Rigorous Floating-Point Mixed-Precision Tuning

Wei-Fan Chiang Mark Baranowski Ian Briggs
Alexey Solovyev Ganesh Gopalakrishnan Zvonimir Rakamarić
School of Computing, University of Utah, Salt Lake City, Utah, USA
{wfchiang,baranows,ibriggs,monad,ganesh,zvonimir}@cs.utah.edu

Abstract

Virtually all real-valued computations are carried out using floating-point data types and operations. The precision of these data types must be set with the goals of reducing the overall round-off error, but also emphasizing performance improvements. Often, a mixed-precision allocation achieves this optimum; unfortunately, there are no techniques available to compute such allocations and conservatively meet a given error target across all program inputs. In this work, we present a rigorous approach to precision allocation based on formal analysis via Symbolic Taylor Expansions, and error analysis based on interval functions. This approach is implemented in an automated tool called FPTUNER that generates and solves a quadratically constrained quadratic program to obtain a precision-annotated version of the given expression. FPTUNER automatically introduces all the requisite precision up and down casting operations. It also allows users to flexibly control precision allocation using constraints to cap the number of high precision operators as well as group operators to allocate the same precision to facilitate vectorization. We evaluate FPTUNER by tuning several benchmarks and measuring the proportion of lower precision operators allocated as we increase the error threshold. We also measure the reduction in energy consumption resulting from executing mixed-precision tuned code on a real hardware platform. We observe significant energy savings in response to mixed-precision tuning, but also observe situations where unexpected compiler behaviors thwart intended optimizations.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.3.4 [*Programming Languages*]: Processors; G.1.0 [*Numerical Analysis*]: General

Keywords Floating-point arithmetic, Program optimization, Rigorous compilation, Precision allocation, Energy-efficient computing

1. Introduction

The use of floating-point arithmetic to carry out real arithmetic on computers is almost standard practice, and finds applications in a wide variety of computational problem domains such as weather simulation, numerical linear algebra, and embedded computing. The standard IEEE floating-point [32] precision choices for computation are single-precision (32 bits) and double-precision (64 bits), with quad-precision (128 bits) occasionally used, and half-precision (16 bits) becoming available on certain platforms (e.g., ARM NEON [2], Nvidia Pascal GPU [43]). These precision choices primarily decide the highest real number magnitude representable as a floating-point number. They also decide the extent of round-off error incurred: the higher the precision, the lesser the round-off error. Unfortunately, code performance goes down dramatically if high precision is used everywhere: it can double cache occupancy and more than double energy costs when going from single- to double-precision, for example. The natural question therefore is whether one can obtain the best error/energy trade-off by allocating low precision almost everywhere, with high precision used selectively — hereafter called *mixed-precision tuning*.

Challenges of Mixed-Precision Tuning. Past research [3, 10] has shown that allocating only a few of the operators of a given expression at higher precision and the rest at lower precision (i.e., a mixed-precision implementation) often suffices to keep round-off error below a given threshold, while also improving performance. For example, for a simple expression such as $(a - b)/(c + d)$, it may be best to carry out division in single-precision and the rest of the operations in double-precision.

Unfortunately, round-off errors are highly non-intuitive both in manifestation as well as accumulation. Floating-point values are bunched up tightly closer to 0, but are astronomically far apart when approaching the largest representable value; and round-off error is proportional to this spacing (known as units in the last place or *ULP*). For a given precision, the absolute error introduced by a calculation is proportional to the magnitude of the computed result. In practical terms, this means that given a complex expression of a few variables comprised of linear, non-linear, and transcendental operations, it is impossible to tell which inputs give rise to high (intermediate) values within the expression tree. Each operation application introduces further round-off error with respect to the calculated result. This error is at best half an ULP of the computed result for some operators, and higher for others. To make matters worse, floating-point error analysis is highly non-compositional and does not obey familiar algebraic laws such as associativity. For example, Kahan has observed that $(e^z - 1)/z$ sometimes exhibits higher round-off error in certain input ranges as compared to $(e^z - 1)/log(e^z)$, even though the error in z is obviously lower than its (real-equivalent) expression $log(e^z)$ [35].[1] Given these challenges, *rigorous* mixed-precision tuning must be driven by rigorous *global* (whole-expression) round-off error analysis over the entire input domain. Our main contribution is such an approach, which we embodied in our new tool FPTUNER that is open-source and freely available.[2]

[1] We have confirmed this behavior using FPTaylor [52].

[2] `https://github.com/soarlab/FPTuner`

Challenges of Rigorous Round-Off Error Estimation. If the underlying error analysis engine of a mixed-precision tuner produces bloated error estimates, it will unnecessarily force the use of more higher precision operations. In this work, we base error analysis on our recently introduced rigorous method called *Symbolic Taylor Expansions* [52]. This method accurately computes round-off error for entire input interval combinations. We have shown that this approach produces far tighter as well as rigorously provable error bounds compared to contemporary methods. It is also the only available method that can handle transcendental functions. This helps FPTUNER produce parsimonious as well as rigorous precision allocations across a wide variety of expressions.

Primary Related Work. The Rosa compiler for reals [18] is the only prior line of work that implements precision allocation driven by rigorous error analysis. However, Rosa does not support mixed-precision tuning.

When it comes to non-rigorous methods, several whole-program mixed-precision tuning approaches have been proposed. One of the earliest efforts in this area is by Buttari et al. [11], with many more vastly refined approaches appearing since then [37, 47, 48]. These methods are based on a simple sampling of inputs, meaning that they are driven by a few (typically a few dozen) program inputs supplied by the user. Then, they employ various heuristics to consider a small subset of the total (exponential) number of different operator precision allocations, trying to reduce the overall number of high precision operators. For each given input and candidate precision selection, the whole program must be executed. The search stops when none of the inputs violate the chosen error threshold.

Given the enormity of the total number of inputs available within typical input-intervals of interest, and the sensitivity of floating-point error to actual data inputs, no guarantees are produced or implied by these tools (for the remaining inputs). For example, after one of these tools tunes a piece of code using inputs $x_1, \ldots, x_n$ that span an interval X, it is still possible to very easily locate (e.g., through randomized search) a new input $x_{n+1} \in X$ that violates the stipulated error threshold. This makes these tools inapplicable for tuning critical pieces of software (e.g., code residing in arithmetic libraries) where rigorous guarantees are expected to be published. Nonetheless, these tools can still be quite useful in practice. As an example, it has been shown that the code obtained after tuning may often yield acceptable numerical convergence rates within iterative solvers [11]. Unlike these sampling-based approaches, our work provides rigorous guarantees across *entire* input intervals.

Versatile Handling of Precision Selection and Type-Casts. During the process of mixed-precision tuning, one must incorporate precision up/down casting operations (e.g., to convert from double-precision to single-precision). These casting operations are like additional operators in that they detract from performance and produce extra round-off errors. This introduces a (meta-)circular dependency in the tuning process. More specifically, precision tuning is driven by error analysis, and each allocation generated by tuning results in the introduction of casting operations that, in turn, can affect error analysis. Our work naturally incorporates such dependencies into its *quadratically constrained quadratic programming* (QCQP [46]) formulation of the tuning problem.

The QCQP approach also provides added versatility. For instance, one can easily incorporate additional QCQP constraints to group a collection of operators to share the same precision, limit the total number of higher precision choices, or limit the total number of type-casting operations. None of the prior efforts in mixed-precision tuning provide such "knobs" to help improve the resulting code quality. As an example, grouping can encourage compiler backends to generate vector instructions, thus potentially providing additional performance benefits.

Careful Empirical Evaluation of Precision Tuning. One may be tempted to jump to the conclusion (as we initially did) that any precision allocation that maximizes the number of low precision operators while meeting the error thresholds is the best. Unfortunately, not all reduced-precision allocations are necessarily improvements in terms of overall computation time and energy, for two main reasons: (1) the cost of the intrinsic instructions involved in type-casting (e.g., [33]), and (2) the highly unpredictable effect that specific precision selections have on compiler optimizations. As we observed, some precision allocations in fact make the compiler suddenly choose software fallback paths or introduce unnecessary loads and stores, thus detracting from performance.

In our benchmarks, we are careful to check that the right compiler optimization levels are chosen to ensure that assembly code truly carries the mixed-precision intent.[3] We also conduct actual energy measurements using a test hardware platform equipped with an accurate wattmeter, revealing how energy consumption and execution vary with precision allocation for two popular compiler choices. These results may help inform work in approximate computing that may take the route of tuning floating-point precision in order to gain performance advantages. Such opportunities seem to be on the rise, as evidenced by the availability of three hardware-supported precision choices — single-, double-, and half-precision in a recently released GPU [43]. While the exponentiality of the overall precision allocation problem may limit practical applicability without further research, our results suggest that designers of library routines can begin to meaningfully employ FPTUNER without facing undue turn-around times while tuning.

Contributions. To recap, our contributions are the following:

- We present a rigorous as well as automated approach for precision tuning of floating-point expressions. Our approach also provides a versatile set of knobs to guide precision selection.
- We implement our approach in a prototype tool called FPTUNER, and apply it on many realistic benchmarks. We released both the tool and the benchmarks.
- We report the results of precision tuning with FPTUNER in terms of performance and energy measurements on actual hardware, and point out delicate interactions between mixed-precision type declarations and compiler actions.

Roadmap. We first provide a reasonably self-contained overview of our work, aided by examples (§2 and §3). We then describe our methodology for generating optimization instances for tuning (§4), and provide the implementation details of FPTUNER (§5). We present our empirical evaluation in §6, and discuss the limitations of our work in §7. In §8 we summarize related work. We end the paper with conclusions and some future directions for research (§9).

2. Preliminaries

Notation. Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of natural numbers. For brevity, we sometimes view $n \in \mathbb{N}$ as the set $\{x \in \mathbb{N} : x < n\}$ (as in set theory). Thus, $i \in 4$ means $i \in \{0, 1, 2, 3\}$.

Floating-Point Arithmetic. We briefly recall all necessary properties of floating-point arithmetic. Our text is based on the exposition from previous work [52]. The IEEE 754 standard [32],

[3] FPTUNER outputs a C program containing floating-point type declarations to express its precision choices; in our experiments, we are extremely careful to ensure that the right intrinsic machine instructions are present in assembly code to account for the prescribed precision changes.

Precision (bits)	ϵ	δ
half (16)	2^{-11}	2^{-25}
single (32)	2^{-24}	2^{-150}
double (64)	2^{-53}	2^{-1075}
quad (128)	2^{-113}	2^{-16495}

Table 1: Rounding to nearest operator parameters.

concisely formalized and explained in previous work [24, 25], defines three floating-point formats: *single-precision* or 'single' (32 bits), *double-precision* or 'double' (64 bits), and *quad-precision* or 'quad' (128 bits). Rounding plays a central role in defining the semantics of floating-point arithmetic. Denote the set of floating-point numbers (in some fixed format) as $\mathbb{F}$. A rounding operator $\mathrm{rnd} : \mathbb{R} \rightarrow \mathbb{F}$ is a function which takes a real number and returns a floating-point number which is closest to the input real number and has some special properties defined by the rounding operator. Common rounding operators are rounding to nearest (ties to even), toward zero, and toward $\pm\infty$. A simple model of rounding is given by the rounding rule

$$\mathrm{rnd}(x) = x(1+e) + d, \tag{1}$$

where $x \in \mathbb{R}, |e| \leq \epsilon, |d| \leq \delta$, and $e \times d = 0$ [25]. The parameter ϵ (or *machine epsilon*) specifies the maximal relative error introduced by the given rounding operator. The parameter δ gives the maximal *absolute* error for numbers which are very close to zero (relative error estimation does not work for these small numbers called subnormals). Table 1 shows values of ϵ and δ for the rounding to nearest operator of different floating-point formats. We denote the machine epsilon corresponding to an n-bit floating-point representation with ϵ_n.

Real Expression. Let $\mathscr{E}$ be a real-valued expression comprised of real-valued constants, variables, and operators taking and yielding real-valued quantities. We view $\mathscr{E}$ as denoting a function $\lambda\mathbf{x}.\ \mathscr{E} : \mathbb{R}^N \rightarrow \mathbb{R}$, where an N-dimensional vector $\mathbf{x}$ matches $\mathscr{E}$'s variables. We write $\mathscr{E}(\mathbf{x})$ to denote the application of such a function to an N-dimensional input vector $\mathbf{x}$. We assume that the inputs of $\mathscr{E}$ belong to a bounded domain $\mathbb{I}$, i.e., $\mathbf{x} \in \mathbb{I}$. The domain $\mathbb{I}$ can be quite arbitrary but we only consider products of intervals in our benchmarks because we rely on an external global optimization tool which does not support more complicated domains. In the benchmarks presented later, we have $a_i \leq x_i \leq b_i$ for all $i = 1, \ldots, n$. In this case, $\mathbb{I} = [a_1, b_1] \times \ldots \times [a_n, b_n]$ is a product of intervals (i.e., an n-dimensional rectangle).

Since we will be allocating specific precision values to different operators of an expression, we need to refer unambiguously to specific sub-expressions of $\mathscr{E}$; hence, we assume that $\mathscr{E}$ has a specific syntax-tree and a fixed numbering (e.g., preorder numbering) for its operators. We use $\dot{\mathscr{E}}$ to denote the principal operator of $\mathscr{E}$, and $p(\mathscr{S}, \mathscr{E})$ to denote the preorder number of a sub-expression $\mathscr{S}$ within $\mathscr{E}$. For example, if $\mathscr{E} = x + y, \dot{\mathscr{E}}$ is $+$, $p(x, \mathscr{E}) = 1$, $p(y, \mathscr{E}) = 2$, and $p(x + y, x + y) = 0$. When there is little risk of confusion and $\mathscr{E}$ is clear from the context, we use $\dot{\mathscr{S}}$ (i.e., principal operator of $\mathscr{S}$) in lieu of $p(\mathscr{S}, \mathscr{E})$ (i.e., its preorder number).

Precision Allocation. Let $A^{\mathscr{E}}$ be an allocation of precision to the constituents of an expression $\mathscr{E}$ (simply written A when $\mathscr{E}$ is clear from the context). An allocation is a mapping from $\mathscr{E}$'s nodes (their preorder numbers) to a set of allowed machine epsilon values. For instance, to model expression $x + y$ with x set to 32 bits, y to 64 bits, and the $+$ operator meant to yield 64-bit outputs, we use the allocation function $A = \{(0, \epsilon_{64}), (1, \epsilon_{32}), (2, \epsilon_{64})\}$. We assume that when an operator such as $+$ yields a certain precision (e.g., 64-bit output), its inputs are also set to be at the same precision (also 64-bit inputs). Thus, in our example, a type-casting operation is needed to elevate x to the input type of $+$. Clearly, these type-casting operations are needed based on the actual intended allocation itself. Hence, while exploring the space of allocations to pick an "optimal" one, we must also take into account the type-casting operations being "silently" introduced — and more importantly, the *additional* round-off errors these type-casting operations may introduce.

Floating-Point Expression. Given a real-valued expression $\mathscr{E}$ and an allocation A, let $\mathscr{E}_A$ denote $\mathscr{E}$ under allocation A (i.e., interpreted according to allocation A). Similarly to a real expression, we view $\mathscr{E}_A$ as denoting a function $\lambda\mathbf{x}.\ \mathscr{E}_A : \mathbb{F}^N \rightarrow \mathbb{F}$. For example, under the allocation function $A = \{(0, \epsilon_{64}), (1, \epsilon_{32}), (2, \epsilon_{64})\}$, expression $x + y$ behaves like a function $\lambda x, y.\ (x + y)_A : \mathbb{F}^2 \rightarrow \mathbb{F}$. Here, x is a 'single' and y is a 'double', and this function yields a double. From the user perspective, when two reals are sent in, x gets rounded as per ϵ_{32} (is "rounded more") and y gets rounded as per ϵ_{64} (is "rounded less"), the calculation gets carried out, and the result is expressed as a double. This process is modeled by the *modeling expression* associated with $(x + y)_A$, described next.

Modeling Expression. A *modeling expression* $\tilde{\mathscr{E}}_A$ is a real-valued expression that models the value yielded by the floating-point expression $\mathscr{E}_A$ using multiple applications of the rounding rule from Equation 1 — one such application at every operator node of $\mathscr{E}$. Thus, we view a modeling expression $\tilde{\mathscr{E}}_A$ as denoting a function $\lambda\mathbf{x}, \mathbf{e}.\ \tilde{\mathscr{E}}_A : \mathbb{R}^N \times \mathbb{R}^{|A|} \rightarrow \mathbb{R}$, where $\mathbf{e}$ is an additional $|A|$-dimensional vector. The elements of $\mathbf{e}$ are the e arguments that each application of the rounding rule at each operator site of $\mathscr{E}$ entails. We call these es *noise variables*, in the sense that if they are all 0, we get the value of the original real-valued expression $\mathscr{E}$.

It is useful to keep in mind several facts associated with the noise variables. (1) One can estimate the rounding error of $\mathscr{E}_A$ by computing a Taylor series whose first-order terms are the partial derivatives of $\tilde{\mathscr{E}}_A$ with respect to the noise variables. (2) By assuming that the noise variables are independent, one can obtain a pessimistic error model. However, by equating noise variables (e.g., when $\mathscr{E}_A$ has two identical sub-expressions), one can model errors more precisely. We illustrate this by taking $x-(x+y)$ as our example expression. In this expression, if we equate the noise variables for x, the round-off errors caused by xs cancel, and the result is affected only by the round-off error of y (if any) and that introduced by the $+$ and $-$ operators. (3) As already mentioned, while searching for an "optimal" allocation, we must also take into account the type-casting operations being "silently" introduced. A clean way to present our theory, and obtain a straightforward implementation thereof is to introduce $2 \times |A|$ noise variables. At each operator site, we not only introduce the operator-specific rounding via e, but also a type-specific rounding via t. We now introduce these ideas step by step. For the sake of enhanced readability, in the remainder of this section we continue to present our main ideas without doubling the number of noise variables (we will address this detail in §4).

Formal Relationship between $\mathscr{E}_A$ and $\tilde{\mathscr{E}}_A$. The gist of the concepts discussed so far lies in the formal relationship between $\mathscr{E}_A$ and $\tilde{\mathscr{E}}_A$. For every valid input assignment $\mathbf{x} \in \mathbb{I}$ of $\mathscr{E}$, we can always find a certain assignment $\mathbf{e}$ to the noise variables such that the condition $\mathscr{E}_A(\mathbf{x}_A) = \tilde{\mathscr{E}}_A(\mathbf{x}, \mathbf{e})$ holds (formally proved in previous work [25]). Vector $\mathbf{x}_A$ is the vector of floating-point numbers obtained by rounding $\mathbf{x}$ under the types specified in allocation A. In §4, we present how we generate the modeling expression $\tilde{\mathscr{E}}_A$ from a given expression $\mathscr{E}$ and an allocation A.

Partial Derivative of Modeling Expression. Given a modeling expression $\tilde{\mathscr{E}}_A$, the partial derivative of $\tilde{\mathscr{E}}_A$ with respect to noise variable e_i is denoted as $\mathcal{D}(\tilde{\mathscr{E}}_A, e_i) = \frac{\partial \tilde{\mathscr{E}}_A}{\partial e_i}$. We prefer $\mathcal{D}(\tilde{\mathscr{E}}_A, e_i)$ to $\frac{\partial \tilde{\mathscr{E}}_A}{\partial e_i}$ for higher readability. As we elaborate in §4, we build on Symbolic Taylor Expansions [52] and estimates the first-order error by computing $\mathcal{D}(\tilde{\mathscr{E}}_A, e_i)$ weighted by noise variables.

3. Motivating Example

To motivate our work, let us consider a simple example expression $\mathscr{E} = x - (x + y)$. Let A_{64} be an allocation that assigns double-precision to every operator and variable in $\mathscr{E}$. That is, $A_{64}[i] = \epsilon_{64}$ for $i \in 5$. The modeling expression $\tilde{\mathscr{E}}_{A_{64}}$ is $(x \cdot (1+e_1) - (x \cdot (1+e_3) + y \cdot (1+e_4)) \cdot (1+e_2)) \cdot (1+e_0)$. 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 distinct gives a pessimistic error estimate, as the round-off errors are allowed to be uncorrelated.

Based on Symbolic Taylor Expansions [52], we can now state an inequality bounding the magnitude of the absolute error of $\mathscr{E}$:

$$\left|\tilde{\mathscr{E}}_{A_{64}} - \mathscr{E}\right| \leq \max_{x,y} \left(\sum_{i \in 5} |e_i| \cdot \left| \mathcal{D}(\tilde{\mathscr{E}}_{A_{64}}, e_i)(x, y, \mathbf{0}) \right| \right) + M_2.$$

Here, $\mathbf{0}$ denotes a zero-vector that assigns 0 to the noise variables, and each e_i is bounded by a positive constant $A_{64}[i]$. M_2 is a bound of all higher order error terms. In general, this is a small positive constant (of order ϵ_{64}^2 in our example). For simplicity, assume that $M_2 = 0$ in our example; we describe how FPTUNER deals with higher-order error terms in §5. Next, we replace each first derivative $\mathcal{D}(\tilde{\mathscr{E}}_{A_{64}}, e_i)$ by its upper bound

$$U_{e_i} = \max_{x,y} \left| \mathcal{D}(\tilde{\mathscr{E}}_{A_{64}}, e_i)(x, y, \mathbf{0}) \right|,$$

thereby obtaining the following bound on the absolute error of $\mathscr{E}$:

$$\left|\tilde{\mathscr{E}}_{A_{64}} - \mathscr{E}\right| \leq \sum_{i \in 5} U_{e_i} \cdot A_{64}[i] \tag{2}$$

In our approach, we follow this relaxed upper bound to formulate precision tuning as a quadratically constrained quadratic-programming problem, in which the upper bounds U_{e_i} are treated as constants. The variables employed in our quadratic-programming formulation are the *allocation variables*, which are discussed next.

Mixed-Precision Allocation. Let A_{mixed} be a mixed-precision allocation that assigns single-precision to the subtraction operator and double-precision to all the other operators. Under A_{mixed}, the modeling expression becomes

$$\tilde{\mathscr{E}}_{A_{mixed}} = (x \cdot (1+e_1) \cdot (1+t_1) - (x \cdot (1+e_3) \cdot (1+t_3) + y \cdot (1+e_4) \cdot (1+t_4)) \cdot (1+e_2) \cdot (1+t_2)) \cdot (1+e_0) \cdot (1+t_0) \tag{3}$$

Recall that t variables introduce type-specific rounding, i.e., they capture type-casting. Note that by equating e_1 and e_3, we correlate the operator-specific errors of the two x operators. Furthermore, e_0 is bounded by ϵ_{32}. On the other hand, t_0 depends on the *context* of the whole expression — if the context expects a b-bit precision for $b \geq 32$, then $t_0 = 0$, as there will be no loss of precision when the expression conveys its result to its context. (Here, context refers to any operator that consumes the result of $\mathscr{E}$.) In addition, t_1 and t_2 help cast a 64-bit computation to 32 bits, and thus are bounded by ϵ_{32}; observe that t_3 and t_4 are 0. Finally, e_1, e_3, and e_4 are bounded by ϵ_{64}. Taking these into account, we bound the overall error with

$$\sum_{i \in 5} U_{e_i} \cdot A_{mixed}[i] + \sum_{i \in \{1,2\}} U_{t_i} \cdot \epsilon_{32} + \sum_{i \in \{0,3,4\}} U_{t_i} \cdot 0.$$

As illustrated by this example, the bounds of the type-specific t variables depend on the type-casts occurring between the operators and their operands. In the next section, we precisely describe how our approach relates the t variables' bounds to the given allocation A. Note that, for the two noise variables e and t at the same operator site, they share the identical expression as their first derivatives (regardless of the allocations; this result can be easily seen in the context of the example in Equation 3, and can be rigorously shown on the full expression syntax through structural induction):

$$\mathcal{D}(\tilde{\mathscr{E}}, e) = \mathcal{D}(\tilde{\mathscr{E}}, t). \tag{4}$$

Thus their upper bounds (U_e and U_t) only need to be calculated once.

4. Methodology

As briefly described in the previous section, there are two possible rounding steps at each operator site:

Operator-Specific Rounding. Each operation introduces one rounding step. The magnitude of this rounding error depends on the type of the operator (its specification).

Type-Specific Rounding. The result of an operation may experience a second rounding step before flowing into the parent operator's argument position (e.g., if the parent operator expects a lower precision data-type). We call this a *type-cast*, and it is typically supported in the assembly code through an *intrinsic* instruction.

Therefore, each operator with preorder number i is associated with two noise variables, e_i and t_i, accounting for the aforesaid rounding steps.

We enrich the notion of an allocation A by defining two different allocation maps, namely A_e and A_t, that specify the bounds of these noise variables. More specifically, given the preorder numbering i of an operator, $A_e[i]$ (resp., $A_t[i]$) yields the bound of e_i (resp., t_i). A given allocation A directly maps to A_e, meaning $A_e = A$. In addition, A_t is implied by (or depends on) A_e in a manner we shall precisely define.

Given a real-valued expression $\mathscr{E}$, we generate the modeling expression $\tilde{\mathscr{E}}_{A_e,A_t}$ (simply written $\tilde{\mathscr{E}}$ when A_e and A_t are clear from the context) by introducing two independent noise variables e and t at each operator site. Equation 3 provides an example of such a modeling expression. The bounds of the noise variables are defined as

$$\forall i \in |A| \,.\ |e_i| \leq A_e[i] \ \wedge\ |t_i| \leq A_t[i].$$

In a manner similar to Equation 2, the absolute error of $\mathscr{E}$ for a given allocation is bounded by the following (recall $U_{e_i} = U_{t_i}$):

$$\sum_{i \in |A_e|} U_{e_i} \cdot A_e[i] + \sum_{i \in |A_t|} U_{t_i} \cdot A_t[i]. \tag{5}$$

Generating $\tilde{\mathscr{E}}$ and A_t. Figure 1 gives inference rules that inductively define the relation $\triangleright$ that maps $(\mathscr{E}, C, A_e)$ to $(\tilde{\mathscr{E}}, A_t)$, where

- $\mathscr{E}$ is the given real expression,
- C is $\mathscr{E}$'s context, i.e., the operator that consumes $\mathscr{E}$'s result as an argument,
- A_e is identical to the allocation A,
- $\tilde{\mathscr{E}}$ is the generated modeling expression corresponding to $\mathscr{E}$,
- A_t is the generated type-cast allocation corresponding to A_e.

For example, let $\mathscr{E}$ be a binary expression where $\mathscr{E} = op(\mathscr{E}_0, \mathscr{E}_1)$. Rule BINEXPR (Figure 1) first recursively generates the modeling expressions $\tilde{\mathscr{E}}_0$ and $\tilde{\mathscr{E}}_1$ of the two operands, and then it uses them

$$\textsc{BinExpr}\frac{(\mathscr{E}_0,\dot{\mathscr{E}},A_e)\rhd(\tilde{\mathscr{E}}_0,A_t^0)\qquad(\mathscr{E}_1,\dot{\mathscr{E}},A_e)\rhd(\tilde{\mathscr{E}}_1,A_t^1)\qquad A_t=\{(\dot{\mathscr{E}},(A_e[C]>A_e[\dot{\mathscr{E}}]\to A_e[C],0))\}\cup A_t^0\cup A_t^1}{(op(\mathscr{E}_0,\mathscr{E}_1),C,A_e)\ \rhd\ (op(\tilde{\mathscr{E}}_0,\tilde{\mathscr{E}}_1)\cdot(1+e_{\dot{\mathscr{E}}})\cdot(1+t_{\dot{\mathscr{E}}}),A_t)}$$

$$\textsc{Var}\frac{A_t=\{(\dot{v},(A_e[C]>A_e[\dot{v}]\to A_e[C],0))\}}{(v,C,A_e)\ \rhd\ (v\cdot(1+e_{\dot{v}})\cdot(1+t_{\dot{v}}),A_t)}$$

$$\textsc{Const\_}b\frac{A_t=\{(\dot{c},(A_e[C]>\epsilon_b\to A_e[C],0))\}}{(c,C,A_e)\ \rhd\ (c\cdot(1+e_{\dot{c}})\cdot(1+t_{\dot{c}}),A_t)}$$

Figure 1: Inference rules for generating modeling expression and type-cast allocation.

$$\textsc{BinExpr}\frac{(\mathscr{E}_0,\dot{\mathscr{E}},A_e)\rhd(\tilde{\mathscr{E}}_0,A_t^0)\qquad(\mathscr{E}_1,\dot{\mathscr{E}},A_e)\rhd(\tilde{\mathscr{E}}_1,A_t^1)\qquad A_t=\{(\dot{\mathscr{E}},\ s_{64}^{\dot{\mathscr{E}}}\cdot s_{32}^{C}\cdot A_e[C])\}\cup A_t^0\cup A_t^1}{(op(\mathscr{E}_0,\mathscr{E}_1),C,A_e)\ \rhd\ (op(\tilde{\mathscr{E}}_0,\tilde{\mathscr{E}}_1)\cdot(1+e_{\dot{\mathscr{E}}})\cdot(1+t_{\dot{\mathscr{E}}}),A_t)}$$

$$\textsc{Var}\frac{A_t=\{(\dot{v},\ s_{64}^{\dot{v}}\cdot s_{32}^{C}\cdot A_e[C])\}}{(v,C,A_e)\ \rhd\ (v\cdot(1+e_{\dot{v}})\cdot(1+t_{\dot{v}}),A_t)}$$

$$\textsc{Const\_}b\frac{A_t=\{(\dot{c},\ s_{64}^{\dot{c}}\cdot s_{32}^{C}\cdot A_e[C])\}}{(c,C,A_e)\ \rhd\ (c\cdot(1+e_{\dot{c}})\cdot(1+t_{\dot{c}}),A_t)}$$

Figure 2: Lifted inference rules: the rules of Figure 1 with pseudo-Boolean variables.

to construct the modeling expression $\tilde{\mathscr{E}}$ with the principal operator op (also denoted with $\dot{\mathscr{E}}$) and the two independent noise variables $e_{\dot{\mathscr{E}}}$ and $t_{\dot{\mathscr{E}}}$. The introduced type-cast noise variable $t_{\dot{\mathscr{E}}}$ is bounded by $(A_e[C] > A_e[\dot{\mathscr{E}}] \to A_e[C], 0)$, where "$(a \to b, c)$" is a conditional expression. This conditional is decided by the comparison of machine epsilons of the principal operator $\dot{\mathscr{E}}$ and the context operator C. Hence, a potential type-cast introduces error only when C's bit-width is smaller than $\mathscr{E}$'s. The rule finally derives A_t by uniting allocations A_t^0 and A_t^1 recursively derived from the operands with the one derived for the current operator. Other rules in Figure 1 are analogous to BINEXPR. Note that rule CONST_b is parameterized by the available bit-widths, meaning that one such rule is instantiated for every bit-width b and applied based on the type of a constant.

Generating Optimization Target Expression. Relation $\rhd$ is a device that generates $\tilde{\mathscr{E}}$ and A_t from a given expression $\mathscr{E}$, context C, and specific allocation A_e. The generation of the modeling expression $\tilde{\mathscr{E}}$ itself is quite mechanical, and achieved through a recursive syntactic transformation of $\mathscr{E}$. On the other hand, the main goal of our approach is for an *optimal* A_e to be computed automatically, instead of being given to us. To achieve that, we must first "lift" $\rhd$ so that $A_e[\dot{\mathscr{E}}]$ and $A_t[\dot{\mathscr{E}}]$, instead of mapping to constants such as ϵ_{32} and ϵ_{64}, map to *symbolic expressions* that capture the available allocation choices. The symbolic expressions are constructed using two pseudo-Boolean variables (i.e., integer variables that take on values 0 or 1) per operator.

More specifically, let $s_{32}^{\dot{\mathscr{E}}}$ and $s_{64}^{\dot{\mathscr{E}}}$ be two pseudo-Boolean variables associated with the top-level operator $\dot{\mathscr{E}}$ of $\mathscr{E}$. (We illustrate our method for only two precision regimes; this can be extended to any number of precision regimes through the use of a sufficient number of pseudo-Boolean variables and suitable constraints over them.) We then establish the following:

- We allow for only one pseudo-Boolean variable per operator to be set, meaning $s_{32}^{\dot{\mathscr{E}}} + s_{64}^{\dot{\mathscr{E}}} = 1$ for $s_{32}^{\dot{\mathscr{E}}}, s_{64}^{\dot{\mathscr{E}}} \in \{0, 1\}$.
- For every expression $\mathscr{E}$, let $A_e[\dot{\mathscr{E}}] = s_{32}^{\dot{\mathscr{E}}} \cdot \epsilon_{32} + s_{64}^{\dot{\mathscr{E}}} \cdot \epsilon_{64}$. Such lifted A_e evaluates to either ϵ_{32} or ϵ_{64} once $s_{32}^{\dot{\mathscr{E}}}$ and $s_{64}^{\dot{\mathscr{E}}}$ are assigned.
- For every expression $\mathscr{E}$, let $A_t[\dot{\mathscr{E}}] = s_{64}^{\dot{\mathscr{E}}} \cdot s_{32}^{C} \cdot A_e[C]$, where C is the context of $\mathscr{E}$. This corresponds to the conditional expression $(A_e[C] > A_e[\dot{\mathscr{E}}] \to A_e[C], 0)$ introduced while defining $\rhd$.

Finally, we must update accordingly the rules in Figure 1 by using the lifted type-cast allocation. Figure 2 shows the updated rules. From now on, $\rhd$ is referring to this lifted version.

Generating Expression to be Optimized. We now define another relation, denoted $\blacktriangleright$, that produces a final expression that is optimized to arrive at a precision allocation. More specifically, this procedure results in generating the absolute error bound expression with *independent* noise variables for an input real expression.

Let the "top level" input real expression to compile be $\mathcal{G}$, its context C_g, and A_e an allocation generated as a symbolic expression as described previously. Then, let $\tilde{\mathcal{G}}$ and A_t be generated using the relation $\rhd$, meaning $(\mathcal{G}, C_g, A_e) \rhd (\tilde{\mathcal{G}}, A_t)$. Figure 3 gives inference rules that inductively define the relation $\blacktriangleright$ that maps $(\mathscr{E}, C, A_e, \tilde{\mathcal{G}}, A_t)$ to T, where $\mathscr{E}$ and C are the current real expression and context (similar to $\rhd$), A_e, $\tilde{\mathcal{G}}$, and A_t are the above constants, and T is the generated expression to be optimized. Notice that we need to carry around $\tilde{\mathcal{G}}$, because the first derivatives $\mathcal{D}(\tilde{\mathcal{G}}, e_{\dot{\mathscr{E}}})$ and $\mathcal{D}(\tilde{\mathcal{G}}, t_{\dot{\mathscr{E}}})$ must be taken with respect to the noise variables associated with $\mathscr{E}$, wherever $\mathscr{E}$ may occur within $\mathcal{G}$.

For example, let $\mathscr{E} = op(\mathscr{E}_0, \mathscr{E}_1)$ be a binary sub-expression of $\mathcal{G}$. Similarly to $\rhd$, rule BINEXPR from Figure 3 first recursively generates the expressions to be optimized T_0 and T_1 of the two operands. Then, we refer to the modeling expression $\tilde{\mathcal{G}}$ of the top-level input real expression $\mathcal{G}$ to calculate the first derivatives $\mathcal{D}(\tilde{\mathcal{G}}, e_{\dot{\mathscr{E}}})$ and $\mathcal{D}(\tilde{\mathcal{G}}, t_{\dot{\mathscr{E}}})$ and their upper bounds $U_{e_{\dot{\mathscr{E}}}}$ and $U_{e_{\dot{\mathscr{E}}}}$ over the two noise variables $e_{\dot{\mathscr{E}}}$ and $t_{\dot{\mathscr{E}}}$. (Note that it is always enough to calculate just one of the upper bounds as per Equation 4, i.e., $U_{e_{\dot{\mathscr{E}}}} = U_{t_{\dot{\mathscr{E}}}} = \max_{\mathbf{x}\in\mathbb{I}} \left|\mathcal{D}(\tilde{\mathcal{G}}, t_{\dot{\mathscr{E}}})(\mathbf{x}, \mathbf{0})\right|$.) The final expression to be optimized T of the current operator is the sum of the sub-expressions T_0 and T_1 with the additional terms $U_{e_{\dot{\mathscr{E}}}} \cdot A_e[\dot{\mathscr{E}}]$ and $U_{t_{\dot{\mathscr{E}}}} \cdot A_t[\dot{\mathscr{E}}]$ for the noise variables. Other rules in Figure 3 are analogous to BINEXPR.

$$\textsc{BinExpr}\ \frac{(\mathscr{E}_0,\dot{\mathscr{E}}, A_e, \tilde{\mathcal{G}}, A_t) \blacktriangleright T_0 \quad (\mathscr{E}_1,\dot{\mathscr{E}}, A_e, \tilde{\mathcal{G}}, A_t) \blacktriangleright T_1 \quad U_{e_{\dot{\mathscr{E}}}} = \max_{\mathbf{x}\in\mathbb{I}} \left|\mathcal{D}(\tilde{\mathcal{G}}, e_{\dot{\mathscr{E}}})(\mathbf{x},\mathbf{0})\right| \quad U_{t_{\dot{\mathscr{E}}}} = \max_{\mathbf{x}\in\mathbb{I}} \left|\mathcal{D}(\tilde{\mathcal{G}}, t_{\dot{\mathscr{E}}})(\mathbf{x},\mathbf{0})\right|}{(op(\mathscr{E}_0,\mathscr{E}_1), C, A_e, \tilde{\mathcal{G}}, A_t) \blacktriangleright (U_{e_{\dot{\mathscr{E}}}} \cdot A_e[\dot{\mathscr{E}}]) + (U_{t_{\dot{\mathscr{E}}}} \cdot A_t[\dot{\mathscr{E}}]) + T_0 + T_1}$$

$$\textsc{Var}\ \frac{U_{e_{\dot{v}}} = \max_{\mathbf{x}\in\mathbb{I}} \left|\mathcal{D}(\tilde{\mathcal{G}}, e_{\dot{v}})(\mathbf{x},\mathbf{0})\right| \quad U_{t_{\dot{v}}} = \max_{\mathbf{x}\in\mathbb{I}} \left|\mathcal{D}(\tilde{\mathcal{G}}, t_{\dot{v}})(\mathbf{x},\mathbf{0})\right|}{(v, C, A_e, \tilde{\mathcal{G}}, A_t) \blacktriangleright (U_{e_{\dot{v}}} \cdot A_e[\dot{v}]) + (U_{t_{\dot{v}}} \cdot A_t[\dot{v}])}$$

$$\textsc{Const\_b}\ \frac{U_{e_{\dot{c}}} = \max_{\mathbf{x}\in\mathbb{I}} \left|\mathcal{D}(\tilde{\mathcal{G}}, e_{\dot{c}})(\mathbf{x},\mathbf{0})\right| \quad U_{t_{\dot{c}}} = \max_{\mathbf{x}\in\mathbb{I}} \left|\mathcal{D}(\tilde{\mathcal{G}}, t_{\dot{c}})(\mathbf{x},\mathbf{0})\right|}{(c, C, A_e, \tilde{\mathcal{G}}, A_t) \blacktriangleright (U_{e_{\dot{c}}} \cdot \epsilon_b) + (U_{t_{\dot{c}}} \cdot A_t[\dot{c}])}$$

Figure 3: Inference rules for generating optimization target expression. Given $\mathcal{G}$ in context C_g to optimize, we first determine $A_e[\dot{\mathscr{E}}]$ (defined as $s^{\dot{\mathscr{E}}}_{32} \cdot \epsilon_{32} + s^{\dot{\mathscr{E}}}_{64} \cdot \epsilon_{64}$) and then obtain $(\tilde{\mathcal{G}}, A_t)$ via $(\mathcal{G}, C_g, A_e) \rhd (\tilde{\mathcal{G}}, A_t)$. Thus, the definition of $\blacktriangleright$ is for the given $\mathcal{G}$ and C_g.

Optimization Problem. The goal of our approach is to compute an optimal precision allocation resulting in the highest benefits (e.g., performance or energy efficiency) for a floating-point computation, while at the same time satisfying our prescribed error criterion. Given a real expression $\mathscr{E}$, allocation A, and user-specified error threshold E, we must satisfy $T \leq E$ where $(\mathscr{E}, C_g, A_e) \rhd (\tilde{\mathscr{E}}, A_t)$ and $(\mathscr{E}, C_g, A_e, \tilde{\mathscr{E}}, A_t) \blacktriangleright T$.

In this setting, we can effect additional desired optimizations. For instance, one can maximize the weighted sum of the number of 32-bit operators allocated, for suitably selected weights $w_{\dot{\mathscr{E}}}$:

$$\text{Satisfy } T \leq E \text{ while maximizing } \sum_{\dot{\mathscr{E}}} w_{\dot{\mathscr{E}}} \cdot s^{\dot{\mathscr{E}}}_{32}. \qquad (6)$$

Doing so will drive the allocation toward mostly 32-bit allocations. Currently, we set the weights of regular operations to 1 and of certain expensive operations to 16 (see §6.1). In our implementation, we use the Gurobi mathematical programming solver [28] to solve the aforementioned optimization problem. We now discuss many other practically important allocation control mechanisms, all of which can be easily realized in our framework through quadratic programming.

4.1 Additional Constraints for Fine-Grained Control

Limiting Type-Casts. Each precision regime crossing (e.g., 64 to 32 bits or vice versa) involves introducing an intrinsic type-cast instruction, which incurs a performance overhead. Our formulation of the quadratic programming problem gives us the ability to symbolically count the number of such precision casts generated when the value of an expression $\dot{\mathscr{E}}$ flows into a context C. For example, the following captures such a count:

$$l(\dot{\mathscr{E}}) = \underbrace{s^{\dot{\mathscr{E}}}_{64} \cdot s^{C}_{32}}_{\text{high-to-low casting}} + \underbrace{s^{\dot{\mathscr{E}}}_{32} \cdot s^{C}_{64}}_{\text{low-to-high casting}}.$$

In addition, we can suitably weigh such expressions or their components for fine-grained control. Note that even though the low-to-high type-cast does not incur round-off error, it can still detract from performance. Finally, given the maximum allowed number of type-casts L, we constrain the sum of $l(\dot{\mathscr{E}})$ across all subexpressions $\mathscr{E}$ with L to control the total number of type-casts in the generated precision allocation.

Ganging of Operators. In certain situations, we may want to assign the same bit-width to a group of operators. For example, when multiple additions are to be executed, one may be able to generate a SIMD instruction by assigning all these operations the same bit-width. To force such an allocation, we allow for "ganging" of operators together to make them share the same precision allocation. Given two operators op_a and op_b, we achieve such ganging using the following constraint:

$$s^{op_a}_{32} = s^{op_b}_{32} \text{ and } s^{op_a}_{64} = s^{op_b}_{64}.$$

5. Implementation

Figure 4 presents an overview of FPTuner, the prototype tool that implements our methodology. FPTuner takes a real expression $\mathscr{E}$ to be optimized as input. Using relation $\rhd$ it first generates the modeling expression $\tilde{\mathscr{E}}$ under a lifted allocation A, and using relation $\blacktriangleright$ the error bound expression T by introducing only one variable c at each operator site; we describe this optimization momentarily. Then, the calculation of the bounds of the first derivatives is performed using Gelpia, which is a scalable and performant global optimizer that we implemented based on previous work [1]. (We describe Gelpia in more detail later in this section.) For each provided error threshold $E_0 \ldots E_M$, FPTuner searches for optimal allocations using the Gurobi [28] mathematical programming solver. As the tool flow illustrates, both of these steps are embarrassingly parallel, which we plan to leverage in future releases of FPTuner. Finally, as described in the end of this section, the generated precision allocations $A_0 \ldots A_M$ are *certified* using the FPTaylor rigorous error estimator to ensure that higher-order errors have no effect on our results. In the case where an allocation would violate the error threshold due to higher-order errors, we could refine the error expression T by adding a small constant to bound the higher-order error, and then re-run FPTuner.

Reducing Noise Variables. Relation $\rhd$ generates a modeling expression by introducing two independent noise variables, e and t, at each operator site, and $\blacktriangleright$ generates one term for each independent variable in the absolute error bound expression. However, recall from Equation 4 that the first partial derivatives with respect to these noise variables are the same. Hence, in Equation 5, we can factor out U_{e_i}, which is equal to U_{t_i}, and weight it by $A_e[i] + A_t[i]$. This allows us to introduce only a single noise variable c whose bound is obtained by adding the absolute values of the bounds of e and t. We modify rule BinExpr of relation $\rhd$ as follows

$$\frac{\begin{array}{c}(\mathscr{E}_0,\dot{\mathscr{E}}, A) \rhd (\tilde{\mathscr{E}}_0, A^0_c),\\ (\mathscr{E}_1,\dot{\mathscr{E}}, A) \rhd (\tilde{\mathscr{E}}_1, A^1_c),\\ A_c = \{(\dot{\mathscr{E}},\ A[\dot{\mathscr{E}}] + s^{\dot{\mathscr{E}}}_{64} \cdot s^{C}_{32} \cdot A[C])\} \cup A^0_c \cup A^1_c\end{array}}{(op(\mathscr{E}_0,\mathscr{E}_1), C, A) \ \rhd\ (op(\tilde{\mathscr{E}}_0,\tilde{\mathscr{E}}_1) \cdot (1 + c_{\dot{\mathscr{E}}}), A_c)}$$

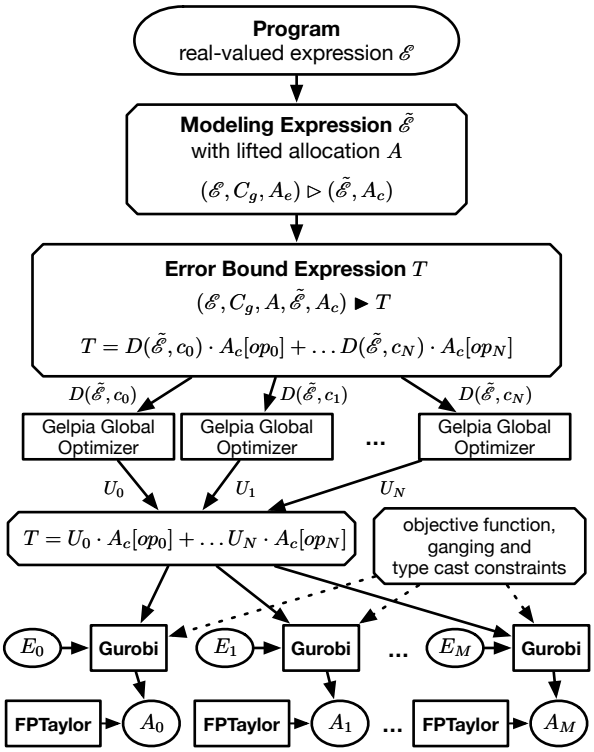


Figure 4: FPTUNER tool flow.

and of relation $\blacktriangleright$ as follows

$$\frac{(\mathscr{E}_0, \dot{\mathscr{E}}, A, \tilde{\mathcal{G}}, A_c) \blacktriangleright T_0 \qquad (\mathscr{E}_1, \dot{\mathscr{E}}, A, \tilde{\mathcal{G}}, A_c) \blacktriangleright T_1 \qquad U_{c_{\dot{\mathscr{E}}}} = \max_{\mathbf{x} \in \mathbb{I}} \left| \mathcal{D}(\tilde{\mathcal{G}}, c_{\dot{\mathscr{E}}})(\mathbf{x}, \mathbf{0}) \right|}{(op(\mathscr{E}_0, \mathscr{E}_1), C, A, \tilde{\mathcal{G}}, A_c) \blacktriangleright (U_{c_{\dot{\mathscr{E}}}} \cdot A_c[\dot{\mathscr{E}}]) + T_0 + T_1}$$

Other rules are modified analogously.

Global Optimization. We have developed a global optimization tool called GELPIA[4] to obtain the upper-bounds U_{ei} from Equation 5. In general, finding the maximum value of an n-variable function requires search over the n-dimensional space of its inputs (i.e., an n-dimensional rectangle). Given the very large number of floating-point n-tuples in this rectangle, exhaustive search is impossible, and sampling can cover only a vanishingly small fraction of possible tuples. In general, approaches for precision estimation and optimization leverage many tools and techniques, including dReal [22], semi-definite programming [41], SMT [18], and classical tools for interval and affine arithmetic [17, 18, 21]. Previous studies [38, 45, 52] have shown that using optimization tools in this arena is promising, often proving to be superior to more classical (e.g., SMT-based) methods that do not support important classes of functions (e.g., transcendental functions). When the interval functions in question are *monotonic* (for rectangle r_1 contained in rectangle r_2, i.e., $r_1 \sqsubseteq r_2$, the upper-bound calculation respects $f(r_1) \sqsubseteq f(r_2)$), one can perform this search using a combination of heuristics. The basic heuristics are to split a rectangle along the longest dimension, obtain upper-bounds for each sub-rectangle, and zoom into the most promising sub-rectangle, while also keeping alive a population of postponed rectangles [1].

[4] `https://github.com/soarlab/gelpia`

GELPIA implements this basic algorithm with a number of improvements. Its performance and the quality of results exceed that of SMT and other classical tools, comparing favorably or exceeding the other tools.

GELPIA is a *rigorous* global optimizer — it guarantees that the returned upper bound is greater than or equal to the global maximum, and the returned lower bound is less than or equal to the global maximum. Key to its efficiency is its use of GAOL [23], an interval library which uses X86 SIMD instructions to speed up interval arithmetic, and also supports transcendental functions such as $\sin$, $\cos$, $\tan$. GAOL is sound as it satisfies the *inclusion* property for interval arithmetic. For example, if $[a, b]+[c, d] = [a+c, b+d]$, where the addition is computed using real arithmetic, GAOL computes the interval as $[a, b] \bigoplus [c, d] = [\underline{a + c}, \overline{b + d}]$, where $\underline{a + c}$ is the nearest double rounded toward $-\infty$ and $\overline{b + d}$ is the nearest double rounded toward ∞. This guarantees the interval inclusion property as $[a, b] + [c, d] = [a + c, b + d] \subseteq [\underline{a + c}, \overline{b + d}] = [a, b] \bigoplus [c, d]$. Since we are operating on real intervals, we employ rewriting to improve results. For example, if $x = [-1, 1]$ then $x \cdot x$ equals $[-1, 1]$ in interval arithmetic; we replace the input subexpression with x^2 which evaluates to $[0, 1]$. We are also able to determine statically if a division-by-zero error occurs, and emit an appropriate message to the user. We implemented GELPIA using the Rust programming language, and we parallelized the search algorithm. We use an *update* thread that periodically synchronizes all solvers to focus their attention on the current most promising sub-rectangle. Additionally, information from other solvers is used to boost the priorities of promising sub-rectangles.

Higher-Order Errors. The method described in §4 computes bounds of the first-order error terms only. As pointed out in previous work [52], the higher-order terms produce only a very minor effect on the error represented by the series.[5]

In our work, we do not compute higher-order errors (which include the round-off effects of subnormal numbers) directly. We estimate the total bound of all higher-order error terms with an external tool (FPTaylor [52]). Then we use the FPTaylor-computed bound in our precision tuning method in order to obtain rigorous results. This can be done in one of the following two ways (we choose the second approach):

1. The first approach is to first estimate a pessimistic higher-order error bound by employing FPTaylor, and feeding this error into our optimization step. The pessimistic estimation is accomplished by considering an expression where all rounding operations are replaced with double rounding (once for e then for t) operations with the lowest precision. We simply add this higher-order error bound to the optimization target expression (which is expression T generated by relation $\blacktriangleright$ described in §4) and solve the corresponding optimization problem. The disadvantage of this method is that the pessimistic higher-order error bound may exclude some allocations that are in fact valid for a given error threshold.
2. A better approach is to start with a higher-order error set to 0, obtain an allocation, and then verify it using FPTaylor to ensure that the stipulated error bound is met even with higher-order errors. If so, we accept the tuned result. If this verification step fails, we iterate our tuning procedure while gradually increasing the higher-order error bound until the verification succeeds.

In all our experiments, the precision allocations generated by FPTUNER immediately passed the verification step even when the higher-order error bound was set to 0.

[5] If it were to have a major effect, it typically would mean that the program already suffers from potential *instability*, which is not the focus of our work.

6. Empirical Evaluation

In this section, we present an evaluation of FPTUNER on a collection of benchmarks, reporting several classes of results.

6.1 Benchmarks and Experimental Setup

Given that most of our benchmarks stem from the Rosa compiler for reals work [18], and that they have performed the only prior rigorous precision analysis (albeit all-quad or all-double), we find it interesting to report what our mixed-precision allocation achieves with respect to pushing more operations toward double-precision, away from quad-precision. This evaluation is reported in Table 2 in terms of the number of operations converted to double-precision, with Table 3 reporting the resulting execution time savings, and Figure 6 showing a scatter plot of these savings.

Given that FPTUNER is unique in being able to incorporate rigorous error analysis for non-linear and transcendental operations, we also include three additional benchmarks into our empirical evaluation.

- The Gaussian distribution benchmark is realized by the following computation:

$$\frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

  where x, μ, and σ are input variables. We prioritize assigning low precision to the expensive exponentiation operation by increasing its weight in Equation 6. This illustrates an important fine-tuning "knob" of FPTUNER that allows for a programmer to influence precision of expensive operations.

- The Maxwell-Boltzmann distribution benchmark is realized by the following computation:

$$\sqrt{\frac{2}{\pi}} \frac{x^2 e^{-x^2/(2a^2)}}{a^3},$$

  where a and x are input variables. Similar to the Gaussian distribution benchmark, we prioritize assigning low precision to the expensive square root and exponentiation operations.

- The cone surface area benchmark is realized by the following computation:

$$\pi \cdot r \cdot \left(r + \sqrt{h^2 + r^2}\right),$$

  where r and h are input variables. Again, we prioritize assigning low precision to the expensive square root operation by increasing its weight in Equation 6.

Finally, we stress-test FPTUNER on a balanced reduction benchmark with 1,023 operators. Figure 5 presents the pseudocode of this benchmark. This example also gives us the opportunity to demonstrate the ganging strategy (see §4.1). The benchmark calculates the sum of 512 elements of input array $data$ using 9 reduction tree levels over which the outer loop iterates. We specify 10 gangs of operators for this benchmark. Our first gang, denoted with $Type_{data}$, groups together all elements of array $data$ into one precision class; we assign this class as single-precision to help minimize the cost of data movement. Next, we chose to gang all operations at the same reduction tree level, resulting in 9 additional gangs denoted with $Type_L$ where $L \in 9$. This allows for FPTUNER to vary precision in a reduction-level-sensitive manner. While prior work (e.g., [10]) has hinted at reduction-level-sensitive precision variation, it has never been achieved automatically and in a rigorous manner. Being able to fine-tune these choices allows us to achieve that, and illustrates the versatility of our approach.

We detail our performance case studies in §6.2. We measure performance on a single Intel Xeon E5-2450 2.10GHz core with 1GB of RAM. For the Rosa benchmarks, we select the same input ranges as the authors of Rosa; for others, we manually select realistic input ranges on our own. (The FPTUNER repository contains source code and input ranges of all our benchmarks.) Note that quad-precision is implemented in software, while double-precision is implemented in hardware.

```
procedure BALANCEDREDUCTION (Type_data data[512])
    for L from 1 to 9 do
        for i from 0 to 2^(9−L) − 1 do
            data[i] = (Type_L)data[i]+(Type_L)data[i+2^(9−L)]
        end for
    end for
end procedure
```

Figure 5: Balanced reduction benchmark.

We detail our energy consumption case studies for selected benchmarks in §6.3, where we measure the net energy savings due to FPTUNER pushing double-precision operations to single-precision. Such an energy study is equitable, given that both these precisions are supported in hardware. The use of built-in performance counters is often not reliable [9], especially at the scale of our examples. Thus, for experimental convenience, we employ an Nvidia Jetson TK1 board with 64-bit ARM quad-core Cortex-A15 CPU at 2.32GHz and 2GB DDR3L RAM at 933MHz, as we were provided with a convenient current/voltage measurement set-up, including scripts to obtain wattage reliably. We took many additional precautions to achieve reliable measurements: (1) we switched the cooling fan to an independent power source, (2) we disabled frequency/voltage scaling, (3) we calibrated our set-up before each experiment with a known resistive load, and (4) we performed FFT-based de-noising of measured results. Finally, we also investigate the effect that two different compilers have on mixed-precision tuning suggestions generated by FPTUNER.

6.2 Performance Case Studies

In these case studies, we explore how different mixed-precision allocations affect performance, where the allocations are automatically obtained with FPTUNER under different error thresholds. We also assess performance benefits of using mixed-precision as opposed to only homogeneous allocations (e.g., all-double or all-quad), which is the regime supported by previous efforts such as the Rosa compiler for reals [18]. Rosa benchmarks are tuned by the Rosa tool by selecting between all-double and all-quad precision (i.e., no mixed-precision). It first rigorously computes the worst-case round-off error for a double-precision benchmark version; then, if the desired error threshold is lower than the worst-case error, Rosa switches to an all-quad version. Our approach allows us to explore the trade-offs in-between these two extremes. We describe our evaluation methodology next.

Tables 2 and 3 give the detailed results of our performance case studies. We first create an all-double implementation of each benchmark, and measure its round-off error using FPTaylor [52], a state-of-the-art rigorous floating-point error estimator. In Table 2, we show these errors in column FPTaylor. Recall that FPTUNER employs a modified version of Symbolic Taylor Expansions that introduces independent noise variables, which in general makes it more conservative than FPTaylor. Hence, column FPTUNER gives errors computed by FPTUNER for comparison; while these errors are slightly higher, they are not excessively pessimistic, and are hence appropriate to be used for precision tuning.

We choose our default error threshold E to be the lowest round-off error value above the FPTUNER-computed error at which an all-double allocation occurs. Hence, when tuning with E as the er-

Benchmark	Estimated round-off error (for all-double allocation)		E	Total ops #	Double ops # per fraction of E	
	FPTaylor	FPTuner			$0.2E$	$0.1E$
verhulst	3.52e-16	3.79e-16	5e-16	5	1	0
sineOrder3	9.97e-16	1.17e-15	5e-15	6	4	2
predPrey	1.89e-16	1.99e-16	5e-16	7	3	2
coneArea	5.75e-13	5.75e-13	1e-12	9	5	2
sine	6.75e-16	8.73e-16	1e-15	11	5	4
doppler1	1.48e-13	1.82e-13	5e-13	11	8	6
doppler2	2.60e-13	3.20e-13	5e-13	11	6	4
doppler3	7.16e-14	1.02e-13	5e-13	11	10	7
rigidBody1	3.86e-13	3.86e-13	5e-13	11	5	3
sqroot	7.12e-16	7.45e-16	1e-15	12	8	5
maxBolt	1.94e-15	5.31e-15	1e-14	12	8	7
rigidBody2	5.23e-11	5.23e-11	1e-10	13	7	6
turbine2	3.13e-14	4.13e-14	5e-14	13	4	2
gaussian	4.79e-16	5.67e-16	1e-15	14	5	1
carbonGas	1.22e-08	1.51e-08	5e-08	15	11	9
turbine1	2.32e-14	3.16e-14	5e-14	16	6	4
turbine3	1.70e-14	1.73e-14	5e-14	16	9	6
jetEngine	1.49e-11	2.68e-11	5e-11	28	18	12
reduction	5.40e-13	5.40e-13	1e-12	1,023	960	768

Table 2: Tuning results preparatory to our performance case studies. Columns FPTaylor and FPTUNER present the worst-case round-off errors computed using FPTaylor and FPTUNER, respectively. Column E provides the default error threshold chosen as the lowest round-off error value above the FPTUNER-computed error at which an all-double allocation occurs. Columns "*Double ops # per fraction of E*" give the number of double-precision operations generated by FPTUNER for error thresholds $0.2E$ and $0.1E$.

ror threshold, FPTUNER automatically synthesizes the all-double benchmark version; in addition, we also synthesize all-quad versions for comparison. We chose additional error thresholds as fractions $0.1E$ and $0.2E$ of the default threshold E, such that synthesis of mixed-precision versions is encouraged. As columns "*Double ops # per fraction of E*" show, the smaller these thresholds are, the fewer number of double-precision operations gets synthesized; in other words, higher numbers of quad-precision operations and the associated precision casting operations are automatically generated. Note that FPTUNER synthesizes only one all-quad precision allocation, which is in the case of *verhulst* under $0.1E$ error threshold. On the other hand, when tuning under $0.2E$ and $0.1E$, we observe that Rosa would always revert to all-quad versions, which is overly pessimistic in most cases as our results show.

Our tool flow automatically generates straight-line C implementations of mixed-precision allocations. Once we arrive at a C implementation for a benchmark, we compile each performance case study benchmark using g++ with `-O0` and `-lquadmath` compiler flags. We use `-lquadmath` to enable support for quad-precision using the GCC Quad-Precision Math Library [40]. To assess performance implications of various precision allocations, we measure execution time of all precision versions. Each experiment consists of executing a precision allocation instance 10^7 times in a loop; we run 100 of such experiments for each precision allocation instance. We average the measured execution times across such 100 runs of an experiment, and we obtain our reported execution times by dividing this average by 10^7. The relative standard deviations of all the measured execution times are less than 1.5%.

In Table 3, columns "*Execution time*" give execution times in nanoseconds of all benchmarks under generated precision allocations. Column *all-double* gives execution times for all-double-precision allocations, while column *all-quad* gives execution times

Benchmark	Execution time (ns)			
	all-double	$0.2E$	$0.1E$	all-quad
verhulst	33.0	175.6	240.2	240.2
sineOrder3	19.4	81.2	144.1	143.9
predPrey	33.3	214.9	202.3	286.5
coneArea	10.4	121.5	228.9	581.7
sine	30.9	562.8	582.9	745.2
doppler1	68.6	261.2	391.3	409.3
doppler2	67.2	383.5	355.0	402.5
doppler3	66.8	202.6	362.9	400.5
rigidBody1	19.8	159.5	194.9	186.8
sqroot	50.2	168.6	383.6	395.3
maxBolt	44.9	171.1	184.1	1,605.8
rigidBody2	28.7	293.3	328.1	416.7
turbine2	37.5	286.2	308.1	326.5
gaussian	53.1	355.5	435.3	1,669.7
carbonGas	22.8	149.7	406.4	492.7
turbine1	78.8	319.3	369.4	516.6
turbine3	79.4	228.3	352.2	514.8
jetEngine	213.3	1,986.5	2,146.7	2,204.7
reduction	1,602.8	10,196.4	9,533.4	14,076.3

Table 3: Execution times of the all-double, all-quad, and mixed-precision ($0.2E$ and $0.1E$) versions of our benchmarks, as generated by FPTUNER. Columns "*Execution time*" give execution times in nanoseconds of the generated precision allocations.

for all-quad-precision allocations; columns $0.2E$ and $0.1E$ give execution times for mixed-precision allocations. Figure 6 compares mixed-precision against all-quad versions. There are only two benchmarks experiencing a small slowdown in their $0.1E$ mixed-precision versions with respect to their all-quad counterparts; all other benchmarks, including all $0.2E$ versions, experience speedups. For example, for $0.2E$ versions the speedups are in the range of 1.1–9.4, with the average being 2.4. It is clear from these results that in many situations it is beneficial to generate mixed-precision versions. While this may not be surprising given the $20 \sim 60\times$ performance difference between the (hardware-supported) double-precision operations and their (software-implemented) quad-precision counterparts, unlike FPTUNER, no previous work on rigorous precision allocation takes advantage of this.

Controlling Type-Casting. There are several exceptional cases in Table 3 that we analyzed in more detail:

- Benchmarks *sineOrder3* and *rigidBody1* have their $0.1E$ allocations resulting in slightly lower performance than the all-quad versions.
- Benchmarks *predPrey* and *doppler2* have their $0.2E$ allocations resulting in slightly lower performance than the $0.1E$ versions.

We diagnosed these to be due to a large number of type-casting intrinsics introduced in the assembly code for these precision allocations, which happens because in these experiments we enforce no constraints on the number of type-casts that FPTUNER generates. For example, the mixed-precision version of *sineOrder3* under the $0.1E$ error threshold results in 3 type-casts, while the total number of operations is only 6 (as indicated by column "*Total ops #*" in Table 2). Upon controlling the number of type-casts as described in §4.1, we obtain the following results:

- When we limit the number of type-casts to 2, the execution time of the generated mixed-precision is 144.20ns, meaning that it remains about the same.
- When we limit the number of type-casts to 1, mixed-precision degenerates into an all-quad version.

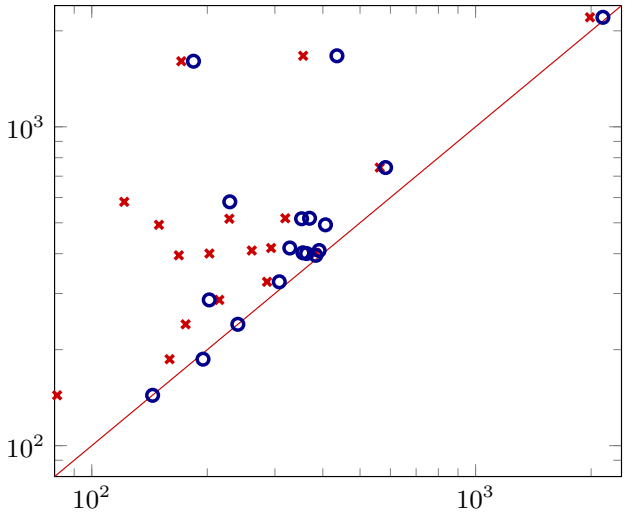


Figure 6: Scatter plot comparing execution times of generated precision allocations from Table 3. The y axis shows execution time of all-quad versions, while the x axis shows execution time of $0.2E$ (red $\times$) and $0.1E$ (blue $\circ$) mixed-precision versions. We omitted *reduction* due to much higher execution times.

Hence, we conclude that an all-quad *sineOrder3* version should be used when $0.1E$ is the error threshold. For *rigidBody1*, the mixed-precision version has 5 type-casts, while the total number of operations is 11. In this case, limiting the number of type-casts to 4 results in the execution time of 228.0ns, limiting to 3 results in 178.0ns, 2 in 212.2ns, while 1 degenerates into an all-quad version. Hence, limiting the number of type-casts to 3 results in the best overall performance. As these experiments indicate, such fine-grained tuning "knobs" are important for reaching optimal performance; to the best of our knowledge, no previous rigorous tuners exhibit such a high degree of control.

6.3 Energy Consumption Case Studies

In these case studies, we explore the effects that mixed-precision tuning has on energy consumption of a program. Given that energy measurements require significant manual set-up effort (detailed later), we identify several more challenging and interesting benchmarks from our performance case studies (Table 2) for our energy consumption experiments. In particular, we choose *sine*, *jetEngine*, and *reduction* due to their long all-quad execution times, and *gaussian*, *maxBolt*, and *coneArea* because they contain at least one expensive operation, such as square root. We restrict ourselves to only single-precision and double-precision because both are supported in hardware, and thus we can measure energy consumption accurately and more directly as described next. Tables 4 and 5 give the detailed results of our energy consumption case studies.

To make our evaluation even more thorough, we employ two different compilers, namely g++ and clang++, to generate two binaries per benchmark (i.e., each benchmark is compiled with both compilers). We also manually check that each generated binary (i.e., assembly code) faithfully follows the prescribed precision assignments. An important precaution in evaluating the benefits of mixed-precision tuning is to measure the energy consumed by the tuned benchmark across multiple inputs, instead of just one arbitrarily chosen input. Hence, for each precision allocation, we measure the average energy consumed across 100 random inputs selected from the input interval, where for each input we execute the tuned benchmark at least 10^7 times. Increasing the number of executions when needed reduces the relative standard deviations of the measured energy consumptions. More specifically, we execute *reduction* 10^7 times, *gaussian* and *maxBolt* 10^8 times, and *coneArea*, *sine*, and *jetEngine* 10^9 times. For all the results shown in Table 5, the relative standard deviations of the energy consumptions are less than 5%.

Benchmark	Estimated round-off error (for all-single allocation)		E	Total ops #	Single ops # per fraction of E	
	FPTaylor	FPTuner			$0.2E$	$0.1E$
coneArea	3.06e-04	3.06e-04	5e-04	9	5	1
sine	3.32e-07	5.14e-07	1e-06	11	6	5
maxBolt	5.30e-06	7.11e-06	1e-05	12	9	8
gaussian	2.78e-07	3.28e-07	5e-07	14	5	1
jetEngine	9.83e-03	1.48e-02	5e-02	28	19	11
reduction	2.90e-04	2.90e-04	5e-04	1,023	896	512

Table 4: Tuning results preparatory to our energy consumption case studies. Columns FPTaylor and FPTuner present the worst-case round-off errors computed using FPTaylor and FPTuner, respectively. Column E provides the default error threshold chosen as the lowest round-off error value above the FPTuner-computed error at which an all-single allocation occurs. Columns "*Single ops # per fraction of E*" give the number of single-precision operations generated by FPTuner for error thresholds $0.2E$ and $0.1E$.

Our energy consumption evaluation methodology is similar to what we employed for our performance measurements. First, Table 4 gives the estimated round-off errors, chosen error thresholds, and numbers of single-precision operations as allocated by FPTuner under the chosen thresholds. Then, Table 5 shows the measured performance and energy consumption results, while Figure 7 plots them. As before, we generate two mixed-precision versions for each benchmark with error thresholds $0.2E$ and $0.1E$. While performing our measurements, we observed that the power usage (energy per unit time) is nearly the same for most precision allocations; however, computations with better allocations have overall reduced execution time, and hence consume less energy, as illustrated by bars "following" lines in the plots. We also note that the binaries generated by different compilers result in similar energy measurements, which is reassuring with respect to portability and the absence of experimental errors. In this respect, *coneArea* is the only outlier, and we provide an explanation for this in the end of this section.

Our results in Table 5 (visualized in Figure 7) show that the mixed-precision versions of benchmarks *sine*, *gaussian*, *maxBolt*, and *coneArea* have moderate performance and energy consumption benefits as we transition from the all-single (column *all-32*) towards all-double (column *all-64*) allocations. For example, when transitioning from the all-double to $0.2E$ mixed-precision versions of these benchmarks, we save on average 23.2% of energy when compiling with g++ and 31.5% when compiling with clang++. This indicates that leveraging mixed-precision is often beneficial even in the context of energy consumption. On the other hand, the mixed-precision versions of *jetEngine* and *reduction* consume more energy than their all-double versions. Explaining this phenomenon required probing into the assembly code, as we describe in what follows.

Compilation Details. In addition to manually deriving C implementations, for the purpose of the energy usage case studies we also have to manually select adequate compiler flags and check for faithful compilation. The compilers we employ are g++4.8 (Linaro 4.8.4 for Ubuntu) and clang++3.5. In our experiments, we apply the following compiler flags: `-std=c++0x`, `-Os`, `-mfpu=neon`, `-march=native`, and `-fno-inline`. Finally, we manually inspect

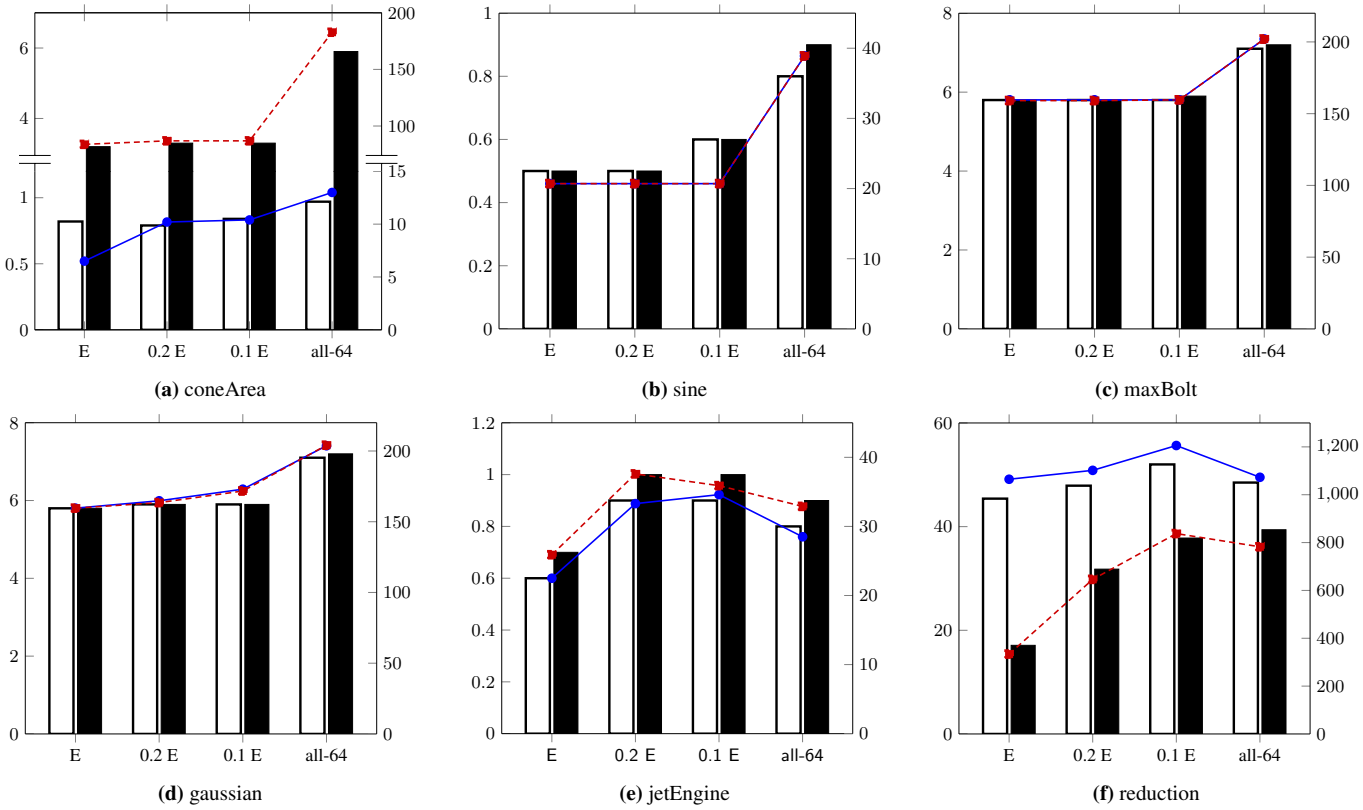
Benchmark	Execution time (ns)				Energy (nJ)			
	all-32	$0.2E$	$0.1E$	all-64	all-32	$0.2E$	$0.1E$	all-64
coneArea	6.5	10.2	10.4	13.0	0.3	0.4	0.4	0.5
sine	20.7	20.7	20.7	38.9	0.5	0.5	0.6	0.8
maxBolt	159.6	159.6	159.6	202.0	5.8	5.8	5.8	7.1
gaussian	159.6	164.8	173.0	203.7	5.8	5.9	5.9	7.1
jetEngine	22.5	33.3	34.6	28.5	0.6	0.9	0.9	0.8
reduction	1,064.7	1,101.9	1,205.7	1,072.9	45.4	47.9	52.0	48.5

(a) Benchmarks compiled with g++

Benchmark	Execution time (ns)				Energy (nJ)			
	all-32	$0.2E$	$0.1E$	all-64	all-32	$0.2E$	$0.1E$	all-64
coneArea	83.8	86.9	86.9	182.8	3.2	3.3	3.3	5.9
sine	20.7	20.7	20.7	38.9	0.5	0.5	0.6	0.9
maxBolt	159.1	159.1	159.6	202.0	5.8	5.8	5.9	7.2
gaussian	159.6	163.5	171.7	204.1	5.8	5.9	5.9	7.2
jetEngine	25.9	37.6	35.9	32.9	0.7	1.0	1.0	0.9
reduction	333.9	646.2	837.6	782.4	17.1	31.8	37.8	39.4

(b) Benchmarks compiled with clang++

Table 5: Execution times and energy consumption of all-single, all-double, and mixed-precision ($0.2E$ and $0.1E$) versions of our benchmarks, as generated by FPTUNER. Columns "*Execution time*" (resp. "*Energy*") report energy consumption in nanojoules (resp. execution times in nanoseconds) of the generated precision allocations. For a billion invocations over a second, each nJ saved is a Watt saved.



(a) coneArea **(b)** sine **(c)** maxBolt

(d) gaussian **(e)** jetEngine **(f)** reduction

Figure 7: Comparison of energy consumptions and execution times of the generated precision allocations from Table 4. White (resp., black) vertical bars give energy consumption of four generated precision allocations compiled with g++ (resp., clang++); the left x axis shows energy in nanojoules. Blue solid (resp., red dashed) lines give execution times of four generated precision allocations compiled with g++ (resp., clang++); the right x axis shows execution times in nanoseconds.

the generated assembly code to ensure that the prescribed mixed-precision allocations are indeed enforced. In the process, we became aware of many compilation issues stemming from the use of mixed-precision, such as unexpected interactions with other compiler optimizations (e.g., redundant memory accesses when using `-O0` flag, omitted fused multiply-add (FMA) instructions). Extensive and detailed exploration of these issues is a research project in its own right and beyond the scope of this work; nonetheless, we do describe next in more detail two such issues that caused unexpected execution times and energy usage.

Benchmark *jetEngine* contains floating-point constants that must be loaded into registers during execution. The $0.1E$ mixed-precision version of this benchmark ends up using the same constant as both a single- and double-precision floating-point number. If enough floating-point registers were available, this constant could have been kept in a register and converted as needed. However, register pressure due to the extra registers used causes this register value to be overwritten and reloaded later; this in turn causes an add instruction to be repeated. We illustrate this situation in Figure 8, where these extra assembly instructions are underlined.

Benchmark *coneArea* surprisingly runs ten times faster when compiled with g++ than with clang++. Our exploration reveals that this is the result of g++ using the fast ARM vector instruction `vsqrt` to compute the square root, and only if the result is

```
vmul.f64 d18, d0, d0      | vmul.f64 d17, d0, d0
vmov.f64 d19, #8          | vmov.f64 d19, #8
vmul.f64 d17, d18, d19    | vmul.f64 d20, d17, d19
vadd.f64 d1, d1, d1       | vadd.f32 s7, s7, s7
                          | vcvt.f64.f32 d18, s7
vmov.f64 d21, #112        | vmov.f64 d16, #112
vadd.f64 d16, d17, d1     | vadd.f64 d18, d20, d18
vadd.f64 d21, d18, d21    | vadd.f64 d16, d17, d16
vsub.f64 d16, d16, d0     | vsub.f64 d18, d18, d0
vdiv.f64 d16, d16, d21    | vdiv.f64 d18, d18, d16
vmov.f64 d24, #16         | vmov.f32 s10, #16
vmov.f64 d20, #24         | vmov.f32 s15, #24
                          | vcvt.f32.f64 s14, d18
nmls.f64 d20, d16, d24    | vnmls.f32 s15, s14, s10
vadd.f64 d23, d0, d0      | vadd.f64 d16, d0, d0
                          | vmov.f32 s12, #8
                          | vcvt.f32.f64 s13, d17
                          | vcvt.f32.f64 s11, d16
vmul.f64 d20, d18, d20    | vmul.f32 s15, s13, s15
vmul.f64 d23, d23, d16    | vmul.f32 s11, s11, s14
vsub.f64 d22, d16, d19    | vsub.f32 s12, s14, s12
vmla.f64 d20, d23, d22    | vmla.f32 s15, s11, s12
                          | vmov.f32 s14, #112
                          | vadd.f32 s13, s13, s14
vmul.f64 d17, d17, d16    | vmul.f32 s15, s15, s13
                          | vcvt.f64.f32 d16, s15
vmla.f64 d17, d20, d21    | vmla.f64 d16, d20, d18
vmla.f64 d17, d18, d0     | vmla.f64 d16, d17, d0
vadd.f64 d17, d17, d0     | vadd.f64 d16, d16, d0
vmla.f64 d17, d16, d19    | vmla.f64 d16, d18, d19
vadd.f64 d0, d0, d17      | vadd.f64 d0, d0, d16
vadd.f64 d0, d0, d16      | vadd.f64 d0, d0, d16
```

Figure 8: Code excerpts in ARM assembly from all-double (left) and $0.1E$ (right) versions of *jetEngine*. They illustrate a compilation issue we observed where mixed-precision increases register pressure and causes constants to be spilled to memory (red underlined instructions).

NaN the computation is repeated in software to appropriately set the `errno` flag as prescribed by the C standard. On the other hand, clang++ only uses a software implementation of square root, which is much slower than its hardware counterpart. Note that the usage of hardware-implemented square root can be forced using the `-ffast-math` compiler flag, but this also enables undesirable optimizations.

7. Discussion

In this section, we discuss the performance of FPTUNER, handling of conditionals and loops, scaling our methodology to large programs, and tuning for more than two candidate precisions.

7.1 Performance of FPTUNER

Figure 9 shows the runtimes of FPTUNER, including the proportions of its three main parts: parsing the given expression, invoking Gelpia for calculating the bounds of the first derivatives, and solving the QCQP problem. We can observe that the runtime of FPTUNER is dominated by the runtime of its global optimizer, namely Gelpia. Note that we do not show the runtimes for tuning *carbonGas* (1200s), *jetEngine* (38s), and *reduction* (90s) since they are much higher. However, we summarize that *carbonGas* takes a larger proportion of its time in Gelpia, while *reduction* takes a larger proportion in parsing.[6]

As noted in §5, we also need to verify that the allocation obtained by setting the higher-order error to 0 still passes the given

[6] Our parser is a quick prototype and can be substantially improved.

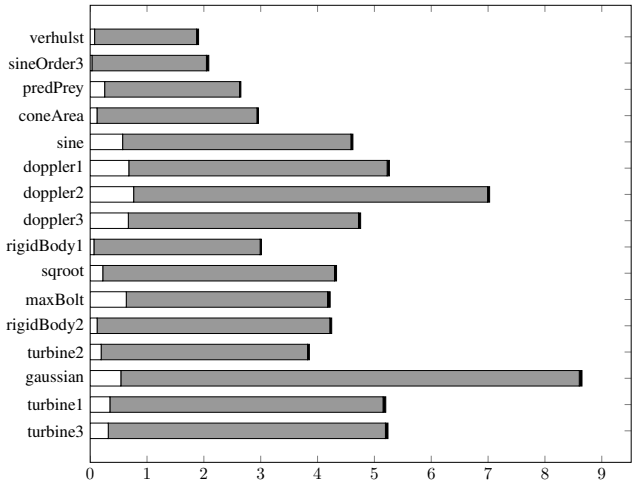


Figure 9: FPTUNER runtimes for tuning our benchmarks with an error threshold of $0.2E$, with each bar (x axis) showing the elapsed time (seconds). The white bands denotes parsing time and gray denotes Gelpia runtime to calculate the bounds of all the first derivatives. The short black band at the tip denotes the time taken by Gurobi to solve the QCQP instances.

error threshold. However, as noted in the same section, this step has passed the first time on all our benchmarks. Hence, verifying the allocation is just a one-of activity, taking as much time as one invocation of FPTaylor.

7.2 Conditionals and Loops

One of the limitations of our precision tuning work is that conditional expressions and loops are not handled. While these may sound quite limiting, rigorous error estimation in the presence of conditionals and loops is known to be a difficult problem *even without precision tuning*. For example, consider a simple real-valued expression f(x) = if $c(x) < 0$ then $f_1(x)$ else $f_2(x)$. The corresponding floating-point expression is then

$$\tilde{f}(x) = \text{if } \tilde{c}(x) < 0 \text{ then } \tilde{f}_1(x) \text{ else } \tilde{f}_2(x),$$

where $\tilde{c}(x) = c(x) + e_c(x)$, $\tilde{f}_1(x) = f_1(x) + e_1(x)$, and $\tilde{f}_2(x) = f_2(x) + e_2(x)$; here, $e_c(x), e_1(x)$ and $e_2(x)$ represent round-off errors. Suppose the error $e_c(x)$ is bounded by a constant E_c, i.e., $\left|e_c(x)\right| < E_c$. Now we need to consider four cases, two when both $f(x)$ and $\tilde{f}(x)$ take the same path and two when they take different paths (this phenomenon is known as "divergence"):

1. Find E_1 s.t. $c(x) < 0 \implies \left|\tilde{f}_1(x) - f_1(x)\right| \leq E_1$.
2. Find E_2 s.t. $c(x) \geq 0 \implies \left|\tilde{f}_2(x) - f_2(x)\right| \leq E_2$.
3. Find E_3 s.t. $-E_c < c(x) < 0 \implies \left|\tilde{f}_2(x) - f_1(x)\right| \leq E_3$.
4. Find E_4 s.t. $0 \leq c(x) < E_c \implies \left|\tilde{f}_1(x) - f_2(x)\right| \leq E_4$.

Finally, we compute the final error as $E = \max\{E_1, E_2, E_3, E_4\}$. Unfortunately, obtaining tight bounds for the error in cases 3 and 4 is known to be difficult. Worse still, when conditionals nest, an exponential number of path combinations need to be considered. The added difficulty of conditionals nested within loops is clear; recent related work [19] discusses preliminary steps in this area based on over-approximations.

Precision allocation adds another dimension to divergence. For example, it is very easy to find an input x such that under single-precision, the 'then' part is executed in $f(x)$ while under double-precision the 'else' part is executed (Chiang et al. [15] have pro-

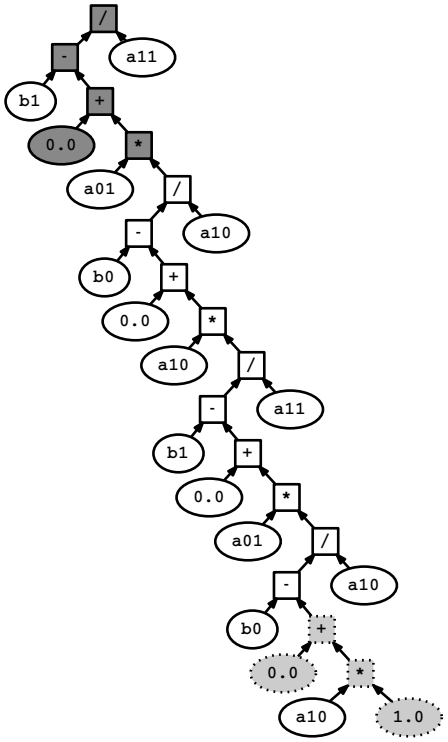


Figure 10: Tuning results for the Jacobi benchmark using three candidate precisions. The computation is represented as an expression tree, where ovals denote variables and constants, and squares denote operators. Dark (resp., light+dotted, white) ovals/rectangles denote single-precision (resp., double-, quad-) variables/operators. Note that we do not show the casting operators explicitly.

posed such a method for divergence detection). This issue raises a fundamental question: can precision allocation be allowed to change the branching behavior of a program? Our future work will study this issue more deeply by restricting tuning to cases where the exact branchings do not affect the higher level correctness assertions (e.g., iteration schemes where an extra iteration being taken is acceptable).

7.3 Improving Scalability

Floating-point round-off error is often much higher for specific input combinations [14, 55]. A generated mixed-precision allocation may be influenced by such rare input combinations. This suggests that having a separate code variant for processing the more frequently encountered inputs may be profitable. The adaptive geometric computation method proposed by Shewchuk [51] is an example of such input-aware code variant selection method.

We have explored the use of machine learning to synthesize an adaptive code variant selector (i.e., precision allocation selector) that is driven by inputs. We have, in fact, extensively evaluated the use of a decision-tree input decoder to pick the "optimal" code variant for a given input range. Our preliminary results are encouraging and we plan to pursue this direction further. An interesting feature of this hybrid approach is that FPTUNER is used to rigorously generate labels for feature vectors, and machine learning takes over to provide added scalability. We believe that achieving scalability and rigor will require such combinations.

7.4 Tuning with Three Candidate Precisions

So far in this paper, we have discussed only two-precision allocation situations (quad/double or double/single). In this section, we demonstrate that FPTUNER can quite naturally tune across three precision regimes (single, double, and quad). For this demonstration, we created a synthetic benchmark that implements the Jacobi iterative method for solving linear systems. The benchmark takes a 2×2 matrix and a vector of size 2 as inputs, and unrolls two iterations of the Jacobi method to compute the output vector. When we set the round-off error of the first element of the output vector to be less than 1e-14, FPTUNER automatically obtained the three-precision allocation shown in Figure 10. Curiously, FPTUNER allocates single-precision closer to the root and higher precision closer to the leaves. This is presumably to control the error at the lower levels of the tree so that it does not get amplified going up the tree.

We suspect that programmers are more likely to exploit FPTUNER's three-precision tuning capabilities to choose between three hardware-supported precisions — say double-, single-, and half-precision, all supported in hardware, or perhaps FPGA implementations that may provide quad-, double-, and single-precision also all in hardware.

8. Related Work

Following pioneering work by Kahan in the mid-eighties leading to the IEEE 754 standard, the properties of floating-point are now widely documented [24, 25]. Yet, with the inherent difficulties of comprehending floating-point behavior, its growing usage in all walks of life, and the herculean task of educating programmers [3, 35], informal approaches are inherently dangerous, as they are highly likely to leave a sprinkle of bugs in programs. Bold approaches to "end all errors" [29] are not currently practical. We now survey related work in correctness and optimization of floating-point computations.

8.1 Correctness and Error Analysis

Floating-point arithmetic had been formalized in many proof assistants, such as Coq [4, 16] and HOL Light [31, 34]. Harrison pioneered the verification of floating-point algorithms through mechanical theorem proving [30], evolving many algorithms that found their way into Intel products. Boldo et al. [5] have used Coq for certifying the implementation of numerical algorithms. An SMT-LIB theory of floating-point numbers was first proposed by Rümmer and Wahl [49] and recently refined by Brain et al. [7]. There have been several attempts to devise an efficient decision procedure for such a theory [6, 8, 39], 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 [21], a tool based on interval arithmetic. Darulova and Kuncak [17, 18] combine affine arithmetic and SMT-solving. Their method also drives a rigorous program-wise precision tuning tool called Rosa. The tool FLUCTUAT [26] combines the error estimates obtained from input-domain subdivisions to improve the overall accuracy of error analysis. Magron et al. [41] proposed a tool called Real2Float that models errors using semidefinite programming. Recently, Solovyev et al. proposed a tool called FPTaylor that models round-off errors using the Symbolic Taylor Expansions approach [52], and obtains rigorous error bounds using global optimization. FPTaylor also produces HOL-light proof certificates for each such error estimate. FPTUNER was inspired by FPTaylor, and employs a modified version of the Symbolic Taylor Expansions to capture round-off error under a generic precision allocation.

As already noted in §7.2, rigorous estimation of round-off errors in programs containing loops and branches is hard. Some approaches include loop unrolling [26] and constructing summary functions [19]. Even so, they require non-nested conditionals and constant loop bounds.

8.2 Code Improvement and Precision Tuning

Rubio-Gonzàlez et al. [47] proposed a delta-debugging approach implemented in a tool called Precimonious. Unlike FPTUNER, Precimonious assigns bit-widths to program variables and leaves the bit-widths of operations to be automatically inferred; FPTUNER allows individual bit-width assignments to operators. Precimonious can be accelerated by a pre-processing blame analysis process [48] that empirically identifies variables that do not significantly affect program behavior, and thus are safe to be assigned lower bit-widths. Lam et al. [37] proposed a breadth-first-search based method (implemented in a tool called CRAFT) that aggressively assigns lower bit-widths to a block of instructions. If the precision requirement is not met, the allocation method gradually converts some portions of instructions in the block to higher bit-widths until a specified precision requirement is satisfied. Graillat et al. [27] propose a tuning approach similar to Precimonious but use discrete stochastic arithmetic (DSA) for confirming the precision requirement. These coarse-grained dynamic tuning approaches do not come with rigorous guarantees, except on the given set of inputs. For example, for the jet benchmark with an error threshold of 1e-4, we have observed Precimonious generating a precision assignment that actually results in an error of 3e-3 in the interval spanned by the training points (30 times the set error threshold). FPTUNER is currently unable to handle the larger Precimonious benchmarks, especially those that contain loops and conditionals. A combined tool is attractive: use Precimonious to identify promising code segments to tune, and employ FPTUNER as a helper to tune these segments.

Some recent efforts have focused on improving accuracy through program rewriting. Tang et al. [53] propose a method that automatically searches for possible expression rewrites from a database of templates. Each template is a specification of two mathematically equivalent expressions. The objective of this method is to improve the numerical stability of programs. A similar rewrite technique was also applied to tuning fixed-point programs [20]. Panchekha et al. [45] 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 [42] for improving floating-point precision. This technique also takes into consideration standard compile-time techniques such as loop unrolling.

Several rewriting-based methods are geared toward effecting precision-efficiency trade-offs. Ansel et al. [12] proposed a programming language and compiler that automatically search and compose a program from algorithmic variants. Schkufza et al. [50] offer a Markov Chain Monte Carlo (MCMC) based method that searches for improved-efficiency compositions of instructions. In Schkufza's work, an instruction composition is considered a valid rewrite if it improves performance while meeting round-off error thresholds. All of the above rewriting-based methods check floating-point precision through input *sampling.* Therefore, their guarantees are only as good as the sample-size, which is typically a minuscule fraction of the total input space. Recently, Lee et al. [38] proposed a verification method that combines instruction rewriting and rigorous precision measurement. This approach can, for instance, be used to prove the correctness of the rewrites generated by Schkufza's method.

There have been some recent efforts aimed at discovering program inputs that cause high round-off errors. The presence of such high-error-causing inputs can also serve as an empirical witness to needing higher floating-point precision (for those inputs). Chiang et al. [14] have proposed search heuristics, and Zou et al. [55] a genetic-algorithm-based method.

Round-off errors can affect floating-point conditionals by making a program take inconsistent branches in sequence. Chaudhuri et al. [13] proposed a method that conservatively detects inconsistencies by (pessimistically) considering *every* conditional to be essentially a non-deterministic choice. Under this over-approximation, they proved the violation of axioms to be preserved by a collection of geometric primitives. Result deviation (also called divergence) has also been studied by Chiang et al. [15] who proposed search heuristics that can detect when a program can change its branching behavior when precision is reallocated. Machine learning has been employed in the field of code variant selection for performance optimization [36, 44, 54].

9. Conclusions and Future Work

Precision/energy trade-off is the central thrust in the design of future computing systems at all scales. In this paper, we provide a rigorous approach to tune the precision of the operators and operands in a conditional-free expression for a user-specified set of input intervals and error threshold. We offer the first rigorous solution in this space through formal error analysis based on global optimization using Symbolic Taylor Forms, and a versatile quadratically constrained quadratic programming formulation. Our tool FPTUNER embodying these ideas uses a constraint-based approach to search for precision allocations that meet practically important optimization goals such as limiting the number of operators at a given precision, grouping operators to share the same precision, and limiting the number of type-casts. We provide, for the first time, true insights on which of the low precision choices actually improve performance and reduce energy consumption. We also offer plenty of insights on how precision choices and compiler optimizations collide. These results are ripe for exploitation in at least two domains: (1) the design of rigorously characterized and optimized arithmetic libraries; (2) the development of precision-oriented compiler peephole optimization passes.

The two previously published coarse-grained precision tuning efforts can process much larger programs that include conditionals, loops, and procedures. These tools tune the code through direct execution on a few dozen inputs, and have shown performance improvements on many benchmarks. Unlike our effort, they do not provide rigorous guarantees across user-given input intervals. They also do not address the issue of divergent conditionals nor offer a constraint-based approach to search for precision allocations.

There appear to be two promising directions to achieve scale. First, the existing coarse-grained tools can identify promising code segments that FPTUNER can then fine-tune. Second, one can apply FPTUNER to help develop multiple code versions through machine learning from which, at runtime, one can select the right code version for the right input ranges. In addition to adding a few more tightening steps into our framework (e.g., equating noise variables) and automating our work flow, some of our other future work plans are: (1) allow different error models (e.g., L2 norm), (2) handle conditionals and loops, (3) improve scalability, and (4) actually integrate FPTUNER inside a compiler.

Acknowledgments

We thank Hari Sundar for helping out with the energy measurements, Cindy Rubio-González for providing support with Precimonious, Eva Darulova for distributing Rosa and its valuable benchmarks, and the anonymous reviewers for their numerous comments and suggestions. This work was supported in part by NSF awards CCF 1531140, CCF 1643056, and CCF 1552975.

References

[1] J.-M. Alliot, N. Durand, D. Gianazza, and J.-B. Gotteland. Finding and proving the optimum: Cooperative stochastic and deterministic search. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*, pages 55–60, 2012.

[2] ARM NEON. ARM NEON general-purpose SIMD engine, 2016. URL `https://www.arm.com/products/processors/technologies/neon.php`.

[3] D. Bailey and J. Borwein. High-precision arithmetic: Progress and challenges, 2013. URL `http://www.davidhbailey.com/dhbpapers/hp-arith.pdf`.

[4] S. Boldo. *Deductive Formal Verification: How To Make Your Floating-Point Programs Behave*. Thèse d'habilitation, Université Paris-Sud, 2014.

[5] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.

[6] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design (FMSD)*, 45(2):213–245, 2014.

[7] M. Brain, C. Tinelli, P. Rümmer, and T. 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.

[8] A. Brillout, D. Kroening, and T. 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.

[9] M. Burtscher, I. Zecena, and Z. Zong. Measuring GPU power with the K20 built-in sensor. In *Proceedings of the 7th Workshop on General Purpose Processing Using GPUs (GPGPU)*, pages 28–36, 2014.

[10] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Exploiting mixed precision floating point hardware in scientific computations. In *High Performance Computing and Grids in Action*. IOS Press, 2008.

[11] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Transactions on Mathematical Software (TOMS)*, 34(4):17:1–17:22, 2008.

[12] C. Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman. Autotuning multigrid with PetaBricks. In *Proceedings of the 21th International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2009.

[13] S. Chaudhuri, A. Farzan, and Z. Kincaid. Consistency analysis of decision-making programs. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 555–567, 2014.

[14] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamarić, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 43–52, 2014.

[15] W.-F. Chiang, G. Gopalakrishnan, and Z. Rakamarić. Practical floating-point divergence detection. In *Proceedings of the 29th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 271–286, 2016.

[16] Coq. The Coq proof assistant, 2016. URL `https://coq.inria.fr`.

[17] E. Darulova and V. Kuncak. Trustworthy numerical computation in Scala. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 325–344, 2011.

[18] E. Darulova and V. Kuncak. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 235–248, 2014.

[19] E. Darulova and V. Kuncak. Towards a compiler for reals. *CoRR*, abs/1410.0198, 2016.

[20] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha. Synthesis of fixed-point programs. In *Proceedings of the 11th ACM International Conference on Embedded Software (EMSOFT)*, pages 22:1–22:10, 2013.

[21] F. De Dinechin, C. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC)*, pages 1318–1322, 2006.

[22] S. Gao, S. Kong, and E. 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.

[23] Gaol. Gaol, not just another interval library., 2016. URL `https://sourceforge.net/projects/gaol`.

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

[25] F. Goualard. How do you compute the midpoint of an interval? *ACM Transactions on Mathematical Software (TOMS)*, 40(2):11:1–11:25, 2014.

[26] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proceedings of the 13th International Static Analysis Symposium (SAS)*, pages 18–34, 2006.

[27] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuiliére. PROMISE : floating-point precision tuning with stochastic arithmetic. In *17th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN)*, 2016.

[28] Gurobi. Gurobi optimizer, 2016. URL `http://www.gurobi.com`.

[29] J. L. Gustafson. *The End of Error: Unum Computing*. Chapman and Hall/CRC, 2015.

[30] J. Harrison. Floating-point verification using theorem proving. In *Proceedings of the 6th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM)*, pages 211–242, 2006.

[31] HOL Light. The HOL Light theorem prover, 2016. URL `http://www.cl.cam.ac.uk/~jrh13/hol-light`.

[32] IEEE 754. IEEE standard for floating-point arithmetic, 2008.

[33] Intel Intrinsics for Type Casting. Intel intrinsics guide, 2016. URL `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#cats=Cast`.

[34] C. Jacobsen, A. Solovyev, and G. 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.

[35] W. Kahan. How futile are mindless assessments of roundoff in floating-point computation?, 2006. URL `https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf`.

[36] M. Lagoudakis and M. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning (ICML)*, pages 511–518, 2000.

[37] M. Lam, J. Hollingsworth, B. de Supinski, and M. Legendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th International Conference on Supercomputing (ICS)*, pages 369–378, 2013.

[38] W. Lee, R. Sharma, and A. 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.

[39] M. Leeser, S. Mukherjee, J. Ramachandran, and T. 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.

[40] lquadmath. The GCC quad-precision math library, 2016. URL `https://gcc.gnu.org/onlinedocs/libquadmath.pdf`.

[41] V. Magron, G. A. Constantinides, and A. F. Donaldson. Certified roundoff error bounds using semidefinite programming. *CoRR*, abs/1507.03331, 2015.

[42] M. Martel. Program transformation for numerical precision. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 101–110, 2009.

[43] T. P. Morgan. Nvidia tweaks Pascal GPU for deep learning push, 2015. URL `http://www.nextplatform.com/2015/03/18/nvidia-tweaks-pascal-gpus-for-deep-learning-push`.

[44] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro. Nitro: A framework for adaptive code variant tuning. In *Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 501–512, 2014.

[45] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. 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.

[46] QCQP. Quadratically constrained quadratic program, 2016. URL `https://en.wikipedia.org/wiki/Quadratically_constrained_quadratic_program`.

[47] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the 25th International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2013.

[48] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1074–1085, 2016.

[49] P. Rümmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *Informal Proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[50] E. Schkufza, R. Sharma, and A. 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.

[51] J. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–363, 1997.

[52] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. In *Proceedings of the 20th International Symposium on Formal Methods Formal (FM)*, pages 532–550, 2015.

[53] E. Tang, E. Barr, X. Li, and Z. 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.

[54] R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, 2004.

[55] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei. A genetic algorithm for detecting significant floating-point inaccuracies. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 529–539, 2015.