Kalpa Publications in Computing

Volume 5, 2018, Pages 19–30

Automated Formal Methods

Moving the Needle on Rigorous Floating-Point Precision Tuning*

Marek Baranowski, Ian Briggs, Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamarić, and Alexey Solovyev

University of Utah, Salt Lake City, Utah, U.S.A.
{baranows,ibriggs,wfchiang,ganesh,zvonimir,solovyev}@cs.utah.edu

Abstract

Virtually all real-valued computations are carried out using floating-point data types and operations. With increasing emphasis on overall computational efficiency, compilers are increasingly attempting to optimize floating-point expressions. Practical reasoning about the correctness of these optimizations requires error analysis procedures that are rigorous (ideally, they can generate proof certificates), can handle a wide variety of operators (e.g., transcendentals), and handle all normal programmatic constructs (e.g., conditionals and loops). Unfortunately, *none* of today's approaches can achieve this combination. This position paper summarizes recent progress achieved in the community on this topic. It then showcases the component techniques present within our own rigorous floating-point precision tuning framework called FPTuner—essentially offering a collection of "grab and go" tools that others can benefit from. Finally, we present FPTuner's limitations and describe how we can exploit contemporaneous research to improve it.

1 Introduction

We live in an era where an increasing number of safety-critical computations are carried out using floating-point arithmetic. It is also the era of the "pinched-off Moore's law" with all its concomitant forces egging us to seek computational efficiency, including modifying compilers to play with floating-point precision. Unfortunately, we have not proportionately grown our floating-point error analysis capabilities: students are sparingly educated [2], and the formal methods community around this enterprise is small and disconnected. While Kahan has characterized floating-point errors as "very rare, too rare to worry about all the time," in the same breath he also cautions us "yet not rare enough to ignore" [18].

Microprocessor vendors already offer us a cornucopia of options to skimp on floating-point precision, such as 16-bit floating-point arithmetic supported by ARM NEON [1] and Nvidia Pascal GPU [24]. FPGAs are gaining popularity due to their flexibility, and they will offer additional opportunities in this regard, as already foreseen [8]. These are tremendous opportunities to save on energy: we recently measured the *reduction* routine presented in Table 5 of

*This work was funded in part by NSF CCF 1704715, 1643056, 1552975, and 1421726.

our recent paper [7] as giving us a saving of 2,230 Joules when instantiated in 32 bits (single-precision) as opposed to 64 bits (double-precision) over 10^{11} invocations—nearly 2% of a laptop battery's capacity.

The capability of floating-point error analysis itself is of immense value within rigorous reasoning systems such as proof assistants. Rigorous precision tuning tools add an extra layer of tooling that iterates over different precision allocations till an error target is met. Such tuning support is offered by some recent efforts [9, 10] where they allocate different *homogeneous* (e.g., all 32, 64, or 128 bits—not a mixture) precision choices. It has been shown (e.g., [2, 6]) that *mixed-precision* allocation is often much better than homogeneous allocations. These authors do not provide a tool: all mixed precision allocations were explored manually. Automated mixed-precision tuning methods were introduced in recent efforts [19, 28, 27]. These researchers achieved substantial savings in terms of the number of double-precision words allocated while meeting stipulated error bounds on a given collection of test cases.

Unfortunately, all the aforesaid mixed-precision tuning efforts (manual and automated) provided their guarantees only on a few hundred test cases that a user supplies. It is easy to observe a violation of the stipulated error bounds on other inputs—even those inputs that may lie within the interval straddled by the given test inputs [7]. This makes the aforesaid solutions unusable in situations demanding rigorous guarantees.

We recently demonstrated how mixed-precision tuning that comes with rigorous guarantees can be achieved [7]. We also, for the first time, actually showed the energy savings accruing from such allocations across a wide variety of experiments. We observe energy savings for many mixed-precision allocations. We also are very careful to present the effect that the choice of the compiler used can have on the quality of the results, and how to "control" the flags of these compilers to achieve good results. All our results are reproducible by the community by downloading our FPTuner tool from Github at `https://github.com/soarlab/FPTuner`.

We now present a taxonomy of strengths and weaknesses of rigorous tools (including of FPTuner) and point out areas of cooperation that will help the field advance.

Error Analysis: The underlying error analysis methods must not generate overly conservative error estimates. When applied for precision tuning, such estimates can lead to excessive (and unnecessary) precision allocation. When applied for verification, it can result in unnecessary verification failures. The FPTaylor [31] approach underlying FPTuner has been shown to provide the tightest of rigorous estimates on a common class of examples, compared to existing rigorous tools.

Conditionals: Handling conditionals has been a vexing problem for researchers in this area [9, 31]. The key issue is that for virtually all practical programs, the round-off error introduced by the conditional expression can only be estimated within a certain tolerance, thus leading to a case analysis that involves incompatible control flows (then/else are both deemed possible) [25, 10]. Recently proposed rigorous round-off analysis methods incorporate techniques to handle such "unstable conditionals"; for instance, support for conditionals exists within PRECISA [25], Real2Float [22], and Rosa [10]. FPTaylor has been shown to generate far tighter error estimates than these tools—albeit on straight-line programs. Unfortunately, FPTaylor (and FPTuner that is based on FPTaylor) cannot deal with conditionals, and this forms a major area of improvement that we seek. Whether we can achieve the same tight bounds that FPTaylor produces on straight-line code even in the presence of conditionals remains to be seen.

Proof Certificates: Generation of proof certificates is supported by Real2Float [22], PRECISA [25], and FPTaylor. The other rigorous tools listed above do not produce proof certificates.

Variety of Operations: FPTaylor (and hence FPTuner) can handle a wide variety of operators that include non-linear and transcendental operators. Real2Float is the only other rigorous tool we are aware of that can handle these families of operators. FPTaylor's approach in this regard is to use a global optimization procedure while Real2Float employs a relaxation procedure based on semi-definite programming. These approaches help FPTaylor and Real2Float side-step a difficulty faced by other techniques that rely on SMT-based methods; this is because there are no well-developed SMT approaches to handle transcendentals.[1]

Mixed-precision Tuning: We have mentioned that Rosa [9, 10] performs only homogeneous precision allocation. By including an extra optimization loop based on quadratically constrained quadratic programming (QCQP), and supported by tools such as Gurobi [16], FPTuner is able to carry out rigorous mixed-precision tuning. We show [7] that in cases where Rosa recommends an all-128 allocation, we can in fact achieve a mixed 64/128 allocation, which has the distinct advantage that 64-bit precision is directly supported in hardware, thus dramatically reducing the overall runtime.

1.1 Moving the Needle on Mixed-precision Tuning

Our primary goal in this position paper is to facilitate advances in the area of rigorous mixed-precision tuning by offering the first comparative study that clearly lists the strengths and limitations of various tools in this area. Our secondary goal is to contribute ideas toward rigorous analysis methods for floating-point round-off error analysis by clearly describing FPTuner and its component technologies that can be used piece-meal in other tools. It is clear that thrusts in these areas should not remain isolated—a clear danger, given the small sizes of communities interested in these areas. Our comparative study of various tools in §1 suggests that each tool in this area stands to benefit from the others by directly borrowing a piece of technology and/or suitably adapting it.

This paper will now present FPTuner and its component technologies in sufficient detail so as to encourage other groups to try using this tool as well as borrow from its components:

- They may be encouraged to employ FPTaylor (the "engine" behind FPTuner) as a stand-alone error analysis facility. We would like to point out that FPTaylor has been released as a stand-alone tool on Github at `https://github.com/soarlab/FPTaylor`.
- They may be encouraged to use FPTaylor's global optimizer backend, namely Gelpia, for solving optimizations. Gelpia also enjoys a stand-alone release on Github at `https://libraries.io/github/soarlab/gelpia`.
- Last but not least, they may learn how FPTuner's tuning loop based on QCQP works. This may allow other groups to build similar precision tuning methods in their own framework.

Roadmap: In §2, we present the overall flow of FPTuner. In §3, we present a case study: the tuning of an unrolled Jacobi iteration scheme. In §4, we provide our concluding remarks,

[1] FPTaylor is also unable to handle discontinuous operators such as *abs* and *mod*; however, it does employ a smooth as well as conservative approximation to these functions, and therefore is able to handle these operations in practice—albeit with exaggerated error at the discontinuity.

including additional related work on rigorous precision tuning. We also present our plans to advance FPTuner by borrowing the best ideas from contemporaneous rigorous analysis tools.

2 Introduction to FPTuner

We provide an overview of FPTuner using a simple illustrative example, while also stepping through Figure 1—the workflow of this tool. Consider a simple expression given over reals: $\mathscr{E} = x - (x + y)$. Let A_{64} be an allocation vector that assigns double-precision (64 bits) to the three variable occurrences as well as the two operators in this expression. That is, $A_{64}[i] = \epsilon_{64}$ for $i \in 5$ where ϵ_{64} is the *machine epsilon* [13] for double precision. Using the approach of Symbolic Taylor Forms (which essentially goes by the round-off error serving as the "noise" around the ideal), we obtain the modeling expression $\tilde{\mathscr{E}}_{A_{64}}$ is $(x \cdot (1 + e_1) - (x \cdot (1 + e_3) + y \cdot (1 + e_4)) \cdot (1 + e_2)) \cdot (1 + e_0)$.. This method of obtaining floating-point error modeling expression is standard (has been rigorously established in many frameworks, including within HOL-lite recently [17]). In $\tilde{\mathscr{E}}_{A_{64}}$, each operator of $\mathscr{E}$ at position $i \in 5$ is associated with a distinct noise variable e_i, where $|e_i| \leq A_{64}[i]$. Note that keeping e_1 and e_3 that are associated with the two instances of x distinct gives a pessimistic error estimate, as the round-off errors are allowed to be uncorrelated.[2] Also notice that by setting all the noise variables to 0, we obtain the value of $\mathscr{E}$. Based on Symbolic Taylor Expansions [31], we can now obtain a formula describing the first order error due to these "noise" terms:

$$\left|\tilde{\mathscr{E}}_{A_{64}} - \mathscr{E}\right| \leq \sum_{i \in 5} U_{e_i} \cdot A_{64}[i]. \tag{1}$$

Figure 1 illustrates these steps. Here, the given expression $\mathscr{E}$ flows in at the top, and the modeling expression is obtained. The error bound expression T is obtained by applying FPTaylor. The $D()$ coefficients are the first partial derivatives with respect to the noise variables, and represents the first-order error introduced by the corresponding operator. The Gelpia optimizer finds the maximum of these first derivative expressions, thus obtaining their upper bounds which we designate using U.

The steps described thus far apply to the homogeneous precision case. For mixed precision allocation, we not only introduce the noise terms, but must also introduce an optional *type-casting* round-off step. This round-off step is necessary when descending from high precision toward lower precision. But since we do not know whether we are descending (or ascending) in precision till the full allocation is done, our formulation actually introduces a quadratic program that captures all these constraints.

The Gurobi optimizer of Figure 1 is the unique additional layer added by FPTuner. It handles the following details:

- It models the precision allocated at every operator site through variable c_i.
- It checks whether one operator at precision c_1 is feeding a second operator's operand position where the second operator is at a lower precision c_2; if so, it introduces a type-casting rounding step.
- It groups (based on user selection) precision allocations of multiple operators ("ganging step"). This is to permit the generation of vector instructions by picking a group of variables and requiring that their precision values be the same.

In summary, the workflow in Figure 1 indicates how the FPTaylor tool was extended to yield a precision tuner. The key in a nutshell is to treat the machine epsilons as variables

[2] It also permits these xs to be assigned different precision values.

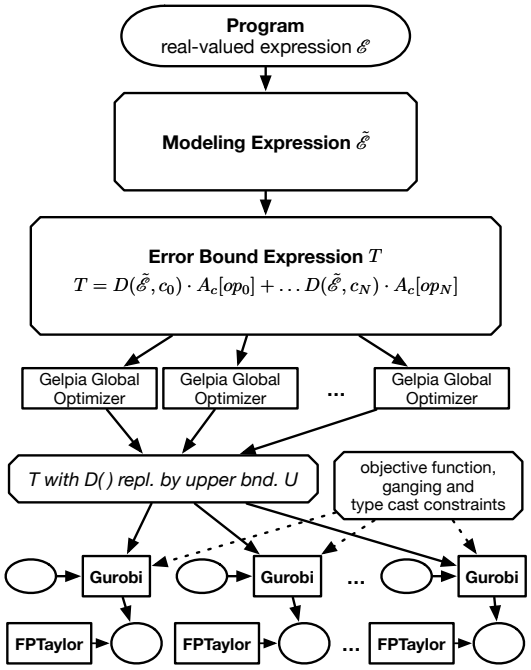

Figure 1: FPTuner workflow.

ranging over the desired range of actual machine epsilon constants for various precision values. We introduce a conditional casting term if a value flows from the regime of one machine epsilon (that of an operator) to the regime of another machine epsilon (operand) at lower precision. Gurobi then seeks an allocation to all machine epsilon variables that minimizes total error to be under a user-given target while meeting users' additional criteria that may include: (1) gang a selected set of operators, (2) keep the number of type-casting steps below some constant, and (3) limit the total number of type-casting steps.

The diagram also shows FPTaylor involved as a final checking step. Instead of computing both a first-order Taylor error and estimating the second-order error (as FPTaylor does), FPTuner takes the following shortcut: (1) it obtains only the first-order error, (2) it attempts the allocation, (3) it finally invokes FPTaylor at the end to re-check that the allocation abides by an FPTaylor run that *includes* the second order error estimate. In all our experiments, this short-cut has worked without the final FPTaylor check failing. (If it were to fail, we would simply tighten the error estimate with the second-order error estimate and rerun.)

3 Case Study: Tune Unfolded Jacobi

3.1 Example Description

Figure 2 is an example we will use to illustrate FPTuner and its actions. This is one of the largest examples run through FPTuner to date (in our paper [7], we only provide the final tuned result; here we provide a few additional details).

With this example we are symbolically unrolling a Jacobi solver and querying for error on one of the final terms. Since all the operations in this example are symmetric, we obtain the per element roundoff error as follows. First we create the input A (a 2d array of real values),

```
import tft_ir_api as IR

n = 3
unrolls = 2

low  =   1.0
high = 10.0

A = list()
for j in range(n):
  row = list()
  for i in range(n):
    row.append(IR.RealVE("a{}{}".format(i,j), 0, low, high))
  A.append(row)

b = list()
for i in range(n):
  b.append(IR.RealVE("b{}".format(i), 1, low, high))

x = list()
for i in range(n):
  x.append(IR.FConst(1.0))

g = 2

#j k = 0
#j while convergence not reached: # while loop
for k in range(unrolls): # replacement for while loop
  for i in range(n): # i loop
    sigma = IR.FConst(0.0)
    for j in range(n): # j loop
      if j != i:
        sigma = IR.BE("+", g, sigma,
                      IR.BE("*", g, A[i][j], x[j]))
        g += 1
    # end j loop
    x[i] = IR.BE("/", g, IR.BE("-", g, b[i], sigma), A[i][j])
    g += 1
  # end i loop
#j check convergence
#j k = k+1
# end while loop

print(x[0])
rs = x[0]
IR.TuneExpr(rs)
```

Figure 2: Python code to generate the FPTuner Jacobi query.

the b vector of real values, and the initial guess vector x comprised of the constant 1.0. The standard Jacobi algorithm is then performed, with FPTuner operations essentially building up the symbolic expressions of the computation.

3.2 Symbolic Taylor Forms

The input Jacobi query to FPTuner generates many sub-queries to FPTaylor such as illustrated in Figure 3. These in turn go through floating point error modeling via Taylor forms generated

```
rnd32((rnd32((rnd32(b0) - rnd32((rnd32((rnd32(0.0) + rnd32((rnd32(a10) *
rnd32((rnd32((rnd32(b1) - rnd32((rnd32((rnd32(0.0) + rnd32((rnd32(a01) *
rnd32((rnd32((rnd32(b0) - rnd32((rnd32((rnd32(0.0) + rnd32((rnd32(a10) *
rnd32(1.0))))) + rnd32((rnd32(a20) * rnd32(1.0))))))) / rnd32(a20))))))) +
rnd32((rnd32(a21) * rnd32(1.0))))))) / rnd32(a21))))))) + rnd32((rnd32(a20)
* rnd32((rnd32((rnd32(b2) - rnd32((rnd32((rnd32(0.0) + rnd32((rnd32(a02)
* rnd32((rnd32((rnd32(b0) - rnd32((rnd32((rnd32(0.0) + rnd32((rnd32(a10)
* rnd32(1.0))))) + rnd32((rnd32(a20) * rnd32(1.0))))))) / rnd32(a20))))))
+ rnd32((rnd32(a12) * rnd32((rnd32((rnd32(b1) - rnd32((rnd32((rnd32(0.0)
+ rnd32((rnd32(a01) * rnd32((rnd32((rnd32(b0) - rnd32((rnd32((rnd32(0.0) +
rnd32((rnd32(a10) * rnd32(1.0))))) + rnd32((rnd32(a20) * rnd32(1.0))))))) /
rnd32(a20))))))) + rnd32((rnd32(a21) * rnd32(1.0))))))) / rnd32(a21)))))))))) /
rnd32(a22)))))))))) / rnd32(a20)))
```

Figure 3: Example FPTaylor query generated by FPTuner.

```
((interval(1.0, 1.0) / a20) * (-((a20 * ((interval(1.0, 1.0) / a22) * (-((a02 * ((b0 - ((interval(0.0,
0.0) + (a10 * interval(1.0, 1.0))) + (a20 * interval(1.0, 1.0)))) * (interval(1.0, 1.0) / a20))))))))))
```

Figure 4: Example Taylor form.

by FPTaylor. These Taylor forms range from 6 to 1244 operators and 2 to 10 input dimensions for this Jacobi query. These are then handed to Gelpia for global optimization. A simple example of such a query is given in Figure 4.

As detailed in [31], FPTaylor can use two different models for rounding error. The simple model carries error terms with each operation modeled according to its precision. This approach generates a differentiable optimization query that can be handled by most mainstream global optimizers. The drawback to this approach, however, is that the model overestimates the round-off error.

We also define an improved rounding model that correlates error terms, thus modeling errors more tightly. A drawback of this approach is that the resulting optimization problems may involve discontinuous functions, and thus not amenable to most global optimizers.

We extended Gelpia to be able to handle these discontinuous queries. However, FPTaylor does not (yet) generate proof certificates for this improved rounding model.

3.3 Assessment of the Gelpia Optimizer

Gelpia utilizes the inclusion property of interval arithmetic. The output of any operation will be an over approximation so that all possible answers for the input ranges are represented in the output range. This approximation can be tight or loose depending on the exact values and operations. For instance if we say $x = [-5, 5]$ then $x * x$ will equal $[-25, 25]$ since the interval arithmetic only sees $[-5, 5] \times [-5, 5]$ and doesn't know that the two ranges given it are the same variable. If the computation is x^2, which is arithmetically equivalent to $x * x$, but has a much tighter bound in interval arithmetic of $[0, 25]$ since the computation given it is $[-5, 5]^2$. We use many such arithmetic substitution that maintain the rigorous nature of Gelpia's optimization, but leverage the underlying interval arithmetic for tighter bounds and faster convergence.

In addition to algebraic simplification, Gelpia uses a mixture of heuristics to accelerate the branch and bound algorithm: these include sampling points in a split domain to guess which branch is more likely to contain the maximum, estimating the derivative at these points to

prioritize steeper domains over flatter domains, and local optima finders to assist in the search. The sampling heuristic is weighted to give priority to boxes which appear to be consistently higher than other boxes over estimates of the derivative. This roughly means that a box containing the maximum and is flat is given more priority than a box with high derivatives which is likely pointing to the flat box. We find that this heuristic drives the search better to eliminate local bounds. Local optimum finding methods are used to quickly find (upper bounds on) local maxima to raise the branch bound. The branch and bound algorithm periodically informs the local methods of an approximate enclosure of the search space. Approximation of the derivative is found through reverse symbolic-differentiation computed in the Gelpia front end. The basic algorithm is outlined in Figure 5.

Since the reverse differentiation can create common subexpressions in the overall computation we also use common subexpression elimination and a SSA type internal representation to eliminate redundant computation. Once the reverse derivation and redundant expression elimination is performed the queries range from 42 to 2658 operators.

We guarantee that the maximum given by Gelpia is above the true global maximum of the function, respectively the minimum is below the true global minimum. The heuristics and simplifications help Gelpia to either find a closer estimate in a given time limit, or find the same estimate in a shorter period of time.

Compared with many other tools, we provide rigorous global optimization of functions containing discontinuous and transcendental functions. Another tool dReal [12] supports many of the same features of Gelpia, but occasionally produces non-rigorous answers, meaning it can produce a purported global minimum, which can easily be shown through sampling to not be the global minimum. However, for many queries dReal is faster than Gelpia, so its input could be useful in finding extrema quickly. Additionally, dReal supports constraints on the query permitting a more flexible query language, which Gelpia currently lacks.

Figure 6 presents the results of tuning the Jacobi example for three precision choices. The selected precision levels at various levels of the expression tree are as indicated.

4 Concluding Remarks

Additional Related Work: Space prevents us from surveying many other tools in this area; for completeness, here are some additional related efforts.

An SMT-LIB theory of floating-point numbers was first proposed by Rümmer and Wahl [29] and recently refined by Brain et al. [4]. There have been several attempts to devise an efficient decision procedure for such a theory [21, 3, 5], but currently most SMT solvers still do not support it. Recent efforts in rigorous floating-point error estimation are based on combinations of abstract interpretation and conservative range calculations. Melquiond et al. offer Gappa [11], a tool based on interval arithmetic. The tool FLUCTUAT [14] combines the error estimates obtained from input-domain subdivisions to improve the overall accuracy of error analysis. Graillat et al. [15] propose a tuning approach similar to Precimonious but use discrete stochastic arithmetic (DSA) for confirming the precision requirement. Tang et al. [32] propose a method that automatically searches for possible expression rewrites from a database of templates. Panchekha et al. [26] propose a method to rewrite expressions similar to Tang's approach. However, Panchekha's method can synthesize simple conditionals that can adaptively select different rewrites according to runtime inputs. Also, the objective of Panchekha's method is to reduce overall round-off errors on program outputs. Martel proposed an operational semantics governing the rewriting of program statements [23] for improving floating-point precision. This technique also takes into consideration standard compile-time techniques such as loop un-

```
function IBBA(f, x, x_tol, f_tol)
  f_best_low ← −∞
  f_best_high ← −∞
  Q ← PriorityQueue()
  Q.push(x)
  while Q ≠ ∅ do
    x_n ← Q.pop()
    fx_n ← f(x_n)
    f_best_low ← max(f_best_low, lower(fx_n))
    if upper(fx_n) < f_best_low or width(x_n) < x_tol or
        width(fx_n) < f_tol then
      f_best_high ← max(f_best_high, upper(fx_n))
      continue
    end if
    x_l, x_r ← split(x_n)
    Q.push(x_l)
    Q.push(x_r)
  end while
  return f_best_high
end function
```

Figure 5: Interval Branch-and-Bound Algorithm (IBBA) underlying Gelpia. Here, f is the function to optimize and x is the input domain (treated as a scalar here, but in general, is an N-dimensional rectangular domain). Parameters x_tol and f_tol are scalars used to suppress the *split* step when either the input or the output interval width are small.

rolling. Schkufza et al. [30] offer a Markov Chain Monte Carlo (MCMC) based method that searches for improved-efficiency compositions of instructions. Recently, Lee et al. [20] proposed a verification method that combines instruction rewriting and rigorous precision measurement.

Path Forward

This position paper brings together many recent efforts that cater to rigorous floating-point error estimation and precision tuning. We have described the components of FPTuner at sufficient depth to encourage other researchers to adopt its techniques. As for FPTuner itself, here is what we foresee as its immediate path forward:

- We plan to incorporate conditionals into FPTuner by employing the approach suggested in previous work [25].
- One of the tool bottlenecks is the time taken by Gelpia. We plan to research better heuristics and more aggressive parallelization methods to speed up this tool.
- Another interesting avenue would be to incorporate expression rewriting [26] as an additional step during precision tuning. The exact manner in which these techniques can aid each other (e.g., whether expression rewriting can reduce the need for precision tuning, or eliminate cases where precision tuning ends up allocating higher bit-widths) would be an important result to obtain.

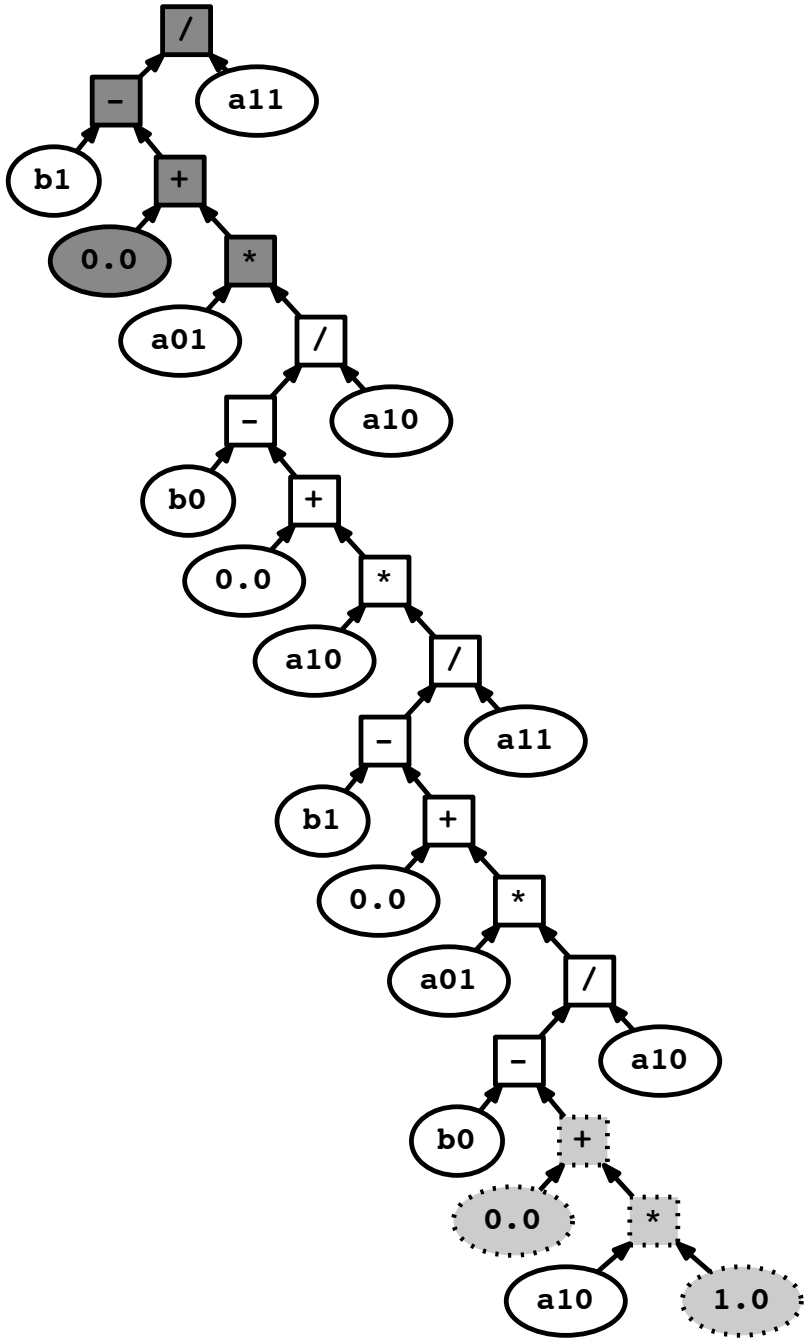

Figure 6: Tuning results for the Jacobi benchmark using three candidate precisions. The computation is represented as an expression tree, where ovals denote variables and constants, and squares denote operators. Dark (resp., light+dotted, white) ovals/rectangles denote single-precision (resp., double-, quad-) variables/operators. Note that we do not show the casting operators explicitly.

References

[1] ARM NEON General-Purpose SIMD Engine, 2016. Available at https://web.archive.org/web/20160410014559/https://developer.arm.com/technologies/neon.

[2] David Bailey and Jonathan Borwein. High-Precision Arithmetic: Progress and Challenges, 2013. Available at http://www.davidhbailey.com/dhbpapers/hp-arith.pdf.

[3] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding Floating-Point Logic with Abstract Conflict Driven Clause Learning. *Formal Methods in System Design (FMSD)*, 45(2):213–245, 2014.

[4] Martin Brain, Cesare Tinelli, Philipp Rümmer, and Thomas Wahl. An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic. In *Proceedings of the 22nd IEEE Symposium on Computer Arithmetic (ARITH)*, pages 160–167, 2015.

[5] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed Abstractions for Floating-Point Arithmetic. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 69–76, 2009.

[6] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance While Achieving 64-Bit Accuracy. *ACM Transactions on Mathematical Software (TOMS)*, 34(4), 2008.

[7] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Rigorous Floating-Point Mixed-Precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 300–315, 2017.

[8] George A. Constantinides, Peter Y. K. Cheung, and Wayne Luk. The Multiple Wordlength Paradigm. In *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 51–60, 2001.

[9] Eva Darulova and Viktor Kuncak. Sound Compilation of Reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 235–248, 2014.

[10] Eva Darulova and Viktor Kuncak. Towards a Compiler for Reals. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(2):8:1–8:28, 2017.

[11] Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Assisted Verification of Elementary Functions Using Gappa. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC)*, pages 1318–1322, 2006.

[12] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT Solver for Nonlinear Theories over the Reals. In *Proceedings of the 24th International Conference on Automated Deduction (CADE)*, pages 208–214, 2013.

[13] David Goldberg. What Every Computer Scientist Should Know about Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[14] Eric Goubault and Sylvie Putot. Static Analysis of Numerical Algorithms. In *Proceedings of the 13th International Static Analysis Symposium (SAS)*, pages 18–34, 2006.

[15] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuiliére. PROMISE: Floating-Point Precision Tuning with Stochastic Arithmetic. In *Proceedings of the 17th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN)*, pages 98–99, 2016.

[16] Gurobi Optimizer, 2016. Available at http://www.gurobi.com.

[17] Charles Jacobsen, Alexey Solovyev, and Ganesh Gopalakrishnan. A Parameterized Floating-Point Formalizaton in HOL Light. In *Proceedings of the 8th International Workshop on Numerical Software Verification (NSV)*, pages 101–107, 2015.

[18] Ronald T. Kneusel. *Numbers and Computers*. Springer, 2017.

[19] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. LeGendre. Au-

tomatically Adapting Programs for Mixed-Precision Floating-Point Computation. In *Proceedings of the 27th International Conference on Supercomputing (ICS)*, pages 369–378, 2013.

[20] Wonyeol Lee, Rahul Sharma, and Alex Aiken. Verifying Bit-Manipulations of Floating-Point. In *Proceedings of the 37th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 70–84, 2016.

[21] Miriam Leeser, Saoni Mukherjee, Jaideep Ramachandran, and Thomas Wahl. Make It Real: Effective Floating-Point Reasoning Via Exact Arithmetic. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, pages 117:1–117:4, 2014.

[22] Victor Magron, George Constantinides, and Alastair Donaldson. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Transactions on Mathematical Software (TOMS)*, 43(4):34:1–34:31, 2017.

[23] Matthieu Martel. Program Transformation for Numerical Precision. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 101–110, 2009.

[24] Timothy Prickett Morgan. Nvidia Tweaks Pascal GPU for Deep Learning Push, 2015. Available at http://www.nextplatform.com/2015/03/18/nvidia-tweaks-pascal-gpus-for-deep-learning-push.

[25] Mariano Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *Proceedings of the International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, pages 213–229, 2017.

[26] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically Improving Accuracy for Floating-Point Expressions. In *Proceedings of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, 2015.

[27] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H Bailey, and David Hough. Floating-Point Precision Tuning Using Blame Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1074–1085, 2016.

[28] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning Assistant for Floating-Point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 27:1–27:12, 2013.

[29] Philipp Rümmer and Thomas Wahl. An SMT-LIB Theory of Binary Floating-Point Arithmetic. In *Informal Proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[30] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Optimization of Floating-Point Programs with Tunable Precision. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 53–64, 2014.

[31] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *Proceedings of the 20th International Symposium on Formal Methods (FM)*, pages 532–550, 2015.

[32] Enyi Tang, Earl Barr, Xuandong Li, and Zhendong Su. Perturbing Numerical Calculations for Statistical Analysis of Floating-Point Program (In)stability. In *Proceedings of the 8th International Symposium on Software Testing and Analysis (ISSTA)*, pages 131–142, 2010.