Cost of Soundness in Mixed-Precision Tuning

ANASTASIA ISYCHEV, TU Wien, Austria
DEBASMITA LOHAR, Karlsruhe Institute of Technology, Germany

Numerical code is often executed repetitively and on hardware with limited resources, which makes it a perfect target for optimizations. One of the most effective ways to boost performance—especially in terms of runtime—is by reducing the precision of computations. However, low precision can introduce significant rounding errors, potentially compromising the correctness of results. Mixed-precision tuning addresses this trade-off by assigning the lowest possible precision to a subset of variables and arithmetic operations in the program while ensuring that the overall error remains within acceptable bounds. State-of-the-art tools validate the accuracy of optimized programs using either sound static analysis or dynamic sampling. While sound methods are often considered safer but overly conservative, and dynamic methods are more aggressive and potentially more effective, the question remains: how do these approaches compare in practice?

In this paper, we present the first comprehensive evaluation of existing mixed-precision tuning tools for floating-point programs, offering a quantitative comparison between sound static and (unsound) dynamic approaches. We measure the trade-offs between performance gains, utilizing optimization potential, and the soundness guarantees on the accuracy—what we refer to as the *cost of soundness*. Our experiments on the standard FPBench benchmark suite challenge the common belief that dynamic optimizers *consistently* generate faster programs. In fact, for small straight-line numerical programs, we find that sound tools enhanced with regime inference match or outperform dynamic ones, while providing formal correctness guarantees, albeit at the cost of increased optimization time. Standalone sound tools, however, are overly conservative, especially when accuracy constraints are tight; whereas dynamic tools are consistently effective for different targets, but exceed the maximum allowed error by up to 9 orders of magnitude.

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

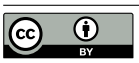
Additional Key Words and Phrases: Floating-Point Optimization, Mixed-Precision Tuning, Static Analysis, Dynamic Analysis, Soundness

ACM Reference Format:
Anastasia Isychev and Debasmita Lohar. 2025. Cost of Soundness in Mixed-Precision Tuning. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 359 (October 2025), 28 pages. https://doi.org/10.1145/3763137

1 Introduction

Numerical software is frequently used in resource-constrained systems, where it is essential to utilize available computational and memory resources efficiently. However, writing efficient and reasonably accurate numerical code from scratch is challenging due to the unintuitive nature of finite-precision arithmetic. Developers must account for issues such as over- and underflows, special values (e.g., $\pm\infty$, Not-a-Number), and rounding errors. This complexity creates a strong demand for automated methods that analyze program accuracy and generate optimized implementations. As a result, extensive research has been dedicated to developing *analysis* [19, 26, 28, 52, 53] and *optimization* techniques [13, 15, 17, 40, 47, 50] specifically designed for finite-precision numerical

Authors' Contact Information: Anastasia Isychev, TU Wien, Vienna, Austria, anastasia.isychev@tuwien.ac.at; Debasmita Lohar, Karlsruhe Institute of Technology, Karlsruhe, Germany, debasmita.lohar@kit.edu.



ACM 2475-1421/2025/10-ART359
https://doi.org/10.1145/3763137

programs. When optimizing numerical code, the optimizer must ensure that the optimized code computes meaningful results while satisfying some accuracy bounds. Therefore, optimizers track rounding errors to ensure they remain within or close to user-defined thresholds.

Typically, numerical code is optimized for accuracy or performance in terms of the program's running time, power, or energy consumption. However, the accuracy and performance objectives conflict: high-precision computations improve accuracy but are slower and energy-demanding, whereas low-precision computations enhance performance and reduce energy consumption, but at the expense of accuracy. Optimizing numerical software thus requires carefully balancing this trade-off, achieving performance and energy efficiency while keeping an acceptable level of accuracy.

In this paper, we consider one of the most widely used performance optimizations for finite-precision: *mixed-precision tuning* for floating-point programs [50]. The key insight behind this optimization is that different operations in a numerical algorithm contribute differently to the overall error. Some operations magnify errors and therefore require higher precision to avoid significant accuracy loss, while others naturally mitigate errors and thus may be computed in lower precision. Mixed-precision tuning explores how much performance gain can be achieved while keeping the total error within an acceptable user-defined limit. The performance is regulated by assigning potentially different precisions to each arithmetic operation, function call, and variable in the code (hence, *mixed* precision), aiming to use the lowest precision as often as possible, and high precision only when necessary to preserve accuracy.

The acceptable error threshold is specified by a user and expresses a hard constraint on how much accuracy loss can be tolerated by the program while keeping results meaningful. To make sure the accuracy constraint is satisfied, while searching for an optimal program, mixed-precision tuning computes rounding error on each candidate implementation. Based on the underlying rounding error analysis approach, optimizers can be categorized as *sound static* [12, 13, 15, 17, 48], where a worst-case sound error bound is computed, typically through static analysis of the code, and *dynamic* [8, 12, 30, 49, 50, 57], where the rounding error bound is estimated on sampled inputs.

Static approaches compute worst-case error bounds and are often overly conservative in their estimates. Additionally, they struggle to scale to large programs, particularly in the presence of complex control-flow structures such as loops and conditional branches [19, 37, 43]. In such cases, static methods often require pre-processing techniques such as explicit loop unrolling or specialized handling of branching behavior. Despite these limitations, sound static analysis provides strong correctness guarantees, making it indispensable for safety-critical applications where the cost of a malfunction is too high.

In contrast, dynamic approaches are more flexible in handling diverse and complex code and, when given sufficiently many program executions, provide realistic bounds on errors that may exclude critical outliers. Typically, though unsound, dynamic approaches are expected to be more efficient in optimizing programs.

Both static and dynamic optimizers must strike a balance between accuracy and performance, but they differ in terms of practicality and safety in terms of correctness guarantees. Despite extensive research in mixed-precision tuning in both directions, there has been no systematic study evaluating how these fundamentally different approaches compare in achieving optimization goals.

To bridge this gap, this paper presents the first comprehensive evaluation of mixed-precision tuning techniques for floating-point programs, comparing optimizers guided by static and dynamic analyses. Our goal is to enable a *quantitative* comparison of static and dynamic tools on programs that *both* can handle, which necessarily excludes programs with complex control flow or loops where dynamic approaches are the only option. On this common ground, we evaluate each approach's ability to balance accuracy and performance, analyze the safety of the optimized programs by

computing how often they exceed the maximum allowed error, and measure the optimizer's running time. Ultimately, we seek to answer a fundamental question:

What is the cost of soundness in mixed-precision tuning, and is it a price worth paying?

There are many mixed-precision tuning tools available, ranging from small-scale research prototypes to widely used optimizers [8, 12, 13, 15, 17, 30, 34, 48–50, 57]. First, we provide an overview of existing dynamic and sound static tools that optimize specifically *floating-point* programs (IEEE-754 [1] or arbitrary floating-point precision [25]). We then perform a quantitative comparison of the programs optimized by these tools. For the experimental part of the study, we selected tools based on the following criteria: the optimizer must 1) be publicly available, 2) compile and install with reasonable manual intervention, and 3) tune between 64-bit double and the next highest floating-point precision (128-bit quad or 80-bit long double). Specifically, we successfully installed and ran seven optimizers: four static tools—Daisy [17], FPTuner [13], and Regina [48] combined with both Daisy and FPTuner, and three dynamic tools Precimonious [50], POPiX [7], and HiFPTuner [30].

Using these tools, we optimized 100 loop- and branch-free benchmarks from the standard FPBench suite [16] for three different target errors (half, fifth and order errors). Our study explicitly quantifies the trade-off between the flexibility of dynamic approaches and the guarantees provided by sound static optimizers using an abstract *cost of soundness*. The cost consists of two metrics of the optimized programs: performance improvement and how much of the user-defined error budget is utilized. Specifically, we measure performance improvement as a relative difference in running time of the optimized programs compared to the baseline 128-bit quad implementation (and 80-bit implementation for Precimonious and HiFPTuner), and use of the budget as a ratio of the dynamically sampled absolute errors to the target error. The dynamic errors are measured with respect to a 300-bit MPFR implementation [25].

Our experimental evaluation indicates that contrary to the common belief, *soundness does not always come with the cost*. We find that the sound static approach combined with regime inference (Regina + Daisy) achieves the best overall cost, winning across all error targets on two-thirds of the benchmarks without compromising accuracy guarantees. Standalone static optimizers are less effective as they tend to be overly conservative, particularly for smaller error budgets that are more difficult to meet. Dynamic optimizers, on the other hand, optimize programs more aggressively. As a consequence, they achieve equally high speedups for all targets, at the cost of violating the accuracy constraint, sometimes exceeding the target by up to 9 orders of magnitude. We also find that, perhaps surprisingly, there is no clear trend of sound tools being significantly slower than dynamic unsound tools. The only outlier is Regina that iteratively calls the underlying sound optimizer, so the high running times are expected.

We discuss the lessons learned in our experimental comparison. The fact that our findings challenge common expectations in the community indicates the dire need for systematic studies and comparisons across different techniques.

Contributions. In summary, we present the following contributions.

- We present a comprehensive overview of existing static and dynamic mixed-precision optimizers for floating-point programs, highlighting their distinguishing features.
- We quantify the *cost of soundness* in mixed-precision tuning, enabling a clearer understanding of the trade-offs between sound guarantees and optimization effectiveness.
- Through extensive experiments on the standard FPBench benchmark suite, we systematically evaluate the cost of soundness across seven available tools. We summarize the key insights of our study that could serve the community navigating the broad and complex landscape of mixed-precision optimizers.

2 Terminology

Before summarizing the distinguishing features of existing optimizers, we introduce the terminology that will be used throughout the paper.

Definition 2.1. **Mixed-Precision Tuning**. A performance optimization technique that assigns potentially different finite-precision types to variables, function calls, and arithmetic operations. Its goal is to increase performance — typically, in terms of running time of the program — while keeping the overall rounding error below a user-defined bound. This optimization is based on the observation that operations performed on low precision (with few bits involved) are faster than the same operations in high precision (with more precision bits). Since some operations contribute to the overall error more than others, mixed-precision tuning attempts to assign low precision when possible and use high precision on operations that require higher accuracy.

Definition 2.2. **Static Optimizer**. A mixed-precision tuning optimizer that uses worst-case rounding error bounds, computed with sound static analysis, to determine if a candidate precision assignment satisfies the user-defined error bounds. In this paper, we use the terms *static* optimizer and *sound* optimizer interchangeably, because all static optimizers included in this study use underlying sound rounding error analysis.

Definition 2.3. **Dynamic Optimizer**. A mixed-precision tuning optimizer that uses dynamically sampled inputs to estimate rounding errors and determine whether the target error is satisfied.

Definition 2.4. **Performance**. We focus on one aspect of performance: running time of the optimized programs. Other metrics, such as power or energy consumption, area of a hardware chip, and so on, are typically directly proportional to running time.

Definition 2.5. **Target Error**. A user-specified *absolute* error bound that must be satisfied by the optimized program. Dynamic and static tools consider different ideal reference implementation. For static tools the error is specified with respect to the real-valued computation result, while for dynamic tools, the ideal implementation is finite-precision computation that uses high precision, which already includes rounding errors due to finite precision.

Definition 2.6. **MPFR**. The GNU MPFR (Multiple Precision Floating-Point Reliable) [25] is a C library for multiple-precision floating-point arithmetic with correct rounding. Its arbitrary assignment of precision bits allows more flexible precision allocation than the IEEE-754 standard precisions [1]. It is commonly used to simulate real-valued computations with very high precision.

3 State of the Art in Mixed-Precision Tuning

We begin with an overview of existing tools developed for mixed-precision tuning of floating-point programs. This section focuses exclusively on tools whose only goal is precision tuning; those that combine tuning with other numerical optimizations or tune for precisions other than floating-point are deferred to Section 6.

Table 1 provides a summary of the optimizers in chronological order, categorized according to the type of underlying analysis they employ: either static or dynamic. Columns 4 and 5 indicate tool availability and whether we could successfully install the tool: a '✓' denotes that the tool is publicly accessible (either as source code or binaries) and can be compiled and installed with reasonable manual effort, whereas a '✗' indicates the opposite. In some cases, we had to contact developers for instructions to troubleshoot a failing installation process. Columns 6 and 7 specify the input and output formats supported by each tool. For tools that are not publicly available (e.g., Reval), we rely on their respective publications to report supported formats. The final column denotes whether the tool was included in our comparison study ('✓' for inclusion). We exclude the

Table 1. Overview of state-of-the-art mixed-precision tuners (in chronological order). The last column indicates which tools are included in our comparison.

Analysis Type	Year	Tool	Available	Compiles	Input Format	Output Format	Included
Dynamic	2013	Precimonious [50]	✓	✓	C	JSON, Bitcode	✓
	2016	Blame Analysis [49]	✓	✗	C	JSON, Bitcode	✗
	2017	fpPrecisionTuning [34]	✓	✗	C	Text, Script	✗
	2018	HiFPTuner [30]	✓	✓	C	JSON, Bitcode	✓
	2020	PyFloT [10]	✓	✓	Python, C++, Fortran, CUDA	Text	✗
	2022	POPiX [8]	✓	✓	Python	Python, Text	✓
	2024	FPLearner [55]	✓	✓	C++	JSON	✗
Static	2017	FPTuner [13]	✓	✓	DSL	Text	✓
	2018	Daisy [17]	✓	✓	Scala	C++, Scala	✓
	2021	Regina [48]	✓	✓	Scala	C++, Scala	✓
	2024	Reval [57]	✗	-	FPCore 2.0	-	✗

tools that are not publicly available or could not be successfully installed from the evaluation. For other excluded tools, we provide justification in Section 4.

3.1 Dynamic Optimizers

We classify an optimizer as *dynamic* if it relies on the *sampled values* of inputs, intermediate variables, and outputs. Typically, a dynamic optimizer executes both a high-precision version (sometimes referred to as 'shadow execution') and a candidate optimized program on a set of sampled inputs. Based on the concrete values of each variable, result of an arithmetic operation and elementary function in the program, a dynamic optimizer estimates an error, either by computing the absolute difference between the high-precision and candidate outputs [30, 50] or by applying more sophisticated over-approximation techniques [8].

If the estimated error of a candidate program output satisfies the target error bound, the candidate is considered valid. Among valid candidates, the optimizer chooses one based on additional criteria, such as naively returning the first valid candidate found or selecting the one with the best performance (e.g., lowest runtime).

The crucial challenge of any mixed-precision optimizer is efficiently navigating the vast search space of possible precision assignments. A naive approach that changes the precision of one floating-point variable at a time and checks the resulting error is inefficient and scales poorly, with a worst-case time complexity of $O(n^2)$ for n variables. To explore the search space efficiently, the optimizers presented below implement more advanced strategies.

3.1.1 Precimonious. Precimonious [50] is one of the first and most prominent dynamic mixed-precision tuners for floating-point programs. Given an annotated program in C and a set of representative inputs, Precimonious attempts to find an efficient and accurate enough precision assignment. The required annotations include logging the reference output values and the target error. The target error can be expressed as an absolute value or relative to the sampled outputs, in which case it has to be computed during the sampling.

Precimonious searches the space of all possible precision assignments using a variation of the *delta-debugging* algorithm [59] (illustrated in Figure 1). At the beginning of the search, each variable is associated with a set of possible precisions. The algorithm refines these sets until there is only one precision left per variable. At every iteration, it considers a pair of the highest and second-highest

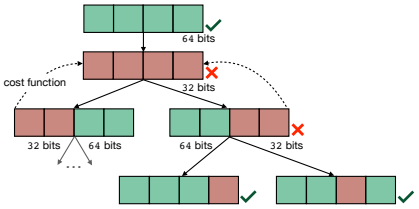


Fig. 1. The delta-debugging algorithm

precisions from the set per variable. If the version with the second-highest precision is sufficiently accurate, the highest precision is removed from the set. Effectively, the search iteratively lowers the precisions of subsets of variables until the accuracy constraint can no longer be satisfied, or there are no assignments to explore. Note that the algorithm is searching for a local minimum and does not guarantee the global lowest possible precision assignment.

For each candidate precision assignment, Precimonious performs two checks. First, it checks for accuracy by sampling the values and checking whether the difference to the reference (sampled) values is within the specified target error bound and discards all candidate programs that fail the check. Second, it checks for performance and dynamically measures the candidate program's performance score, and its running time. Finally, Precimonious chooses the precision assignment with the lowest seen score. If neither of the candidates improved running time compared to the original program, Precimonious reports that no configurations are found.

The original program in C is compiled to the low-level virtual machine (LLVM) [42] intermediate representation, and all precision modifications are performed directly on the bitcode. Because of this dependency, Precimonious requires the version of LLVM 3.0 and does not support any language or compiler features that came after 2013. The tuning is possible among the following set of standard precisions: `float`, `double`, and `long double` that represents an 80-bit floating-point type. As an output, if a local optimum is found, Precimonious generates the bitcode of the optimized program and a JSON file with the precision assignment for variables and function calls of the program.

3.1.2 Blame Analysis. One of the limitations of Precimonious is its slow optimization time. Since the performance of every candidate program is measured through actual execution, the total tuning time includes the cumulative runtime of all these executions. In the follow-up work, the authors of Precimonious introduce Blame Analysis [49], a technique that combines a concrete and shadow execution to reduce the search space for precision assignments and improve optimization time. Like Precimonious, Blame Analysis takes C programs as input and generates the bitcode of the optimized program and a JSON file with the precision assignment.

Blame analysis operates on the instruction level and performs only local fine-grained changes in the instructions without considering its impact on the global solution. Given a target precision for the result of a single floating-point instruction in the program, the shadow execution runs it with multiple precisions for the operands. For each variation, the analysis checks if the concrete result is accurate up to what the precision can express. It then records the minimum required operand precisions in a *blame set*. When a blame set contains only a single precision for a variable, that information can be used to prune the search space during the full-program tuning phase.

The main goal of the Blame Analysis is to reduce optimization time by narrowing the precision assignment search space. The analysis focuses on how different precision configurations affect a program's accuracy, and unlike Precimonious, it does not evaluate the performance of the optimized

programs. The authors note that Blame Analysis tends to lower the precision of more variables compared to Precimonious. However, this does not guarantee improved runtime performance.

3.1.3 HiFPTuner. Another follow-up to the original Precimonious addresses the scalability challenge from a different perspective. To reduce the need for an exhaustive exploration of all possible precision configurations, HiFPTuner [30] introduces a guided search strategy that leverages the dependencies among floating-point variables. This not only reduces the optimizer's running time but potentially generates better optimized programs. HiFPTuner also uses a similar input-output format as Precimonious.

The key insight behind HiFPTuner is that type casts affect programs' performance as much as using high precision. Therefore, minimizing type casts may improve the performance of the tuned program. Based on this insight, HiFPTuner builds a weighted dependence graph for read-after-write accesses to variables. It then iteratively analyzes the graph and builds a hierarchical community structure, where the variables that are likely to require the same precision are grouped together. Initially, all variables are placed in separate groups at the lowest level of the hierarchy. With each iteration of the graph analysis, they are merged at each level until the top hierarchy is formed. HiFPTuner searches the constructed hierarchical structure top-down to generate precision assignments. At each level of the structure, all precision alternatives for a group of variables are explored collectively. Such grouping prunes the search space both from the top down and through the hierarchy, thus improving overall optimization time and reducing the number of type casts.

3.1.4 fpPrecisionTuning. Another optimizer from the same period, fpPrecisionTuning [34], formulates the precision tuning as finding the minimum required word length, where the 'word length' refers to the number of bits in the floating-point mantissa. The tuning framework consists of two components. The first component is a converter that transforms standard C code into a version using arbitrary precision based on the MPFR library [25]. This converted program takes as input precision assignments (in bits) for each variable. The second component is the tuner itself, which operates in two phases.

First, the tuner computes so-called 'influence groups', a set of variables along a path π that are affected by the value of a variable x. These influence groups are constructed for each variable in the program. Within each group, the algorithm performs a binary search to find the minimal precision required for variable x, while keeping the precision of all other variables in the group fixed. To reduce the running time, this search is carried out in parallel across all influence groups.

In the second phase, the tuner selects among configurations for all influence groups the one with the most error reduction. It then iteratively increases precision from the minimal configuration identified in the first phase until the dynamically sampled error satisfies the user-defined target.

The two-phase tuning approach is performed for one fixed input. To account for ranges of inputs, fpPrecisionTuning further refines the obtained configurations by iteratively applying it to a training set consisting of multiple input samples. It then retains the highest precision required across those samples. The refinement is performed on a fairly small set of inputs (authors suggest 100 inputs) and aims to satisfy the user-specified error statistically, for instance, by ensuring that the average signal-to-quantization-noise ratio (SQNR) remains below a predefined quality threshold.

fpPrecisionTuning provides a configuration file with the number of mantissa bits per variable (in order of appearance) and a script to run the MPFR version along with this configuration. Adding the generated precision assignment in the original program in C has to be done manually.

3.1.5 PyFloT. A more recent addition to the space, PyFloT [10] is a dynamic tuner that targets full-scale high-performance computing (HPC) applications written in various languages, such as Python, C++, and Fortran, that repeatedly perform function calls. Unlike other methods listed in this

section, PyFloT does not tune the precision of individual variables and arithmetic operations, but focuses on choosing the best function among available implementations (with different precisions), and reports the results directly in the terminal. For instance, an application using the mathematical function *exp*() may have the option to choose between `expf(float)` for 32-bit single precision, `exp(double)` for 64-bit double precision, and `expq(quad)` for 128-bit quad precision results.

PyFloT collects traces of program execution and clusters *occurrences* of each function invocation based on a combination of two factors: static control-flow information and dynamically collected stack backtrace, so-called temporal locality. For each cluster of function invocations, PyFloT iteratively lowers precision and dynamically checks if the target error is satisfied. PyFloT performs best on applications with complex control flow and frequently repeated function invocations, which differs in spirit from other mixed-tuning optimizers and may be used as a complementary approach.

3.1.6 POPiX (also, POP). Precision OPtimizer (POP) [7] has been originally proposed as a tuning assistant for floating-point programs and later extended to fixed-point and arbitrary floating-point precisions and renamed to POPiX [8]. POPiX uses static analysis and integer linear program (ILP) constraint solving to determine precision assignment; however, here we intentionally classify it as a dynamic optimizer because the ranges for intermediate variables are determined dynamically using sampling and do not provide formal guarantees for inputs outside of sampled value ranges (as evidenced by our experiments in Section 4).

After sampling 10^5 inputs (hardcoded constant, can be adjusted in the source code), POPiX performs abstract-interpretation-based static analysis. First, it performs forward analysis to compute the number of digits up to which the result of each arithmetic operation is expected to be accurate. The analysis uses sampled ranges to obtain information about units in the last place (*ulp*) and unit in the first place (*ufp*) for each operation. The combination of both metrics is used to determine when the carry bit, an additional bit needed to represent the results of a computation, appears in each operation and provides a precise estimate of the number of accurate bits, referred to as *number of significant bits* (*nsb*). In the second phase, the backward analysis uses the target error specified as *nsb* and the results of the forward analysis to build a constraint system that represents required accuracy relations. The constraint system constitutes an ILP problem. Using an off-the-shelf solver GLPK [24], POPiX determines the minimum size of mantissa for each floating-point variable and operation. While other tools treat square root and elementary mathematical functions as atomic operations, POPiX uses custom digit-recurrence algorithm for the former and Taylor expansion for the latter, and iteratively refines the order of the approximation until a given *nsb* accuracy constraint is satisfied [8]. The resulting precision can be used directly for fixed-precision assignments or arbitrary floating-point precisions, such as MPFR. The precision can also be mapped back to the IEEE-754 floating-point types using the following function:

$$ieee\_fp(nsb) = \begin{cases} \texttt{float} & \text{if } nsb \leq 23 \\ \texttt{double} & \text{if } 24 \leq nsb \leq 52 \\ \texttt{quad} & \text{if } 53 \leq nsb \leq 112 \end{cases}$$

POPiX automatically generates a Python program with MPFR precisions and a fixed-point program using `fixmath.h` library [3]. We had to perform the translation to the IEEE-754 floating-point precisions ourselves.

3.1.7 FPLearner. Recently, there has been an effort to combine machine learning (ML) with traditional dynamic analyses to enhance the efficiency of precision tuning [55]. The proposed tool, FPLearner, utilizes a Gated Graph Neural Network to extract features from a Precision Interaction Graph, representing mixed-precision floating-point programs, and predicts their performance and

computational accuracy. By integrating with traditional dynamic precision tuners, FPLearner reduces the need for exhaustive precision assignments by predicting promising configurations with high speedup and acceptable error thresholds, executing only these configurations to validate predictions. Currently, FPLearner supports tuning between `single` and `double` precision; extending it to other formats, such as `quad`, would require retraining the model.

The workflow comprises three stages. In the *Pre-run Stage*, the dynamic precision tuner generates a small set of mixed-precision programs for the target application, labeling their performance and accuracy as ground truth for fine-tuning. During the *Fine-tuning Stage*, the pre-trained models are adapted to this limited dataset using techniques like random oversampling to address class imbalances, ensuring robust training without overfitting [46]. In the final *Optimization Stage*, the fine-tuned models direct the precision tuner by predicting the speedup and accuracy of candidate programs, executing only those classified as promising. Experimental evaluations on standard HPC benchmarks written in C++ demonstrate that FPLearner, when combined with tools like Precimonious and HiFPTuner, reduces the number of necessary candidate program executions by an average of 25.54% while maintaining or surpassing the quality of traditional dynamic analyses. The final precision assignments are produced in the same format as Precimonious or HiFPTuner.

3.2 Sound Static Optimizers

In contrast to dynamic optimizers that use sampled errors, *sound* mixed-precision tuners rely on *worst-case rounding error bound* to determine if a candidate precision assignment satisfies the target error. To compute the worst-case error, the underlying analysis uses symbolic abstractions of floating-point operations. Given a real-valued function $f(x)$ over an input domain I and its floating-point counterpart $\hat{f}(\hat{x})$, an absolute error can be expressed as

$$e_{abs} = max_{x \in I} \left| f(x) - \hat{f}(\hat{x}) \right|. \tag{1}$$

While relative error can be a meaningful accuracy measure, it is inherently difficult to compute due to division by zero [39]. Therefore, most tools compute absolute rounding error e_{abs}. Moreover, computing the difference between the real-valued function and its floating-point counterpart directly is difficult for two reasons. First, the real-valued result is not available for most programs. Secondly, the floating-point function $\hat{f}(\hat{x})$ is highly discontinuous due to the necessary rounding, since not all real-valued numbers can be exactly represented in floating points. Therefore, state-of-the-art tools abstract the $\hat{f}(\hat{x})$ function. Each arithmetic operation is modeled as follows:

$$x \hat{\circ} y = (x \circ y)(1 + \varepsilon) + d \tag{2}$$

where $|\varepsilon|$ is the relative error introduced by each operation bounded by the machine epsilon ϵ_M, the maximum relative error determined by the floating-point precision. The term $|d| \leq \delta$ represents maximum absolute error due to rounding of subnormal floats (numbers very close to zero with special representation of exponent bits in the floating-point precision). For instance, $\epsilon_M = 2^{-53}$ and $\delta = 2^{-1075}$ for double floating-point precision. Similar equations hold for the square root, unary minus, and the rounding of a real-valued number. Furthermore, elementary functions are modeled using the same equation, but the values of ϵ_M and δ are increased correspondingly. Using Equation 2, sound tools replace $\hat{f}(\hat{x})$ in Equation 1 and compute absolute errors either by solving an optimization problem or through data-flow-based analysis.

Global optimization-based analyses [13, 53] generally provide tighter error bounds, but often at the expense of higher computational cost. In contrast, data-flow-based approaches [15, 17, 28] are generally more efficient, though they yield more conservative error bounds. Below, we review tools that employ both kinds of analyses for sound mixed-precision tuning. Sound tuners using

either technique are typically applied to relatively simple floating-point programs without complex program structures like conditionals and loops. When tuning, optimizers do not distinguish between arithmetic operations and elementary function calls and treat them as atomic constructs.

3.2.1 FPTuner. FPTuner [13] adopts a sound approach to mixed-precision tuning by formulating the problem as a global optimization task. It takes as input a real-valued expression (in a domain-specific language), along with input domain specifications and a target error bound, and produces precision allocation for each operation in the expression. The tuning task is formulated as a quadratically constrained quadratic program (QCQP). The objective function consists of the weighted number of low precisions and their corresponding rounding errors. The QCQP problem can include additional constraints, such as limiting the number of type casts or grouping operations to have the same precision, which is useful for vectorization.

To construct the QCQP, FPTuner first generates a symbolic model of the program by iteratively applying the abstraction Equation 2. It then expresses the overall error on the program using symbolic Taylor expansions, and the underlying error analysis is as in FPTaylor [52]. To compute the worst-case error bound, the error expression is maximized over the entire input domain using Gelpia [4], a custom global optimizer developed by the authors that employs branch-and-bound techniques. The computed upper bound on the error is then incorporated as a constant weight to allocation variables, for which the optimization problem is ultimately solved. Note that the error bound depends on the allocation; therefore, the weights have to be computed for each allocation variable. The resulting optimization problem is solved using the Gurobi solver [31].

3.2.2 Daisy. In contrast to FPTuner, Daisy [17] employs a data-flow-based error analysis to guide precision tuning for numerical programs. In this paper, we focus on tuning for floating-point precisions, but note that Daisy also tunes for fixed-point precisions using the same procedure.

Daisy takes as input a real-valued program written in a DSL closely resembling Scala, annotated with input ranges and a target error bound. The tuning begins by assigning the highest available precision to all variables and then applying a delta-debugging search strategy similar to that of Precimonious (illustrated in Figure 1). However, instead of validating the errors on sampled values as Precimonious does, Daisy checks sound worst-case error bound.

To compute sound upper bounds on rounding errors, Daisy relies on *data-flow-based analysis* that uses intervals [33] and affine arithmetic [20] to track variables' ranges and error bounds. To further reduce over-approximations due to static analysis, Daisy offers an (optional) interval subdivision mode. When enabled, the subdivision splits the range of each input variable into a user-configurable number of sub-intervals, takes their Cartesian product, and analyzes each subdomain independently. Thus, it computes tighter error bounds but at the cost of increased runtime.

To decide among multiple precision assignments that satisfy the target error, Daisy uses a static cost function that assigns an abstract cost of 1, 2, and 4 to single, double, and quad precision arithmetic and casts respectively. Daisy also provides alternative cost functions, such as the number of operations in each precision, the inverse of rounding error magnitude, and a hardware-specific cost model obtained by benchmarking arithmetic operations execution time in different precisions.

The final output is a mixed-precision program in Scala or C++, including all required type casts, that is guaranteed to meet the specified target error. Similarly to Precimonious, Daisy relies on a heuristic search; it does not guarantee globally optimal precision assignments.

3.2.3 Regina. Regina [48] is a meta-optimization tool that infers effective *regimes* for sound floating-point optimizations. A regime is a partition of the program's input domain and a mapping from each sub-domain to the optimized program code. Regina can be combined with any sound optimization, including mixed-precision tuning.

Most optimizers, both static and dynamic, try to improve the program on the whole input domain. However, different parts of the input space contribute to the overall rounding error differently. It is common that large errors only occur for inputs from a small sub-domain and therefore have to be handled more carefully, while the rest of the input domain can be optimized more aggressively. Especially for sound optimizations that rely on the worst-case error bound, a small sub-domain inducing high errors blocks the optimization on the whole input domain, as treated by the tools. Regina utilizes this uneven distribution of errors and adjusts the optimization degree accordingly.

In a nutshell, it subdivides the program's input domain into boxes and applies optimization on each of the boxes separately. To prevent too fine-grained partition, Regina tries to merge boxes where the optimizer produced the same optimized expression. The tool proposes several alternative strategies for sub-dividing the input space, including splitting the domain into equal-sized boxes top-down, bottom-up (split fine-grained, then merge), and the genetic algorithm that splits a box at a random point. Regina applies these strategies alone or combined in a two-phase approach. For instance, the most successful inference strategy was shown to be a combination of the bottom-up inference refined by the genetic algorithm splits [48].

Regina was shown to be beneficial for mixed-precision tuning with different techniques — dataflow approach by Daisy and optimization-based approach by FPTuner— as well as for rewriting, an orthogonal optimization that improves the program's accuracy.

3.2.4 Reval. A recently proposed tool Reval explores mixed precision to efficiently mimic real-valued evaluation [57]. Real number evaluation usually relies on a strategy known as Ziv's onion-peeling method [60]. Ziv's method iteratively increases (uniform) precision until the result of an expression under evaluation is sufficiently accurate. Reval's goal is to increase the performance of Ziv's method by boosting the performance of each iteration and minimizing the number of iterations needed to converge to an accurate result. Building on symbolic Taylor approximation techniques (similar to those used by FPTuner) and condition numbers [61], Reval simplifies them specifically for interval arithmetic. It avoids computing higher-order terms with their novel *exponent trick*, which uses only the exponents of previously computed intervals to approximate amplification factors (how much the error is amplified by each operation). With this, Reval computes sound and reasonably tight error bounds quickly. Unlike other sound tools considered in this section, Reval relies on MPFR arbitrary precision rather than IEEE-754 standard floating-point arithmetic. At the time of writing (March 2025), Reval is not publicly available.

4 Quantitative Comparison

The main contribution of our work is to quantify the trade-off between the flexibility offered by unsound dynamic optimizers and the safety guarantees of sound tools. We evaluate the programs optimized with dynamic and static tools using multiple metrics. Specifically, we aim to answer the following research questions:

RQ1: What is the cost of soundness?
RQ2: Which optimizers achieve highest performance improvement?
RQ3: How do the optimizers use the target error budget?
RQ4: How fast do the optimizers generate programs?

4.1 Selection of Tools

In Section 3, we reviewed existing sound static and unsound dynamic tuners dedicated exclusively to mixed-precision tuning for floating-point programs that were available at the time of writing (March 2025). While we aimed to include as many tools as possible in our comparison, we could not run some due to various practical limitations. These include not being available in open-source (e.g.,

Reval), failure to build due to deprecated dependencies or a lack of support (e.g., Blame Analysis), unresolved usability issues even after partial fixes (e.g., fpPrecisionTuning), and require extensive modifications for a fair comparison using our evaluation criteria (e.g., FPLearner cannot tune between 64 and 128 bits). Additionally, some tools were excluded because their tuning objectives or application domains differ significantly from the focus of this paper, which we elaborate on below.

We did not apply Blame Analysis [49] because its primary goal is to reduce optimization time rather than improve the performance of the generated code. Furthermore, it has been superseded by HiFPTuner, which we do include in our comparison.

PyFloT [10] targets large HPC applications that repeatedly call functions, for which it tunes precision on each invocation. However, such large applications with complex control flow fall outside the scope of sound static tuners, which are typically designed for small, straight-line numerical programs (kernels). Moreover, PyFloT does not tune the precision of individual arithmetic operations that constitute most of the numerical kernels. Since there is little common ground for sound optimizers and PyFloT, such a comparison would not be meaningful, we therefore exclude PyFloT.

FPLearner currently supports tuning only between 32-bit single and 64-bit double precision. In contrast, our comparison focuses on tuning between 64-bit double and 128-bit quad precision (explained in Section 4.3). Including FPLearner would, therefore, require significant code modifications and retraining of its underlying models, making it impractical for this comparison.

The remaining tools listed in Table 1 are successfully installed and included in our comparison. In total, we evaluate 7 optimizers: 4 sound static approaches – Daisy, FPTuner, and their combinations with Regina– and 3 dynamic tools – POPiX, Precimonious, and HiFPTuner.

4.2 Benchmarks

Mixed-precision tuning takes as input a specification consisting of the program to be optimized and a target error— representing the maximum error tolerated by the application. Since our goal is to perform a *quantitative comparison* of sound static and unsound dynamic optimizations, we select benchmarks that can be handled by *both types of tools*. This requirement effectively excludes programs with loops and conditionals, as only dynamic tools can optimize such programs without reducing them to straight-line code. We perform our experiments on the benchmarks from the standard FPBench set [16], specifically on the subset from Regina's experimental setup. These contain straight-line code, all loops are reduced to a loop body (same as one iteration), and conditional statements are excluded. In total, we include 100 benchmarks, 32 of which contain elementary function calls, and 15 that contain the square root operator.

Regina's setup already contained benchmarks in the formats suitable for Daisy, FPTuner, and their combinations with Regina. For POPiX the input format is very similar to Python programs, which we pretty-print from Regina's benchmarks. Precimonious and HiFPTuner expect C code that follows the predefined template. We use Regina's printer to C and extend it to automatically add the (hardcoded) surrounding template (reading inputs, logging outputs, and performance score) to each benchmark. Note that for straight-line code numerical programs, such translation is fairly straightforward. The most effort was to deduce the necessary components in each input format and specification, and circumvent some special behavior of tools (for instance, POPiX required rewriting a unary minus operator `-x` into binary `0.0-x`).

Target Errors. Following previous work [13, 17, 48], we create three sets of target errors: *half-error*, *fifth-error* and *order-error*. It has been shown that mixed-precision tuning is most beneficial when the target error is just below the uniform precision error. Therefore, we first compute absolute

errors with double uniform precision, then scale them by 0.5x for half-, 0.2x for fifth- and 0.1x for order-error benchmarks.

We generate target errors with sound tools as they produce deterministic error bounds. Moreover, dynamic tools do not explicitly report the resulting error bound. Daisy and FPTuner (which uses FPTaylor's analysis internally) apply different error estimation techniques, and their computed errors can vary significantly. To ensure fair treatment of the tools, we generate separate target errors: Daisy's targets are based on Daisy's reported errors, while FPTuner's targets are derived from error bounds computed with FPTaylor. For all dynamic tools except POPiX we use Daisy's targets.

Unlike other tools that specify target error as a decimal number, POPiX does it using a number of significant bits. We translate the targets obtained with Daisy using a formula:

$$nsb = \lceil abs(log_2\ err) \rceil$$

where err is a target error bound in decimal format. For instance, an error $4.56e-14$ is translated into $nsb = 45$ (when translated back, it amounts to $2.84e-14$). Naturally, due to this translation, target error bounds do not exactly match the ones in Daisy, but they are the closest equivalent that we can specify. We use these targets to optimize with POPiX and evaluate its accuracy.

4.3 Setup

We tune the programs between 64-bit double and 128-bit quad precision, implemented by the GCC quadmath library [2]. This choice is consistent with prior work [13, 48], it has been observed that on machines equipped with a floating-point unit (FPU), single and double precisions typically result in similar performances, making tuning between double (implemented in hardware) and quad (implemented in software) more meaningful for performance trade-offs.

For Precimonious and HiFPTuner, the highest precision available during tuning was `long double`, which corresponds to 80-bit precision on x86 architectures. All our metrics evaluate relative change in performance in accuracy; we therefore use the performance and accuracy of `long double` implementations as the high-precision baseline for these tools. For completeness, we tried to bring all generated programs to a common ground and performed additional experiments where we upgraded all `long double` assignments in the optimized programs to `__float128` from GCC quadmath. However, this had a significant negative effect on the program's performance. As both Precimonious and HiFPTuner use *dynamically measured* program's running time to filter out precision assignments, we keep the original programs for a fair comparison. We comment on the differences between the 80-bit and 128-bit versions in Section 5.

Each optimizer was executed 3 times to account for potential fluctuations in running time, with a timeout of 30 minutes per run. For dynamic tools that rely on sampling (Precimonious, HiFPTuner), we provided different (fixed) random seeds for each run. Because dynamic tools may generate different precision assignments across runs, for some benchmarks, only a subset of seeds managed to generate a precision assignment. To account for this, we compute the average running time and dynamic errors over programs generated with *all random seeds*. In contrast, sound tools are deterministic. Therefore, their optimized programs were the same across all runs. Although POPiX samples the ranges dynamically, it did not allow specifying a random seed and produced deterministic results in all 3 runs.

All optimized programs are translated to C. Daisy and Regina generate C code automatically. For FPTuner and all dynamic tools that output precision assignments in custom formats (e.g., JSON, text output in the terminal), we created scripts that generate C files with the provided mixed-precision assignment. All programs are compiled with g++ 12.2 with flags `-O2 -fPIC` and

`-lquadmath` for GCC quad precision. We additionally use `-lmpfr -lgmp` flags to compile programs that measure accuracy for MPFR and GMP libraries [29].

We measure the program's running time using C's `high_resolution_clock` and accuracy in terms of *maximum* dynamically sampled absolute error with respect to the 300-bit MPFR implementation. Both metrics are measured on 10^6 inputs uniformly sampled from the input domains specified for each benchmark. We repeat the measurement 10 times with different random seeds and compute *an average* running time and dynamic error. For accuracy, an alternative measure could be to compute the *maximum* error across 10 random runs; however, the average value provides a more realistic estimate of the program's behavior.

We consider that an optimizer *failed* if it was unable to generate a valid precision assignment within 30 minutes, or crashed with an internal error. For cases where an optimizer could not find a suitable precision assignment, we keep the original high-precision program (128 or 80 bits).

All experiments were executed on a Debian 12-bookworm machine with AMD EPYC 7662 64-Core Processor CPU @ 1.9GHz with 15.6G RAM.

4.4 Experimental Results

To answer the research questions, we designed several experiments that compare the effectiveness of static and dynamic mixed-precision optimizations. We discuss our findings below.

4.4.1 RQ1: Cost of Soundness. We start with the main question of this study, *what is the cost of soundness in mixed-precision optimizations*? To quantify the trade-off between performance improvements and (un)soundness (target error violations and under-utilization), we introduce an abstract cost function for an original program p and its optimized version p':

$$cost(p, p') = 2 \times performance(p, p') - |1 - budget\_use(p, p')|, \tag{3}$$

$$performance(p, p') = \frac{runtime(p) - runtime(p')}{runtime(p)} \tag{4}$$

$$budget\_use(p, p') = \frac{error(p')}{reference\_error(p) + target\_error(p)} \tag{5}$$

where $performance(p, p')$ is the normalized improvement in running time of the optimized program p', $budget\_use(p, p')$ is the portion of the target error used by the optimized program p', and $reference\_error(p)$ is the error of the reference implementation of p. Here, high values of $cost(p, p')$ denote effective optimization.

Since the primary goal of the mixed-precision tuning is to improve *performance*, the cost function $cost(p, p')$ assigns higher weight to performance improvements rather than accuracy changes. Changes in accuracy are expressed as the portion of the target error used by the optimized program. We define the accuracy component of the cost with the negative coefficient -1 to "penalize" optimized programs that either exceed or underutilize the allowed error budget. Specifically, $|1 - budget\_use(p, p')|$ measures deviation from the target; thus, programs that almost did not use the error budget are considered equally *bad* as those that violate the target error by 2x. This formulation avoids biasing the cost function in favor of sound tools and instead rewards optimizations that make full, efficient use of the available error margin.

Recall that for sound tools, the target error is specified with respect to ideal real-valued result and $reference\_error(p) = 0$. For dynamic tools, however, the reference implementation is already noisy, so $reference\_error(p) = error(p)$, the average dynamic error sampled for the baseline high-precision original program.

Results. To evaluate the abstract cost of soundness, we measure the average running time and dynamically sampled errors as described in the setup. Given performance and accuracy information, we compute the $cost(p, p')$ for all benchmarks and plot the results in Figure 2, higher y-values are better. We present the results using a cactus plot, where all costs are sorted in ascending order, so a point with the same x coordinate does not always correspond to the same benchmark across different tools. The results of all sound optimizers are marked with triangles and dynamic optimizers with dots. Negative values $y = -6$ denote failures of a corresponding optimizer: a crash or a 30-minute timeout, or where a measurement failed. For instance, on 3 benchmarks that compute variations of matrix determinant, we could not compute dynamic errors for all tools. Due to the high number of variables in these benchmarks, the MPFR version of the program exceeded the memory limit on our machine. For several outliers, the accuracy drop was orders of magnitude larger than the target error, which led to a large negative cost; we include such outliers into the $y = -6$ group. A cluster of points around $y = -1$ denotes optimized programs with almost no performance improvement and accuracy loss.

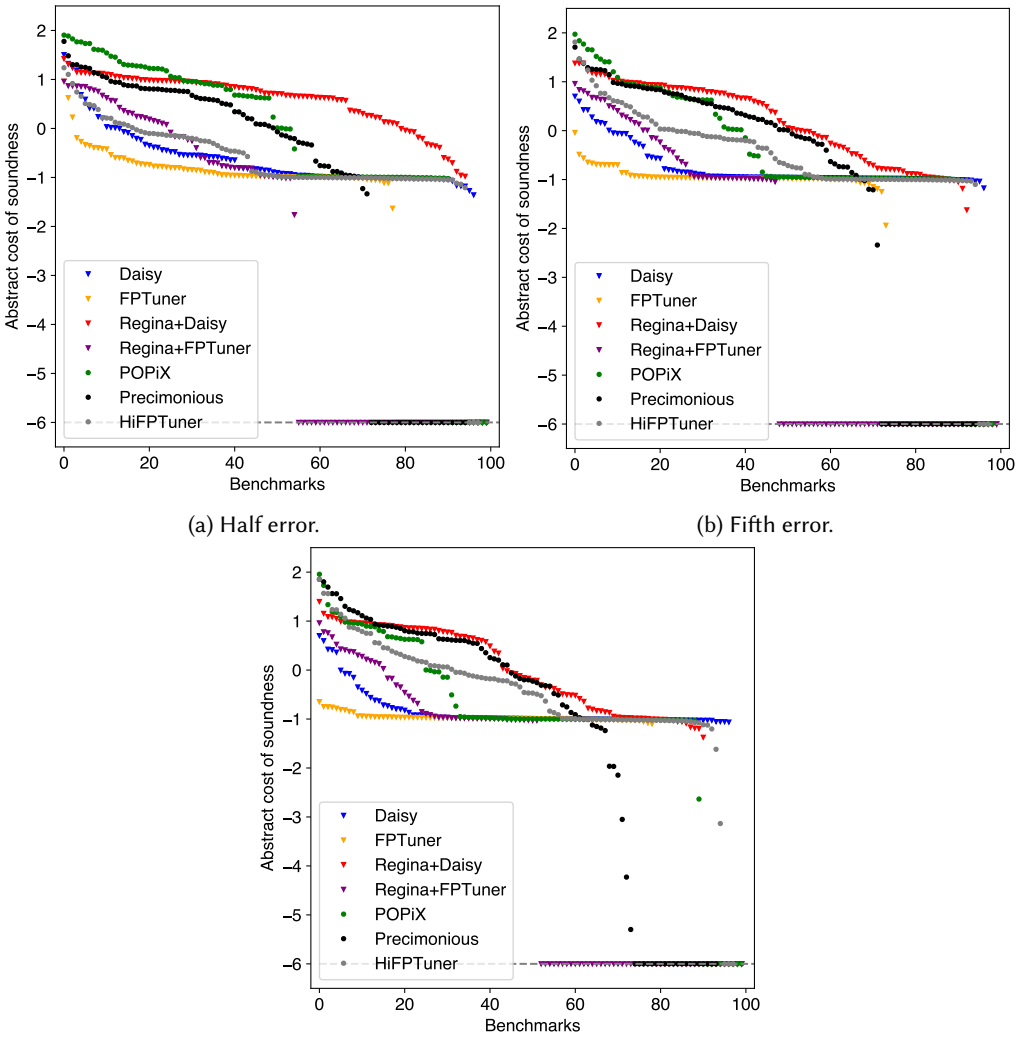
The overall trend is as expected: as the error targets tighten, *all* optimizers become less effective; static tools struggle to prove tight error bounds, and dynamic tools violate them more often. This is indicated by the steeper decline of the cost profiles in figures 2b and 2c compared to Figure 2a.

Our results show that the combination of *Regina + Daisy with interval subdivision* achieves the best cost across all targets (red triangles), outperforming all other tools on 64 out of 100 benchmarks on half error, 77 on fifth- and 65 on order errors. Dynamic tools follow with POPiX winning on 33 and 15 benchmarks on half- and fifth-error targets (green dots), and Precimonious winning on 19 order-error targets (black dots). These findings challenge the common assumption that dynamic optimizers *consistently* generate better optimized programs. Our results show that on simple programs sound tools with regime inference are more effective than dynamic tools alone.

A key factor in the success of the sound tools (Regina + Daisy) is *regime inference*, which utilizes the fact that errors are distributed unevenly across the input domain, and essentially customizes the optimization to parts of the input domain. Interestingly, Regina's regime inference combined with FPTuner has a much more modest cost profile compared to Regina + Daisy. This is due to multiple factors. First, the cost of FPTuner's programs is overall smaller than Daisy's, which is unlikely to be offset with regime inference alone. Secondly, FPTuner's slow running time leads to more timeouts and forces Regina to use a less advanced regime inference strategy. As shown in Regina's original experiments [48], the overall best regimes are inferred with the two-phase bottom-up and genetic search strategy. However, for Regina + FPTuner, this strategy timed out on more than half of the benchmarks, making the comparison infeasible. We therefore choose the one-phase top-down strategy recommended for FPTuner in Regina's original paper.

Beyond Regina, our results indicate that dynamic analysis generally worked best. Note that since Precimonious and HiFPTuner tune between 64 and 80-bit precisions, they inherently have a higher chance for effective optimizations: the difference in accuracy between 64 and 80 bits is smaller than between 64 and 128 bits (considered by other tools), which makes it easier to generate a candidate assignment that satisfies the target error.

For the half-error target, the overall range of the cost for sound and dynamic tools were comparable: all sound tools combined ranged from $[-1.76, 1.51]$, and all dynamic tools ranged from $[-1.33, 1.90]$, excluding 6 large negative outliers in range $[-4.11 \times 10^7, -1.33 \times 10^3]$ due to extreme violation of target errors. An average cost was slightly better for dynamic tools -0.03 (outliers aside), compared to the one of sound, -0.23. For smaller targets, dynamic methods' costs had larger ranges: for fifth error $[-5.30, 1.96]$ (excluding 5 outliers in $[-1.17 \times 10^8, -1.75 \times 10^5]$) and for order error $[-7.44, 1.97]$ (excluding 8 outliers in $[-2.02 \times 10^9, -8.05 \times 10^4]$); while the ranges for sound tools remained similar to the half-error target: $[-1.94, 1.38]$ for fifth- and $[-1.37, 1.40]$ for order



(a) Half error.

(b) Fifth error.

(c) Order error.

Fig. 2. Abstract cost of soundness in mixed-precision tuners expressed in terms of performance improvement and error budget use in the optimized programs (higher values are better).

error. On average, dynamic tools cost for order error was −0.13 and −0.24 for fifth-, and order-error targets, and for sound tools −0.45 and −0.56 respectively, confirming that dynamic tools are more flexible in finding fast programs, as they explore a wider range of performance-accuracy trade-off.

Answer (RQ1): The cost of soundness depends on the target error; all tools perform better on larger targets. On simple numerical programs, sound tools enhanced with regime inference (Regina + Daisy) are *just as effective or superior* to dynamic tools, achieving high performance without compromising correctness. This result shows that *soundness does not always come at a cost*. However, without regime inference, sound tools tend to be more conservative, which leads to less effective optimization, more on smaller targets, while dynamic tools are effective across all targets, but at the expense of exceeding the target error, more frequently for smaller targets.

Table 2. Maximum and average speedups achieved by programs optimized with different tools.

Tool	Half error		Fifth error		Order error	
	avg	max	avg	max	avg	max
Static						
Daisy	2.5x	79.5x	1.2x	6.6x	1.2x	6.4x
FPTuner	1.7x	54.4x	1.0x	1.8x	1.0x	1.1x
Regina +Daisy	**14.0x**	**85.8x**	9.8x	**86.7x**	**8.5x**	**86.0x**
Regina +FPTuner	4.0x	53.8x	3.5x	54.4x	2.9x	54.0x
Dynamic						
POPiX	11.0x	84.6x	**10.0x**	86.0x	8.1x	82.1x
Precimonious	5.4x	35.1x	5.2x	35.8x	5.0x	26.9x
HiFPTuner	2.1x	24.9x	2.5x	27.1x	2.2x	27.1x

To better understand how performance and accuracy changes affect our abstract cost, we evaluate these components separately in the following sections (RQ2 and RQ3).

4.4.2 RQ2: Optimized Programs' Performance. We start with evaluating the performance component, as the main objective of mixed-precision tuning is to increase programs' performance. We measure the average running time of each optimized program across 10 random runs and compute the speedup with respect to the uniform quad baseline (uniform 80-bit baseline for Precimonious and HiFPTuner) as *baseline/tool*.

We summarize the speedups per optimizer and report the results in Table 2 (the highest is marked in bold). The table shows the average and maximum speedups achieved by the programs optimized with various tools across all 100 benchmarks. When computing an average, we exclude benchmarks with timeouts and crashes, and we replace the benchmarks where a dynamic optimizer failed to generate a precision assignment with the original uniform high-precision program.

The overall best performance improvement in the optimized programs up to 86.7x has been achieved by Regina + Daisy with subdivisions for all target errors. It also achieved the highest average performance improvement for the half- and order-error targets, and ranked second for the fifth-error. Regina's regime inference allows more aggressive optimizations on parts of the input domain. When inputs are sampled uniformly (as in our experiments), such partial aggressive optimizations are enough to compensate for potentially conservative precision assignments in some input subdomains.

POPiX ranks second, improving program performance by up to 84.6x on the half-error target, 86x on fifth- and 82.1x on the order-error target. Both Regina + Daisy and POPiX rely on underlying *static* techniques for error estimation and precision tuning rather than sampling-based evaluation.

Consistent with trends observed in RQ1 — and contrary to common belief — dynamic tools do not consistently produce the best-performing optimized programs across all error targets. Both Precimonious and HiFPTuner discard candidate mixed-precision assignments that do not show immediate improvements in their dynamically measured performance scores. As a result, small fluctuations in runtime can significantly influence which configurations are kept. This aggressive pruning strategy means that potentially beneficial configurations may be prematurely discarded. Additionally, HiFPTuner explores fewer configurations than Precimonious, which speeds up

optimization time but limits its ability to find high-performing assignments, ultimately leading to lower speedups across benchmarks, as observed in our results.

Standalone sound optimizers achieve the smallest average speedup across the benchmarks, with an average speedup between 1x (no speedup) and 2.5x. This aligns with the common understanding of sound optimizers' limitations: provably sound bounds (on the whole input domain) are conservative, and they limit the number and flexibility of valid precision configurations. The extreme case is FPTuner with order target error, where the maximum achieved speedup was 1.1x and average 1.0x, showing that the target was too fine-grained for the optimizer to find precisions that improve performance. This was partially mitigated with regime inference that significantly increased the maximum speedup from 1.1x to 54x.

Answer (RQ2): The highest speedup across different target errors — up to 86.7x — was achieved by the sound combination of Regina + Daisy. Without Regina, sound tools show significantly smaller performance improvements compared to dynamic ones.

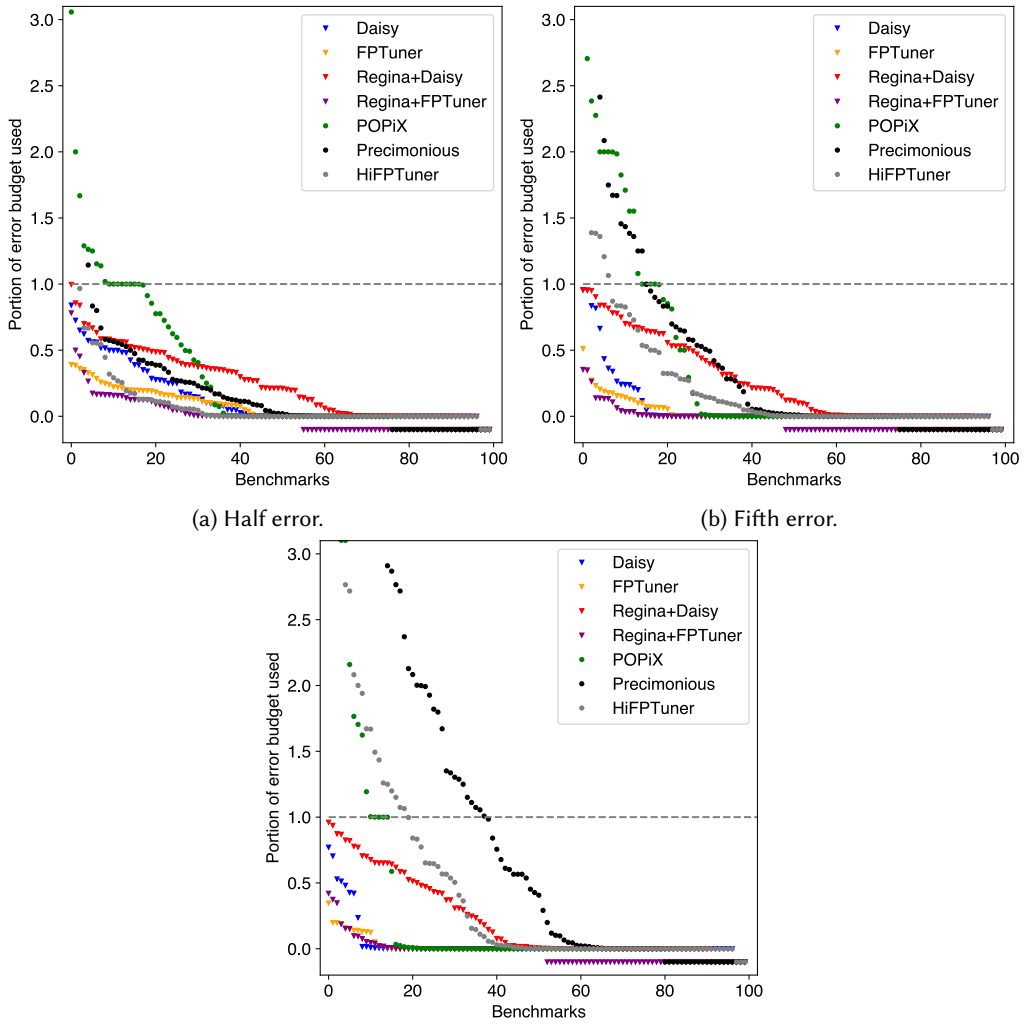
4.4.3 RQ3: Error Budget Use. To assess optimized programs' accuracy, we evaluate how close the dynamically sampled errors are to the target error, i.e., how much of the allocated error budget is actually used. Recall that the target error represents the maximum increase in overall error with respect to the reference implementation. For sound tools, the reference is the result of a real-valued algorithm, and for dynamic tools, the reference implementation is the high-precision version of the program that already includes some rounding error.

We do not compare the absolute values of the sampled errors because they alone are not representative of the optimization quality. For instance, larger errors are not necessarily better, as they may significantly overshoot the target error bound, and very small errors may indicate that the program has the original uniform high-precision assignment.

For each optimized program, we measure the maximum error occurring in one run on 10^6 input samples and then compute the average across 10 random runs. We perform the same procedure to determine the baseline errors: uniform 128-bit quad precision programs for static tools and POPiX, and 80-bit implementation for Precimonious and HiFPTuner. Recall that, for dynamic tools, the programs generated across 3 optimization runs may differ, as they rely on randomly sampled inputs during the tuning process. To account for these differences, we compute the average error over all 3×10 maximum sampled errors for dynamic tools (for sound tools only in 10, deterministic optimizers generate exactly the same optimized programs in 3 runs).

Figure 3 shows the portion of the error budget used by the optimized programs for all target settings (half, fifth and order error). We use cactus plots with sorted results, where negative values denote cases where the optimizer did not generate an optimized program or a measurement failed. Values between $0 \leq y \leq 1$ denote benchmarks that satisfy the target error. As expected, only programs optimized with dynamic tools exceed the given target. Note also that programs optimized with POPiX exceed the allocated error budget despite the static modeling of the errors. We expect this to be due to its reliance on dynamically sampled inputs when estimating the variables and operation ranges, or a bug in the implementation.

Predictably, both sound static and dynamic optimizers make better use of the larger budget (half error) than the small ones (fifth and order error). Additionally, Regina + Daisy used the budget significantly better than other sound tools due to its ability to optimize aggressively smaller parts of the input domain. The gap in accuracy between 128-bit and 64-bit precisions is large, so lowering the precision of one variable may already lead to the overall error not being satisfied. On average, across all tools, programs optimized for the half-error target were *14.3 and 12.8 orders of magnitude* less accurate than the uniform 128-bit quad and 80-bit baselines (used by Precimonious



(a) Half error.

(b) Fifth error.

(c) Order error.

Fig. 3. Used-up portion of the target error in the optimized programs. Values below $y = 1$ satisfy the target error; the closer to $y = 1$, the better.

and HiFPTuner), respectively. For the fifth-error target, the accuracy dropped by *19 and 12.6 orders of magnitude*, respectively. Similarly, for the order-error target, which in absolute values is very close to the fifth-error, the difference was *19.5 and 12.5 orders of magnitude*.

Dynamic tools failed to meet the accuracy target most frequently for order-error setting: Precimonious violated targets on 5 (half-error), 17 (fifth-error) and 40 (order-error) benchmarks, and HiFPTuner on 4, 14 and 24, respectively. POPiX exceeded the target error for 14, 18 and 15 on half-, fifth-, and order-error benchmarks. The violations occurred on 20 distinct benchmarks for half-error target, 36 for fifth- and 51 for order-error. There were no benchmarks that consistently caused all dynamic tools to fail. Predictably, most error violations occurred on benchmarks with nonlinear expressions (e.g., jetEngine) or elementary functions (e.g., complex_sine_cosine).

Table 3. Average optimization time (in seconds) and a standard deviation over 100 benchmarks. A timeout of 30 minutes is not included in the average; it is reported in a separate column #TO.

Tool	Half error		Fifth error		Order error	
	avg time, s	#TO	avg time, s	#TO	avg time, s	#TO
Static						
Daisy	8.13 ± 16.37	0	10.77 ± 25.75	0	12.42 ± 33.88	0
FPTuner	68.00 ± 126.52	7	119.37 ± 291.88	7	141.47 ± 288.17	6
Regina +Daisy	67.22 ± 189.97	3	110.97 ± 258.23	4	120.98 ± 242.38	6
Regina +FPTuner	298.27 ± 332.64	31	200.79 ± 238.08	33	266.92 ± 379.51	34
Dynamic						
POPiX	**0.30 ± 0.12**	0	**0.30 ± 0.12**	0	**0.29 ± 0.12**	0
Precimonious	104.31 ± 192.67	1	101.10 ± 203.92	1	87.33 ± 144.01	0
HiFPTuner	14.42 ± 13.91	0	12.65 ± 11.88	0	12.77 ± 10.66	0

These results further support the conclusion that the *cost of (un)soundness is higher for smaller target error*, as static tools underutilize the budget and dynamic tools improve performance by discarding the accuracy constraint more frequently.

We do not plot the errors beyond 3x use of the target error budget and note that for POPiX the outliers were up to 3.1x, 47.4x and 5.4x for half-, fifth- and order errors. For Precimonious the largest errors exceeded the target by factors of 4.1×10^7, 10^8 and 2×10^9 on half, fifth and order errors, and for HiFPTuner the factors were 3.1×10^7, 1.2×10^8 and 2.3×10^8 respectively. That said, even though such large errors are rare, a value that is 7 to 9 orders of magnitude less accurate than expected may have dramatic consequences in the software. One can reduce the likelihood of violations by increasing the number of input samples at the cost of higher optimization times. It is, however, not possible to eliminate them completely unless input samples include all possible combinations of values, which is infeasible.

Answer (RQ3): Dynamic tools frequently exceed the target error, more so for small targets, with extreme violations up to 7-9 orders of magnitude. Sound tools consistently satisfied the target error, sometimes only by keeping the original high precision. Aside from Regina, sound tools were less effective than dynamic ones in using the available accuracy budget, especially when the budget is small.

4.4.4 RQ4: Optimizers Running Time. Finally, we evaluate the optimization time of each tool by measuring its wall-clock execution. We used the standard `time` command for Daisy, Regina, Precimonious and HiFPTuner, and the difference of two time points with Scala's `System.currentTimeMillis` for POPiX and FPTuner because they required pre- and post-processing, and the command calling the tools was executed from inside Scala code. We show the average optimization times across 100 benchmarks, their standard deviation, and the number of 30-minute timeouts in Table 3. Note that optimization times for each benchmark vary greatly; therefore, the standard deviation is expected to be very high. The tool with the lowest average runtime is highlighted in bold.

A clear winner is POPiX with optimization times below one second for almost all benchmarks. It was equally fast to find an optimized program and reporting that it cannot find a solution. Though both POPiX and FPTuner assign precisions by solving an optimization problem, the problem formulation in POPiX is significantly simpler and uses more efficient encoding, and overall value

range and error modeling are less involved, which leads to a drastic difference in optimization times.

There is no consistent trend indicating that either sound or dynamic tools are universally faster. For instance, sound Daisy's optimization times are comparable to dynamic HiFPTuner's, and sound FPTuner on average takes almost as long as Precimonious, aside from a few timeouts. Regina in both combinations (Regina + Daisy and Regina + FPTuner) is predictably slower than other tools because it repeatedly invokes the underlying optimizer.

Answer (RQ4): POPiX is the fastest optimizer overall, with average optimization times less than 0.5 seconds. The running times of other tools vary significantly, and there is no clear trend indicating that either static or dynamic approaches are inherently faster. As expected, meta-optimization tools like Regina have longer runtimes due to the repeated invocation of the underlying optimizers.

5 Discussion

In this section, we reflect on key insights from our study that can help the community make informed decisions when applying mixed-precision tuning, and guide future research in this space.

Limitations of Dynamic Criteria. Our study indicated that relying solely on dynamically measured criteria has its drawbacks. For instance, using dynamic performance scores to prune precision configuration can lead to discarding promising configurations. We observed that small fluctuations in runtime may affect the choice of the final precision assignment, which makes optimization non-deterministic and, potentially, less effective. Moreover, our results suggest that a smaller search space does not necessarily lead to better optimizations. For example, while HiFPTuner explores fewer configurations than Precimonious and optimizes faster, Precimonious generated optimized programs with higher speedup (though lower accuracy).

Challenging Common Hypotheses. The main goal of this study was to go beyond the well-known trade-off between accuracy and performance by evaluating the effectiveness of sound static and unsound dynamic optimization techniques.

When starting the study, the expectation was that sound tools would *always* produce slower programs because they keep accuracy guarantees. In practice, the picture is more complex. The sound tools alone indeed provided an overall smaller speedup on the optimized programs. However, complemented with the regime inference technique, sound optimization Regina + Daisy produced faster optimized programs than dynamic tools on many benchmarks.

The big lesson learned from conducting this study is that researchers should challenge commonly used hypotheses. While some hypotheses may be timeless, others should be re-evaluated at regular intervals. The long-standing hypothesis that "sound optimizers are inherently less effective than dynamic ones" may have held true in the past— however, advancements in sound optimizations (for instance, sound regime inference) are changing this trend.

Finding Fair Baselines. Our study compares fundamentally different approaches and tools, which made it challenging to find common ground for comparison. We discuss the main design choices and alternatives here.

A key challenge was the fundamentally different view on target accuracy by static and dynamic tools. Static tools target an ideal, noise-free real-valued computation and require absolute error bounds, while dynamic tools compare against a high-precision implementation that may already include error and can handle both absolute and relative thresholds. For consistency, we used absolute error targets throughout. When available, we derived the target errors from the error

bound reported by the tool itself; otherwise, we used the largest error reported by Daisy. We believe this setup is closest to the common ground and allows for a fair treatment of the optimizers.

Another key distinction is that dynamic tools are more flexible in handling complex programs with loops and conditionals, whereas static tools are typically restricted to simple, straight-line code. This criterion is hard to quantify, since it depends heavily on the context and domain where the tools are applied. To ensure a fair comparison, we only focused on loop-free and branch-free benchmarks from FPBench [16], a widely accepted community standard that *both types of tools* can handle. Including more complex programs would have skewed the comparison and emphasized known limitations of static analyses rather than providing new insights.

However, this benchmark choice naturally limits the conclusions we can draw about the practical applicability of static tools. Many real-world programs include complex control flow that static sound mixed-precision tuners currently cannot handle. Therefore, our findings on the effectiveness of static tools may not be directly generalized to larger, more complex applications. In these cases, dynamic tools remain the practical and *the only viable option*, though they may come with significant accuracy violations. An interesting future direction would be to combine sound static analysis for numerically critical functions with dynamic techniques for the whole program (as it has been explored in verifying special values and cancellations in large floating-point programs [43]). Such a combined approach could achieve a more balanced trade-off between soundness and practicality.

80 vs 128 Bit Precision. We would like to additionally highlight the difference in high-precision baselines in our comparison. Initially, we replaced all 80-bit `long double` precision assignments generated by Precimonious and HiFPTuner with 128-bit quad precision. The assumption was that if a program is faster than the 80-bit reference, the transformed version would be faster than the 128-bit baseline. Similarly, if 80 bits offer enough accuracy, 128 bits should suffice as well. In practice, however, we noticed that while the relative metric for accuracy did hold, the speedup of mixed-precision programs with 128-bit quad type over the 128-bit uniform baseline was significantly lower than the speedup for 80-bit `long double`. The 128-bit versions of Precimonious and HiFPTuner optimized programs were on par with programs generated with sound tools.

We found this observation especially interesting, as both Precimonious and HiFPTuner cannot explicitly tune for higher than 80-bit precision, but on some newer hardware (with SSE2 instead of x87 FPU) 80-bit precision is not supported and `long double` defaults to 64 bits for many compilers. It therefore calls for the major maintenance work to update the optimizers for changing hardware.

On Practical Usability. The usability of tools varied considerably. Some, such as Daisy, Regina, and POPiX, required minimal setup — typically just input range annotations and a target error specification. Others, like FPTuner, required more extensive modifications, including rewriting programs in a less common domain-specific language (for instance, `c = a + b` in FPTuner's input language is `c = IR.BE("+", id, a, b)`, where *id* is the variable's `c` unique ID). The dynamic tools Precimonious and HiFPTuner required the most setup effort: we had to create a set of inputs, generate reference outputs, and write custom code to read pre-sampled inputs during tuning. Additionally, specifying the target error in these tools was not fixed and often required modifying the benchmark source code. While this offered flexibility, such as switching between absolute and relative errors, it also increased the potential for user error. We did not include usability as a formal comparison metric, as it is tied more to tool design than to being static or dynamic; however, it significantly impacted the effort required to perform the study.

Tool Maintenance Challenges. Unsurprisingly, many tools were difficult to install or run. This is a known issue with academic prototypes, which often lack long-term support. In some cases, resolving compatibility issues required only minor updates in the source code or installation scripts.

For instance, updating fpPrecisionTuning from Python2 involved simply adding brackets to all `print` statements. However, even after this, we could not run the tool due to other compatibility issues. Several tools relied on outdated dependencies. Precimonious and HiFPTuner, for example, required LLVM version 3.0, whereas the latest available version at the time of our study (March 2025) was LLVM 20.1. Similarly, Daisy uses pre-compiled libraries and requires Java 8 version, while the current version is Java 24. In some cases, supporting infrastructure such as build systems or package managers had been discontinued or significantly changed, further complicating installation. These issues illustrate how quickly toolchains evolve and how fragile older tools can become when not maintained. Thus, we highlight the importance of providing ready-to-run artifacts, especially for tools intended for wider or future use (as was the case for Precimonious and HiFPTuner). This supports reproducibility and allows other researchers to build upon prior work easily.

Finally, we are grateful to the developers of several tools for their responsiveness and general willingness to assist with troubleshooting when installing and running their tools.

6 Related Work

The most closely related work to ours compares static and profiling-guided (dynamic) precision tuning in the tool TAFFO [21]. However, its goals and setup differ from those in this study. While our study focuses on comparing fundamentally different optimization strategies, TAFFO evaluates different methods for *annotating* floating-point variables *with range information* — such as static expert annotations, naive input-based annotations, and profiling-based annotations — while using the same underlying precision allocation strategy. Thus, the reported speedups are similar across strategies, with variations only in the relative error. Additionally, TAFFO transforms floating-point computations into fixed-point and does not directly focus on floating-point precision tuning, aiming at performance and energy gains on hardware without FPUs.

In the following, we briefly review mixed-precision tuning approaches specifically targeting fixed-point programs, along with other optimization techniques for numerical programs.

Mixed-Precision Tuning for Fixed-Point Programs. Among reviewed tools, Daisy [17] and POPiX [8] support mixed-precision tuning for fixed-point arithmetic, and use the same precision allocation strategies described in Section 3. Alternative approaches for tuning mixed fixed-point precisions include the Min+1 or Max-1 simulation-based strategies [11], as well as a hybrid approach that combines Bayesian optimization with Min+1 [32]. Another alternative method solves an approximate optimization problem by relaxing constraints that encode word length (precision) from integers to real values [23], but such techniques are applicable only to digital signal processors (DSPs). In general, these techniques either treat errors as uncorrelated and additive or estimate them dynamically through simulation, meaning they do not provide formal soundness guarantees.

Recent work has explored fixed-point mixed-precision tuning in the context of neural networks. The tool, Aster [44], provides sound guarantees by formulating the tuning problem as a mixed-integer linear program (MILP), focusing on feed-forward networks with ReLU and linear activations. Other approaches [6, 36] support non-linear activations such as tanh or max pooling, target convolutional architectures, and aim to minimize fixed-point precision while keeping the relative error to the floating-point baseline within a specified threshold. These methods rely on dynamic range analysis and do not provide full formal guarantees. Stochastic arithmetic has also been used for estimating rounding errors in tuning [22]. Other dynamic techniques such as Seedot [27] and Shiftry [41], perform mixed fixed-point tuning by iteratively reducing the precision of variables and dynamically comparing the resulting classification accuracy with that of the original model. However, all these methods are specialized for neural networks and are orthogonal to our focus on general-purpose floating-point precision tuning.

In addition to precision tuning, another line of work explores polynomial approximations to identify suitable implementations for mathematical library functions [9, 18, 40].

Expression Rewriting. Orthogonal to improving program's performance, *rewriting* optimization improves accuracy by rewriting expressions based on algebraic laws such as distributivity and associativity. Since these laws do not hold in floating-point arithmetic, rewriting equations can reduce rounding errors. Several sound static tools have explored this direction. Daisy [17] employs a genetic search-based approach, Salsa [14] incorporates Sardana [35] to perform expression rewriting using abstract equivalence graphs.

Beyond sound techniques, several tools focus on repairing numerical programs by mitigating high rounding errors. Herbie [47], AutoRNP [58], and the tool by Wang et al. [54] employ dynamic analysis to partition the input domain, detect subdomains where floating-point errors are high, and apply targeted rewrites to reduce rounding errors. Herbie, in particular, uses rules based on polynomial approximations and equality saturation via the egg library [56], while other tools derive approximations using handwritten specialized rewrite rules. Note that these dynamic optimizers repair accuracy by using expressions syntactically different from the original programs.

Efforts have also been made to combine sound and unsound optimizers, specifically Daisy and Herbie [5]. In this work, Daisy serves as a backend validator for Herbie's optimizations, revealing limitations in both tools for floating-point rewriting optimization.

In addition to fully automated approaches, Odyssey [45] introduces an interactive mode, allowing users to identify problematic inputs using dynamic analysis, refine input domains in real time, and fine-tune optimized expressions. This user-guided approach provides flexibility in improving numerical accuracy while retaining control over program modifications.

Combined Approaches. Some tools combine multiple optimization strategies to improve both numerical accuracy and performance. Daisy and Salsa [15] follow static approaches that integrate expression rewriting with mixed-precision tuning, whereas PHerbie [51] (built on and now merged into Herbie) uses a dynamic analysis technique. Both Daisy and Salsa begin by rewriting arithmetic expressions into mathematically equivalent but more numerically stable forms. Daisy then applies mixed-precision tuning as described in Section 3.2.2, while Salsa uses a combination of forward and backward data-flow analyses to determine the minimal precision required at each program point to meet user-specified accuracy targets. It then encodes the constraints into positional logic formulas and relations between affine expressions and solves them using state-of-the-art SMT solvers to produce a precision assignment.

In contrast, PHerbie interleaves expression rewriting with precision tuning during its dynamic analysis. This allows it to optimize accuracy and performance simultaneously throughout the tuning process. However, like Herbie, Pherbie does not fully preserve real-valued semantics. As rewriting is not the focus of this paper, we did not include combined approaches in our comparison.

7 Threats to Validity

Our experimental evaluation involves randomness in several aspects: sampled inputs for optimizers, sampled errors for comparing the accuracy of optimized programs, and fluctuations in running time. We account for these factors by using sufficiently many random seeds: 3 for running optimizers, 10 for evaluating performance and accuracy of optimized programs. To account for differences in the sizes of errors the optimizers can prove or check, we use the relative metric of a portion of the used-up target. Similarly, to account for differences in 128-bit and 80-bit baseline high precision running time, we measure relative performance improvement compared to the corresponding baseline.

While some tools were excluded due to being closed-source or requiring extensive, undocumented modifications, we carefully selected and evaluated, to the best of our knowledge, all publicly available tools that met our criteria, as detailed in Section 4.1. We followed official documentation and, where required, guidance from the authors and used default or recommended configurations for typical usage. While deeper tool-specific tuning might yield marginal improvements, our setup ensures a fair and representative comparison.

8 Conclusion

Mixed-precision tuning is a widely applied optimization that balances performance and accuracy in numerical programs. While numerous tools exist — ranging from sound static analyzers to efficient but unsound dynamic techniques — a systematic and quantitative comparison of their trade-offs has not been considered before. This paper presents the first comprehensive evaluation of mixed-precision optimizers for floating-point programs that both sound and dynamic tools can handle, introducing a quantitative measure for the cost of soundness. Our results show that on simple programs sound tools—especially when enhanced with meta-optimizations like regime inference—are competitive and often outperform dynamic tools without compromising correctness guarantees. Contrary to common belief, *soundness does not inherently sacrifice performance* for programs with sufficient optimization potential, though it may come at the cost of longer optimization time. For applications that fall outside the scope of current static analyses—complex control flow, loops, or large-scale code typical of HPC or scientific computing—dynamic tuning remains the practical, and often the only choice, albeit at the risk of significantly violating accuracy constraints. We hope this work will serve as a guide for researchers and practitioners navigating the broad and complex landscape of mixed-precision tuning for numerical programs.

9 Data Availability Statement

We provide an artifact for our experimental study via Zenodo [38]. The artifact contains the experimental data results and the scripts we used to run the tools and compare and plot the results. The data includes the original benchmarks, optimized programs for every tool, and their measured running times and dynamic errors.

Due to the substantial effort required to install various tools on the machine where we performed the experiments, we do not see it as feasible to repeat this effort on a virtual machine. We note, however, that Precimonious and HiFPTuner provide the Docker image with the tools pre-installed. It can be pulled with the command: `docker pull ucdavisplse/precision-tuning`.

Acknowledgments

This work was partially supported by funding from the pilot program for Core Informatics of the Helmholtz Association (HGF). We also thank Dorra Ben Khalifa, Ganesh Gopalakrishnan, and Minh Ho for their help in troubleshooting the installation process of their tools.

References

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.
[2] 2020. *GCC libquadmath*. https://gcc.gnu.org/onlinedocs/libquadmath/
[3] Accessed: 2025-03-20. Fixmath: A Library of Fixed-Point Math Operations and Functions. https://www.nongnu.org/fixmath/doc/index.html.
[4] Mark S. Baranowski and Ian Briggs. [n. d.]. Gelpia: Global Extrema Locator Parallelization for Interval Arithmetic.
[5] Heiko Becker, Pavel Panchekha, Eva Darulova, and Zachary Tatlock. 2018. Combining Tools for Optimization and Analysis of Floating-Point Computations. In *Formal Methods (FM)*. https://doi.org/10.1007/978-3-319-95582-7_21

[6] Dorra Ben Khalifa and Matthieu Martel. 2024. Efficient Implementation of Neural Networks Usual Layers on Fixed-Point Architectures. In *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. https://doi.org/10.1145/3652032.3657578

[7] Dorra Ben Khalifa, Matthieu Martel, and Assalé Adjé. 2019. POP: A Tuning Assistant for Mixed-Precision Floating-Point Computations. In *Formal Techniques for Safety-Critical Systems (FTSCSs)*. https://doi.org/10.1007/978-3-030-46902-3_5

[8] Sofiane Bessaï, Dorra Ben Khalifa, Hanane Benmaghnia, and Matthieu Martel. 2022. Fixed-Point Code Synthesis Based on Constraint Generation. In *Design and Architecture for Signal and Image Processing (DASIP)*. https://doi.org/10.1007/978-3-031-12748-9_9

[9] Ian Briggs and Pavel Panchekha. 2022. Choosing Mathematical Function Implementations for Speed and Accuracy. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3519939.3523452

[10] Hugo Brunie, Costin Iancu, Khaled Z. Ibrahim, Philip Brisk, and Brandon Cook. 2020. Tuning Floating-Point Precision Using Dynamic Program Information and Temporal Locality. In *High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1109/SC41405.2020.00054

[11] M-A Cantin, Yvon Savaria, D Prodanos, and Pierre Lavoie. 2001. An Automatic Word Length Determination Method. In *International Symposium on Circuits and Systems (ISCAS)*. https://doi.org/10.1109/ISCAS.2001.921982

[12] Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. 2021. Architecture-aware Precision Tuning with Multiple Number Representation Systems. In *Design Automation Conference (DAC)*. https://doi.org/10.1109/DAC18074.2021.9586303

[13] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-Point Mixed-Precision Tuning. *ACM SIGPLAN Notices* 52, 1 (2017), 300–315. https://doi.org/10.1145/3009837.3009846

[14] Nasrine Damouche and Matthieu Martel. 2017. Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. In *AFM@ NFM*. https://doi.org/10.29007/j2fd

[15] Nasrine Damouche and Matthieu Martel. 2018. Mixed Precision Tuning with Salsa. In *Pervasive and Embedded Computing and Communication Systems (PECCS)*. https://doi.org/10.5220/0006915501850194

[16] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. 2016. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *Numerical Software Verification (NSV)*. https://doi.org/10.1007/978-3-319-54292-8_6

[17] Eva Darulova, Einar Horn, and Saksham Sharma. 2018. Sound Mixed-Precision Optimization with Rewriting. In *International Conference on Cyber-Physical Systems (ICCPS)*. https://doi.org/10.1109/ICCPS.2018.00028

[18] Eva Darulova and Anastasia Volkova. 2019. Sound Approximation of Programs with Elementary Functions. In *Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-030-25543-5_11

[19] A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, and P. Panchekha. 2020. Scalable yet Rigorous Floating-Point Error Analysis. In *High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1109/SC41405.2020.00055

[20] Luiz Henrique De Figueiredo and Jorge Stolfi. 2004. Affine Arithmetic: Concepts and Applications. *Numerical algorithms* 37 (2004), 147–158. https://doi.org/10.1023/B:NUMA.0000049462.70970.B6

[21] Lev Denisov, Gabriele Magnani, Daniele Cattaneo, Giovanni Agosta, and Stefano Cherubin. 2024. The Impact of Profiling Versus Static Analysis in Precision Tuning. *IEEE Access* 12 (2024), 69475–69487. https://doi.org/10.1109/ACCESS.2024.3401831

[22] Quentin Ferro, Stef Graillat, Thibault Hilaire, Fabienne Jézéquel, and Basile Lewandowski. 2022. Neural Network Precision Tuning Using Stochastic Arithmetic. In *International Workshop on Numerical Software Verification (NSV)*. https://doi.org/10.1007/978-3-031-21222-2_10

[23] Paul D Fiore. 2008. Efficient Approximate Wordlength Optimization. *IEEE Trans. Comput.* 57, 11 (2008). https://doi.org/10.1109/TC.2008.87

[24] GNU Free Software Foundation. Accessed: 2025-03-20. GLPK (GNU Linear Programming Kit). https://www.gnu.org/software/glpk/.

[25] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-Precision Binary Floating-Point Library With Correct Rounding. *ACM Transactions on Mathematical Software (TOMS)* 33, 2 (2007), 13–es. https://doi.org/10.1145/1236463.1236468

[26] R. Garcia, C. Michel, and M. Rueher. 2020. A Branch-and-bound Algorithm to Rigorously Enclose the Round-Off Errors. In *Principles and Practice of Constraint Programming (CP)*. https://doi.org/10.1007/978-3-030-58475-7_37

[27] Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019. Compiling KB-Sized Machine Learning Models to Tiny IoT Devices. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3314221.3314597

[28] E. Goubault and S. Putot. 2011. Static Analysis of Finite Precision Computations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. https://doi.org/10.1007/978-3-642-18275-4_17

[29] Torbjörn Granlund and the GMP development team. 2012. GNU MP: The GNU Multiple Precision Arithmetic Library. http://gmplib.org/.
[30] Hui Guo and Cindy Rubio-González. 2018. Exploiting Community Structure for Floating-Point Precision Tuning. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3213846.3213862
[31] Gurobi Optimization, LLC. 2016. *Gurobi Optimizer Reference Manual*. https://www.gurobi.com.
[32] Van-Phu Ha and Olivier Sentieys. 2021. Leveraging Bayesian Optimization to Speed Up Automatic Precision Tuning. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. https://doi.org/10.23919/DATE51398.2021.9474209
[33] Timothy Hickey, Qun Ju, and Maarten H Van Emden. 2001. Interval Arithmetic: From Principles to Implementation. *Journal of the ACM (JACM)* 48, 5 (2001), 1038–1068. https://doi.org/10.1145/502102.502106
[34] Nhut-Minh Ho, Elavarasi Manogaran, Weng-Fai Wong, and Asha Anoosheh. 2017. Efficient Floating Point Precision Tuning for Approximate Computing. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. https://doi.org/10.1109/ASPDAC.2017.7858297
[35] Arnault Ioualalen and Matthieu Martel. 2012. Sardana: an Automatic Tool for Numerical Accuracy Optimization. In *SCAN: Scientific Computing, Computer Arithmetic and Validated Numerics*.
[36] Arnault Ioualalen and Matthieu Martel. 2019. Neural Network Precision Tuning. In *International Conference (QEST)*. https://doi.org/10.1007/978-3-030-30281-8_8
[37] Anastasia Isychev and Eva Darulova. 2023. Scaling up Roundoff Analysis of Functional Data Structure Programs. In *Static Analysis Symposium (SAS)*. https://doi.org/10.1007/978-3-031-44245-2_17
[38] Anastasia Isychev and Debasmita Lohar. 2025. Artifact for the Paper "Cost of Soundness in Mixed-Precision Tuning", OOPSLA'25. https://doi.org/10.5281/zenodo.16915563
[39] Anastasiia Izycheva and Eva Darulova. 2017. On Sound Relative Error Bounds for Floating-Point Arithmetic. In *Formal Methods in Computer Aided Design (FMCAD)*. https://doi.org/10.23919/FMCAD.2017.8102236
[40] Anastasiia Izycheva, Eva Darulova, and Helmut Seidl. 2019. Synthesizing Efficient Low-Precision Kernels. In *Automated Technology for Verification and Analysis (ATVA)*. https://doi.org/10.1007/978-3-030-31784-3_17
[41] Aayan Kumar, Vivek Seshadri, and Rahul Sharma. 2020. Shiftry: RNN Inference in 2KB of RAM. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428250
[42] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2004.1281665
[43] Debasmita Lohar, Clothilde Jeangoudoux, Joshua Sobel, Eva Darulova, and Maria Christakis. 2021. A Two-Phase Approach for Conditional Floating-Point Verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/978-3-030-72013-1_3
[44] Debasmita Lohar, Clothilde Jeangoudoux, Anastasia Volkova, and Eva Darulova. 2023. Sound Mixed Fixed-Point Quantization of Neural Networks. *ACM Transactions on Embedded Computing Systems* 22, 5s (2023). https://doi.org/10.1145/3609118
[45] Edward Misback, Caleb C Chan, Brett Saiki, Eunice Jun, Zachary Tatlock, and Pavel Panchekha. 2023. Odyssey: An Interactive Workbench for Expert-Driven Floating-Point Expression Rewriting. In *User Interface Software and Technology (UIST)*. https://doi.org/10.1145/3586183.3606819
[46] Roweida Mohammed, Jumanah Rawashdeh, and Malak Abdullah. 2020. Machine Learning with Oversampling and Undersampling Techniques: Overview Study and Experimental Results. In *International Conference on Information and Communication Systems (ICICS)*. https://doi.org/10.1109/ICICS49469.2020.239556
[47] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. *Acm Sigplan Notices* 50, 6 (2015). https://doi.org/10.1145/2737924.2737959
[48] Robert Rabe, Anastasiia Izycheva, and Eva Darulova. 2021. Regime Inference for Sound Floating-Point Optimizations. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5s (2021). https://doi.org/10.1145/3477012
[49] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. 2016. Floating-Point Precision Tuning Using Blame Analysis. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2884781.2884850
[50] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning Assistant for Floating-Point Precision. In *High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/2503210.2503296
[51] Brett Saiki, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. 2021. Combining Precision Tuning and Rewriting. In *Computer Arithmetic (ARITH)*.
[52] Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 1 (2018), 1–39. https://doi.org/10.1145/3230733
[53] Laura Titolo, Marco A. Feliú, Mariano Moscato, and César A. Muñoz. 2018. An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. In *Verification, Model Checking, and Abstract Interpretation*

(VMCAI). https://doi.org/10.1007/978-3-319-73721-8_24

[54] Xie Wang, Huaijin Wang, Zhendong Su, Enyi Tang, Xin Chen, Weijun Shen, Zhenyu Chen, Linzhang Wang, Xianpei Zhang, and Xuandong Li. 2019. Global Optimization of Numerical Programs Via Prioritized Stochastic Algebraic Transformations. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2019.00116

[55] Yutong Wang and Cindy Rubio-González. 2024. Predicting Performance and Accuracy of Mixed-Precision Programs for Precision Tuning. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3597503.3623338

[56] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

[57] Artem Yadrov and Pavel Panchekha. 2024. Fast Real Evaluation Through Sound Mixed-Precision Tuning. arXiv:2410.07468

[58] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient Automated Repair of High Floating-Point Errors in Numerical Libraries. *ACM on Programming Languages* 3, POPL (2019). https://doi.org/10.1145/3290369

[59] A. Zeller and R. Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[60] Abraham Ziv. 1991. Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit. *ACM Trans. Math. Softw.* 17, 3 (1991), 410–423. https://doi.org/10.1145/114697.116813

[61] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2019. Detecting Floating-Point Errors via Atomic Conditions. *Proc. ACM Program. Lang.* 4, POPL (2019). https://doi.org/10.1145/3371128

Received 2025-03-26; accepted 2025-08-12