time, however.

[4]For non-uniformly distributed inputs the weight w_i can reflect the probability of the i-th sub-regime being executed.



(a) Input expression (b) Herbie's optimized expression with regime

Fig. 10. NMSE-example-3.9 benchmark.

5 EXPERIMENTAL EVALUATION

We evaluate Regina on a standard benchmark set for floating-point analysis, and compare sound optimizations with and without regime inference. In particular, we focus on the following research questions:

RQ1: Does regime inference improve over whole-domain optimizations?
RQ2: Is the two-phase approach beneficial over each one separately?

5.1 Benchmarks

We evaluate regime inference on the FPBench benchmark set [11], a standard benchmark set for floating-point verification and optimization tools. For those benchmarks that originally contain loops, we generate a new version that consists of the loop body only (i.e. corresponds to one loop iteration). We exclude benchmarks that contain conditional statements, as well as benchmarks for which Daisy is not able to compute a roundoff error, e.g. when Daisy's analysis is not precise enough to show that a division is safe, i.e. does not divide by zero.

Many FPBench benchmarks already come with preconditions that bound the ranges of inputs. We use these as the initial domains for our optimization. When a precondition is missing or does not provide a closed range for all variables, we add input range bounds ourselves. For a few benchmarks, Daisy is not able to compute the roundoff error for the original precondition, but it is able to do so for a slightly modified—more constrained—one. In these cases, we consider the modified precondition for our experiments. In total, we consider 100 out of the 131 FPBench benchmarks, including 32 that contain elementary function calls and 15 that contain square root operations.

The existing and chosen input variable domains cover realistic preconditions, but are relatively small in the sense that they do not contain e.g. very large values (double floating-point precision supports exponents of up to 2^{1023}). With such preconditions, most of the benchmarks are numerically stable in the sense that the committed rounding errors are not very large, as computed by state-of-the-art sound rounding error analysis tools [13, 42].

5.2 Comparison with Herbie

Regina is the first tool that infers regimes for *sound* floating-point optimizations, i.e. those that guarantee that the rounding error of an optimized program does not exceed a specified bound. In contrast, todays state-of-the-art tools that infer regimes [37, 44, 46] use *dynamic analysis* to estimate rounding errors and therefore do not provide rounding error guarantees. Hence, there is no regime inference tool that we can directly compare to.

For completeness, we nonetheless perform a comparison with the dynamic analysis-based tool Herbie [37] that is closest to Regina in terms of the optimization that is being applied. Herbie's goal is to reduce (large) rounding errors by rewriting an arithmetic expression. Error guarantees aside, the goal is similar to when Regina is instantiated with Daisy's rewriting optimization. Unlike Regina, Herbie can split the domain only along a single variable. First, Herbie randomly samples

points and identifies which inputs cause large rounding errors, then it isolates these inputs into a sub-domain, and, when possible, improves the errors on this sub-domain with rewriting. Unlike Daisy, Herbie rewrites not only using real-valued identities, but also polynomial approximations (that are not real-semantics preserving).

Two other tools—AutoRNP [46] and the tool by Wang et al. [44]—are further away from Regina's goal. Like Herbie, they identify large rounding errors using a dynamic analysis, but their rewrite rules are more specialized or do not preserve real-valued semantics, i.e they only use approximations. Since the more specialized rules are largely not applicable to the general-purpose FPBench benchmarks, and it is not meaningful to compare error bounds obtained on semantically different expressions, we do not compare Regina's results with AutoRNP and the tool by Wang et al.

We run Herbie on all of our benchmarks four times to account for randomness, since it is using heuristic search and randomly sampled inputs. Herbie created regimes only for two benchmarks out of 100. In all four runs, Herbie created a regime for the *nmse_example_3.9* benchmark, the example regime is shown in Figure 10(b) (exact output slightly differs between the runs). Additionally, in one of the runs Herbie also found a regime for a second benchmark, *nmse_example_3.3*. For both benchmarks Herbie used rewrite rules that do not preserve real-valued semantics, thus, we cannot compare the optimized expression's rounding error with Regina's results (the same reason we do not compare with AutoRNP, Wang et al.).

These limited results are not particularly surprising, given that Herbie's stated goal is to repair *large* rounding errors—numerical instabilities. The results confirm that regime inference for—especially sound—floating-point optimizations of numerically stable code is missing.

We conclude that Herbie (and the other existing repair techniques [44, 46]) are complementary to Regina's goal: they can be used to first repair a program with large errors, so that Regina can optimize branches of the resulting program.

5.3 Experimental Setup

5.3.1 Mixed-Precision Tuning. The goal of mixed-precision tuning is to reduce the running time of an arithmetic expression as compared to a uniform precision implementation, while nonetheless meeting a user-provided error bound. For our evaluation, we thus have to define target error bounds, the precisions that we consider for tuning, as well as a suitable comparison baseline.

Following previous work,[5] we generate two sets of *target error bounds*. We first compute the rounding error for a given benchmark assuming uniform 64 bit double precision, and then multiply this error by 0.5 or 0.1 to obtain the target error bound. We choose two error bounds for each benchmark, because different bounds provide for different optimization opportunities. Choosing a smaller target error (using the factor 0.1), we generally expect less opportunities for mixed-precision tuning, and less improvements w.r.t. a uniform precision baseline. We will denote the benchmark set with error factor 0.5 by *half-double benchmarks*, and the benchmark set with factor 0.1 by *order benchmarks* (for an order-of-magnitude smaller error).

For comparison with Daisy's mixed-precision tuning, we compute the 64 bit double precision errors using Daisy, and for the comparison with FPTuner we compute the baseline errors correspondingly with FPTuner (since they use different techniques, the errors generally differ). It is not a goal of this paper to compare Daisy's or FPTuner's tuning, rather we want to show that regime inference is beneficial for both techniques.

[5]It has been observed that mixed-precision tuning is most useful when the target error bound is *just* below a uniform precision error [9, 13].

As in previous work [9], we consider mixed-precision tuning with *double and quad precision*, where quad is implemented by the GCC quadmath library [1]. The goal is to improve the running time over a uniform quad precision implementation of each benchmark.

For hardware platforms where single and double precision (32 and 64 bit) have different running times, tuning would be equally possible (with an appropriate cost function).

We compare the running time of programs generated by Regina against the running time of programs generated by Daisy's mixed-precision tuning. To ensure a fair comparison, we run Daisy's tuning using its subdivision method for computing ranges, using the same number of subdivisions as in the bottom-up phase. By doing so, we avoid seemingly improving over Daisy simply by using a more accurate range computation method. We compare Regina instantiated with FPTuner's mixed-precision tuning routine against FPTuner alone.

Mixed-precision programs are generated as C code that is compiled with `g++` 9.3 with the flags `-O2 -fPIC` and whose running time we measure using C's `high_resolution_clock` on 10^6 uniformly distributed random inputs. We repeat the measurement three times and take the average of those three runs for comparisons. We compute the *improvements* as (baselineTime - regimeTime)/baselineTime. We checked that the such computed performance improvements are accurate to within 0.02, hence we count a benchmark's performance as improved if the improvement is larger than 0.02.

5.3.2 Rewriting. We evaluate regime inference instantiated with Daisy's rewriting and compare the accuracy improvements w.r.t. to rewriting without regimes. Similarly as for mixed-precision tuning, we run Daisy's vanilla rewriting with the subdivision method for computing the ranges for a fair comparison. We compute the improvement in worst-case absolute error as $(baselineError - regimeError)/baselineError$.

5.3.3 Hardware Details. Because FPTuner requires Ubuntu, we run our mixed-precision tuning experiments on a compute cluster node with a dual-core Intel Xeon E5 v2 processor at 3.3 GHz and 16x16GB RAM running Ubuntu 16.04.7. We run the experiments with the rewriting optimization on a Mac mini with an 6-core Intel i5 processor at 3 GHz with 16 GB RAM running macOS Catalina, because the rewriting optimization runs in parallel and runs significantly faster on a 6-core machine.

We set a timeout of 30min per benchmark for all experiments.

5.4 RQ 1: Improvements over Whole-Domain Optimizations

Table 1 summarizes our experimental results for the three different optimizations: Daisy's mixed-precision tuning with Daisy, FPTuner's mixed-precision tuning and Daisy's rewriting. We have marked in bold the overall best results. For FPTuner, we report only results of the first *or* the second phase of our algorithm alone, because the running time of FPTuner's mixed-precision tuning is very high, and running the two-phase algorithm led to timeouts for most benchmarks.

Regina improves running time over Daisy's mixed-precision tuning for 73 half-double benchmarks with an average improvement of 65.7%, and improves 46 order benchmarks with an average improvement of 65.6%. Regina also improves over FPTuner's mixed-precision tuning for up to 31 half-double benchmarks with an average improvement of 52.2%, and for 18 order benchmarks with an improvement of 56.3%. The number of order benchmarks improved is lower, as expected, since a smaller error bound provides less opportunities for optimizations.

Regina with rewriting is able to improve the worst-case error for 62 out of 100 benchmarks with an average improvement of 54.4%. That is, with regime inference, we are able to essentially half the optimized (proven) error at compile time.

Table 1. Summary Statistics for different Optimizations, Comparing Regime Inference Against Optimizations without Regimes

	method	improv. >0.02	avrg. improv.	# best	regime size >1	avrg. regime size	avrg. runtime (s)	TO
Daisy mixed-tuning	*half-double*							
	bottom+genetic	**73**	**65.7%**	**54**	57	5.3	67.8	14
	bottom+top	74	64.4%	56	62	5.6	50.0	14
	bottomUp	74	63.5%	39	63	5.2	52.2	13
	topDown	64	60.6%	35	60	3.9	66.5	11
	genetic	72	62.5%	37	60	3.1	93.9	9
	order-error							
	bottom+genetic	**46**	**65.6%**	**35**	52	4.8	160.5	21
	bottom+top	45	65.6%	28	55	5.0	141.9	24
	bottomUp	47	58.5%	21	56	4.6	133.9	21
	topDown	40	52.4%	21	37	2.5	55.4	10
	genetic	45	61.1%	28	38	2.0	181.7	11
FPTuner	*half-double*							
	bottomUp	**31**	**52.2%**	**-**	50	7.8	665.6	23
	topDown	27	44.5%	-	30	2.7	677.8	19
	order-error							
	bottomUp	**18**	**56.3%**	**-**	30	5.3	656.6	35
	topDown	18	43.3%	-	25	3.2	574.1	18
Rewriting	**bottom+genetic**	**62**	**54.4%**	**59**	45	7.2	383.5	15
	bottom+top	53	40.5%	48	48	7.4	191.4	7
	bottomUp	43	44.3%	43	45	7.0	279.8	7
	topDown	43	52.5%	39	56	7.9	212.0	11
	genetic	58	48.1%	41	54	4.7	306.5	11

Column 2 gives the number of benchmarks for which there is an improvement over the optimized baseline without regimes, column 3 gives the average improvement over those benchmarks, column 4 gives the number of benchmarks for which a method produces an improvement that is within 2% of the best result among all methods. We do not report the number of the best results per method for FPTuner (marked '-'), as it is not meaningful for comparing only two methods. Columns 5 and 6 give the number of benchmarks where generated regime has multiple sub-regimes, and the average size of regimes. Columns 7 and 8 give the average running time of regime inference and the number of benchmarks that timed out.

The improvements by regime inference that we report in Table 1 are w.r.t. to the *already optimized baseline*. For comparison, Daisy's vanilla mixed-precision tuning without regimes improves performance over a *uniform quad precision baseline* by only 28% and 25%, respectively for half-error and order benchmarks, and Daisy's rewriting without regimes improves accuracy w.r.t. the *original* expressions by only 13%. We conclude that regime inference with Regina provides significant performance improvements over mixed-precision tuning without regimes, as well as accuracy improvements over rewriting.

Details: Mixed-Precision Tuning. In fact, e.g. the bottom-up approach generates *uniform double* precision code for 24 half-double benchmarks, i.e. does not use mixed precision at all. The reason why Daisy was not able to discover this uniform precision is that even though we run Daisy with the interval subdivision method for computing accurate ranges, the optimization itself nonetheless considers the entire domain at once, which leads to over-approximations. For the order bench-

marks, Regina generates uniform double precision code for 6 benchmarks. Hence, an additional side-effect of regime inference is that it reduces inherent over-approximations of the static analysis for individual optimizations.

Details: Rewriting. The bottom part of Table 1 further shows that the number of benchmarks with regime size greater than one, i.e. with several sub-regimes, is smaller than the number of benchmarks improved overall. This is due to the fact that during regime inference, the verification is performed on smaller sub-domains, which leads to a smaller overall computed error bound, which in turn may help to discover a suitable rewriting. Since Regina still merges sub-regimes with equal expressions in the end, we may end up with just a single expression. Thus, the 'on-demand' splitting that the methods bottom+genetic, bottom+top and genetic perform helps to effectively find sub-domains for which suitable rewriting can be found (and proven).

5.5 RQ 2: Evaluation of Two-Phase Approach

Table 1 also lists a number of variations of regime-inference methods. Our full two-phase algorithm (as described in Section 3) runs the bottom-up then the top-down phase with genetic or simple search and is denoted by 'bottom+genetic' and 'bottom+top', respectively. Furthermore, we evaluate each of the two phasess separately: 'bottomUp' method stands for applying the bottom-up phase alone, and 'topDown' and 'genetic' methods are results of applying the top-down phase with a corresponding search procedure to the whole specified input domain (as opposed to resulting regime of the bottom-up phase). Note that we limit the number of regimes that the top-down and genetic methods are allowed to consider to the number of subdivisions that the bottom approach generates for a fair comparison.

We compare the performance of these methods visually in the cactus plots in Figures 11 and 12, for Daisy's mixed-precision tuning and rewriting optimizations, respectively. That is, we have sorted the performance and accuracy improvements for each method individually, hence vertically aligned points do not always correspond to the same benchmark. Values below 0.0 in Figure 11 correspond to timeouts and slowdowns. For clarity, Figure 12 shows only those benchmarks for which one of the methods provides some accuracy improvement.

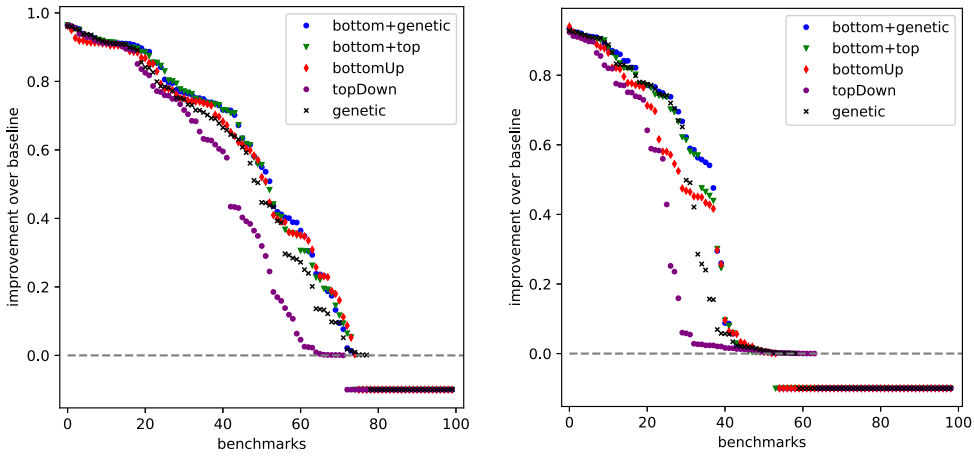
Overall, we observe that the combined approach (bottom+genetic and bottom+top) performs better than each phase of the algorithm alone (bottomUp, topDown and genetic). The top-down phase with the simple search procedure alone (topDown method), while still outperforming Daisy, performs worst overall. Our hypothesis is that it gets stuck in a local optimum, whereas the top-down phase with genetic search and combined phases overcome these local optima thanks to randomization and an initial exploration of the domain.

The performance of the combined two-phase approach comes at the expense of increased running time to compute the regime inference, and correspondingly also more timeouts (>30min). We have not observed memory to be an issue for Regina.

Since the genetic search procedure relies on randomization, we evaluated the influence of different random seeds on the generated performance over three runs. We observe that the variation in generated performance improvements is small (e.g. averages are within 2%), hence we conclude that the genetic method is able to improve performance reliably.

5.6 Summary

Our experiments confirm that our regime inference algorithm is general with respect to floating-point optimizer and optimization. Our regime inference approach does not rely on a particular technique used by the optimizer, and is equally applicable to the two (and the only publicly available) current state-of-the-art sound floating-point optimizers Daisy and FPTuner. Furthermore, it



(a) Half-double benchmarks (b) Order benchmarks

Fig. 11. Performance improvements of Regina over Daisy's mixed-precision tuning (cactus plot).

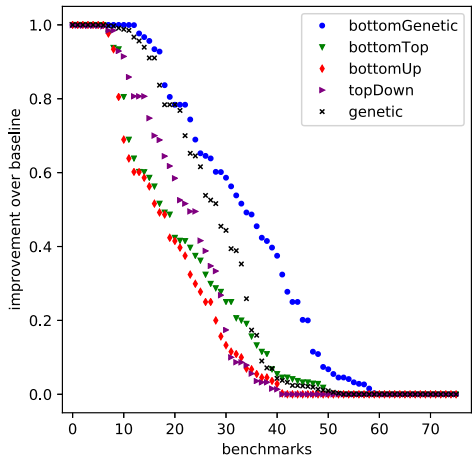


Fig. 12. Accuracy improvements of Regina over Daisy's whole-domain rewriting optimization (cactus plot).

provides a benefit for mixed-precision tuning both with Daisy and FPTuner that use very different techniques. Additionally, we were able to instantiate Regina for optimizations with different objectives—optimizing average performance with mixed precision tuning and optimizing worst-case rounding error with rewriting. For both optimizations, regime inference is able to provide improvements for a significant portion of the benchmarks—over a baseline that has already run an optimization.

6 RELATED WORK

Regime Inference. As discussed before, closest to our work are tools Herbie [37], AutoRNP [46] and the tool by Wang et al. [44]. They use a dynamic analysis to locate inputs with large rounding errors and based on these, infer regimes on which different types of repairs are applied: piecewise-quadratic or Taylor-based approximations [37, 46], or expression rewrites based on real-valued

identities [37, 44]. By relying on a dynamic analysis, these techniques fundamentally target and generate a different kind of regime. Specifically, for numerically stable expressions, i.e. those where rounding errors do not vary widely, it is—by definition—difficult for dynamic analysis to identify problematic inputs and thus to find partitions.

It is also not straight-forwardly possible to use a dynamic analysis to determine input domain partitions and then to run a sound optimization technique. Since sound tools inherently need to use abstractions, they will likely only be able to prove (and optimize) very different input partitions than the one identified by dynamic analysis.

Further Related Work. Sound polynomial approximations of elementary functions are often generated as piece-wise polynomials [16, 27, 29], e.g. with binary search. This is also a type of specialized regime inference, but limited to a single variable.

Partitioning of programs' input domain is widely used by sound verification tools to reduce the over-approximation on error abstractions. Existing techniques apply to floating-point programs [13, 20] as well as a mixture of floating-point code with bit-level operations [21, 31, 33].

We instantiated our regime inference for mixed-precision tuning with optimization routines of Daisy and FPTuner. Daisy and FPTuner are using sound dataflow analysis and branch-and-bound optimizers to find their precision assignments. Alternatively, one might consider mixed-precision tuning that uses a combination of backward static analysis and SMT solving in Salsa [10]. For programs where soundness guarantees are not required, our regime inference could also be instantiated with optimizations guided by dynamic analysis, as in Precimonious [38], STOKE-Float [39] and others [22, 25, 28, 30, 40], or algorithmic differentiation in ADAPT [32]. Techniques applied to numerical programs in the context of approximate computing, such as arithmetic operations that with a certain probability return an erroneous value [34] may be another target for regime inference.

To increase performance of floating-point numbers for machine learning and high-performance computing researchers proposed the alternative representations Bfloat16 [43] and TensorFloat32 [36]. They use fewer bits for the mantissa than a single-precision IEEE-754 float [26], and therefore incur higher errors. It is recommended to combine Bfloat16 and TensorFloat32 with the IEEE-754 single-precision floats [36, 43] which fits the mixed-precision tuning objective.

A different flavor of finite precision is implemented in the posit number system [23]. Unlike IEEE-754 floating points, posits dynamically adjust the number of bits for precision depending on the represented value, but fundamentally also provide only finite precision. Dynamic precision will likely result in a different error distribution in a program's input domain. However, since our regime inference technique is independent of a particular error distribution, we expect it to also be useful with optimizations on posit-based programs.

7 CONCLUSION

In this paper, we have shown that regime inference is beneficial not only for repairing large floating-point rounding errors, but for sound floating-point optimizations targeting numerically stable code as well. Even though we consider relatively simple interval-based regimes, these have proven to be remarkably successful in optimizing the performance and accuracy of straight-line expressions. The success comes exactly because of this simplicity: interval-based regimes allow for efficient runtime checks and are well-supported by today's sound floating-point analyzers. We observe that the major cost in sound regime inference are the individual optimizations themselves, and we show that a combination of breadth-first and depth-first search provides a good tradeoff.

REFERENCES

[1] 2020. *GCC libquadmath*. https://gcc.gnu.org/onlinedocs/libquadmath/.

[2] 2020. *The GNU MPFR Library*. https://www.mpfr.org/.

[3] 2021. *ARM developer: Floating Point*. https://developer.arm.com/architectures/instruction-sets/floating-point.

[4] 2021. *EEMBC FPMark^{TM} Floating-Point Benchmark Suite*. https://www.eembc.org/fpmark/.

[5] Adolfo Anta, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. 2010. Automatic verification of control system implementations. In *International Conference on Embedded Software (EMSOFT)*. https://doi.org/10.1145/1879021.1879024

[6] 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/.

[7] Joachim Bard, Heiko Becker, and Eva Darulova. 2019. Formally verified roundoff errors using SMT-based certificates and subdivisions. In *Formal Methods (FM)*. https://doi.org/10.1007/978-3-030-30942-8_4

[8] Hans-J. Boehm. 2020. Towards an API for the real numbers. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3386037

[9] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous floating-point mixed-precision tuning. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3009837.3009846

[10] Nasrine Damouche and Matthieu Martel. 2018. Mixed precision tuning with salsa. In *International Joint Conference on Pervasive and Embedded Computing and Communication Systems (PECCS)*. https://doi.org/10.5220/0006915501850194

[11] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alex Sanchez-Stern, and Zachary Tatlock. 2016. Toward a standard benchmark format and suite for floating-point analysis. In *International Workshop on Numerical Software Verification (NSV)*. https://doi.org/10.1007/978-3-319-54292-8_6

[12] Eva Darulova, Einar Horn, and Saksham Sharma. 2018. Sound mixed-precision optimization with rewriting. In *International Conference on Cyber-Physical Systems (ICCPS)*. https://doi.org/10.1109/ICCPS.2018.00028

[13] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy - Framework for analysis and optimization of numerical programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/978-3-319-89960-2_15

[14] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *TOPLAS* 39, 2 (2017). https://doi.org/10.1145/3014426

[15] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. 2013. Synthesis of fixed-point programs. In *International Conference on Embedded Software (EMSOFT)*. https://doi.org/10.1109/EMSOFT.2013.6658600

[16] Eva Darulova and Anastasia Volkova. 2018. Sound approximation of programs with elementary functions. In *Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-030-25543-5_11

[17] Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. 2020. Scalable yet rigorous floating-point error analysis. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1109/SC41405.2020.00055

[18] Rémy Garcia, Claude Michel, and Michel Rueher. 2020. A branch-and-bound algorithm to rigorously enclose the round-off errors. In *Principles and Practice of Constraint Programming (CP)*. https://doi.org/10.1007/978-3-030-58475-7_37

[19] David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* 23, 1 (1991). https://doi.org/10.1145/103162.103163

[20] Eric Goubault and Sylvie Putot. 2011. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. https://doi.org/10.1007/978-3-642-18275-4_17

[21] Eric Goubault, Sylvie Putot, Philippe Baufreton, and Jean Gassino. 2007. Static analysis of the accuracy in control systems: Principles and experiments. In *Formal Methods for Industrial Critical Systems (FMICS)*. https://doi.org/10.1007/978-3-540-79707-4_3

[22] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. 2019. Auto-tuning for floating-point precision with discrete stochastic arithmetic. *Journal of Computational Science* 36 (2019). https://doi.org/10.1016/j.jocs.2019.07.004

[23] Gustafson and Yonemoto. 2017. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.: Int. J.* 4, 2 (2017). https://doi.org/10.14529/jsfi170206

[24] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2021. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*. http://vhosts.eecs.umich.edu/mibench//index.html.

[25] Nhut-Minh Ho, Elavarasi Manogaran, Weng-Fai Wong, and Asha Anoosheh. 2017. Efficient floating point precision tuning for approximate computing. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. https://doi.org/10.1109/ASPDAC.2017.7858297

[26] IEEE. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008).

[27] Anastasiia Izycheva, Eva Darulova, and Helmut Seidl. 2019. Synthesizing efficient low-precision kernels. In *Automated Technology for Verification and Analysis (ATVA)*. https://doi.org/10.1007/978-3-030-31784-3_17

[28] Pradeep V. Kotipalli, Ranvijay Singh, Paul Wood, Ignacio Laguna, and Saurabh Bagchi. 2019. AMPT-GA: Automatic mixed precision floating point tuning for GPU applications. In *Proceedings of the ACM International Conference on Supercomputing*. https://doi.org/10.1145/3330345.3330360

[29] Olga Kupriianova and Christoph Quirin Lauter. 2014. A domain splitting algorithm for the mathematical functions code generator. In *Asilomar Conference on Signals, Systems and Computers, (ACSSC)*. https://doi.org/10.1109/ACSSC.2014.7094664

[30] Michael O. Lam and Jeffrey K. Hollingsworth. 2018. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications* 32, 2 (2018). https://doi.org/10.1177/1094342016652462 arXiv:https://doi.org/10.1177/1094342016652462

[31] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying bit-manipulations of floating-point. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2908080.2908107

[32] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. ADAPT: Algorithmic differentiation applied to floating-point precision tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. Article 48. https://doi.org/10.1109/SC.2018.00051

[33] Antoine Miné. 2012. Abstract domains for bit-level machine integer and floating-point operations. In *ATx'12/WInG'12: Joint Proceedings of the Workshops on Automated Theory eXploration and on Invariant Generation*.

[34] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Object Oriented Programming Systems Languages & Applications (OOPSLA)*. https://doi.org/10.1145/2660193.2660231

[35] Mariano Moscato, Laura Titolo, Aaron Dutle, and Cesar Muñoz. 2017. Automatic estimation of verified floating-point round-off errors via static analysis. In *SAFECOMP*. https://doi.org/10.1007/978-3-319-66266-4_14

[36] NVIDIA. 2020. TensorFloat-32 in the A100 GPU Accelerates AI Training. (2020). https://blogs.nvidia.com/blog/2020/05/14/tensoroat-32-precision-format/.

[37] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2737924.2737959

[38] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, D. H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/2503210.2503296

[39] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2594291.2594302

[40] Kushal Seetharam, Lance Ong-Siong Co Ting Keh, Ralph Nathan, and Daniel J. Sorin. 2013. Applying reduced precision arithmetic to detect errors in floating point multiplication. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*. https://doi.org/10.1109/PRDC.2013.44

[41] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. 41, 1, Article 2 (2018). https://doi.org/10.1145/3230733

[42] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2015. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM*. https://doi.org/10.1007/978-3-319-19249-9_33

[43] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. 2018. A transprecision floating-point platform for ultra-low power computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. https://doi.org/10.23919/DATE.2018.8342167

[44] Xie Wang, Huaijin Wang, Zhendong Su, Enyi Tang, Xin Chen, Weijun Shen, Zhenyu Chen, Linzhang Wang, Xianpei Zhang, and Xuandong Li. 2019. Global optimization of numerical programs via prioritized stochastic algebraic transformations. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2019.00116

[45] Darrell Whitley. 1989. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms*.

[46] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient automated repair of high floating-point errors in numerical libraries. *Proceedings of the ACM on Programming Languages* 3, POPL (2019). https://doi.org/10.1145/3290369

Received April 2021; accepted July 2021